Rex: A Toolset for Reproducing Software Experiments

Somu Perianayagam, Gregory R. Andrews, John H. Hartman
Department of Computer Science, University of Arizona
{somu,greg,jhh}@cs.arizona.edu

Abstract

Being able to reproduce experiments is the cornerstone of the scientific method. Software experiments are hard to reproduce even if identical hardware is available because external data sets could have changed, software used in the original experiment may be unavailable, or the input parameters for the experiment may not be documented. This paper presents Rex, a toolset that allows one to record an experiment and archive its apparatus, replay an experiment, conduct new experiments, and compare differences between experiments. The execution time overhead of recording experiments is on average about 1.6% and the space overhead of archiving an experiment ranges from 5 to 7 GB.

1. Introduction

A scientific wet-lab experiment is said to be *reproducible* if one can reconstruct the experiment's apparatus then conduct the original experiment and get identical results. Analogously, a software experiment is reproducible if one can reconstruct the original experimental apparatus and reproduce the results of the prior experiment. The benefits of reproducing an experiment are:

- The results of the original experiment can be independently verified.
- The reproduced apparatus can then be used to conduct new experiments.

A software experiment is hard to reproduce because over time it becomes impossible to reconstruct the apparatus, external data sources might have changed, or aspects of the experiment, such as input parameters, may not be documented. One way to reconstruct an apparatus is to download and compile source programs, but over time the right version of the compiler or libraries may not be available. Another way to reconstruct an apparatus is to install it from binaries, but again the right libraries may not be available. Even if the apparatus can be reconstructed, there is no guarantee that an experiment will read the same input if it comes from an active source such as a database or a web server. Finally, if the command-line arguments or other input parameters are not specified, which often happens for descriptions of experiments in published papers, then again an experiment cannot be reproduced.

Current systems for reproducing software experiments are tailored for a specific subset of experiments, address a specific problem such as debugging, or reproduce only the data analysis used in an experiment. RA[15] is a toolkit that helps one organize and reproduce software experiments that are written in Java. Tornado[6], iDNA[4], R2[12], and ReVirt[14] are systems that support replaying programs for debugging purposes. Madagascar[1] and BioConductor [11] support reproducing the data analysis part of a published experiment and tie it closely to the publication itself[10, 5].

This paper presents Rex, a set of tools that allows one to reproduce any software experiment. Rex does not require access to source code or compilers, and it does not require modifying either an experiment or the operating system on which it executes. The Rex tools are:

- `record`, which archives an experiment and its apparatus;
- `replay`, which reproduces an archived experiment;
- `runnew`, which runs a new experiment using an archived apparatus; and
- `expdiff`, which compares two recorded experiments and reports differences between them.

Rex can handle sequential, multi-process, and multi-threaded programs. It reproduces the results of a recorded experiment as long as the experiment has deterministic output behavior—i.e., it produces the same output given the same input. Rex is also able to reproduce the behavior of stochastic programs that generate random numbers from system sources such as `/dev/random`. Rex imposes only a small execution overhead for recording, replaying, and running new experiments.

The remainder of the paper is organized as follows. Section 2 describes the Rex tools and presents three use cases that illustrate how they can be used. Section 3 describes how the tools are implemented. Section 4 presents performance results. Section 5 discusses related work, and Section 6 gives concluding remarks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00

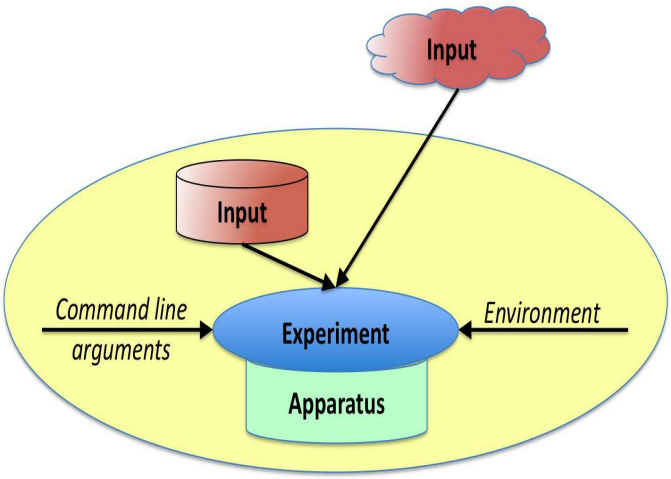

Figure 1. Archive of an Experiment

2. Rex Tools and Use Cases

This section defines the four Rex tools and gives examples of their use. The unifying concept is that of an *archive*, which contains everything necessary to reproduce an experiment: its apparatus, input, and output. Rex reproduces an experiment by reconstructing an identical apparatus, starting the experiment in the same environment and with the same command line parameters, and feeding it the same input. Figure 1 gives an overview.

The apparatus of an experiment consists of all application programs and the libraries or helper applications that the applications require.[1] (We assume that the operating system on which an experiment will be reproduced is identical to the one on which the original experiment was run.) The input to an experiment is the command-line arguments, data read from input sources such as files and servers, and data read from the operating system (such as the time of day or pseudo-random numbers). The output of an experiment is all data written to terminals, files, or network sockets.

2.1 The Rex Tools

The `record` tool archives an experiment. First it records the environment variables and input parameters. Then it starts the experiment and monitors its execution to determine the applications it uses, the input it reads, and the output it produces. An experiment is recorded by executing

record [−l<archive name>] experiment <input parameters>

The experiment and input parameters are those of the original experiment. The `-l` option specifies a name for the archive that will be created; its default value is `__default`.

The `replay` tool takes an archived experiment as input, sets up the environment for the experiment, and starts the experiment with the recorded input parameters. As the experiment executes, `replay` intercepts all input requests and feeds recorded input data back to the experiment. A recorded experiment is replayed by executing

replay [−l<archive name>] [−e<expname>] [−o<output directory>]

The `-l` option specifies the archive containing the recorded experiment; the default value is `__default`. The `-e` option specifies which experiment in the archive to replay.[2] The output from the replayed experiment is stored in the output directory specified using the `-o` option; the default is `__out_default`.

The `runnew` tool allows a user to select an experiment from an archive; make changes to the recorded experiment, such as changing input parameters, environment variables, or input files; and then run a new experiment. The user can optionally record the new experiment, in which case it is added to the same archive. Runnew is started by executing

runnew [−l <archive name>]

where the `-l` option specifies the archive to use for running new experiments (the default is `__default`). Invoking `runnew` pops up a graphical user interface (GUI) that allows a user to select an experiment, make changes (e.g., to the command-line arguments or input sources), and then execute (and optionally record) the new experiment. An example is given in Figure 3 and described below.

The final tool, `expdiff`, compares two archived experiments and reports differences between them, including differences in their environments, input parameters, inputs, outputs, and apparatuses. The `expdiff` tool is invoked by executing

expdiff [−d number] archive1:exp1 archive2:exp2

The arguments are the archives and experiments within those archives that are to be compared. Expdiff also has different levels of reporting, which are controlled by the `-d` option (see Section 3 for details).

2.2 Use Cases

This section describes three kinds of experiments, explains why they are hard to reproduce, and illustrates how Rex can be used to reproduce them. Two experiments come from computational life sciences; one is a multi-process application and the other is a multi-threaded application. The third example is a network simulation experiment from computer science.

2.2.1 Basic Local Alignment Search Tool (BLAST)

BLAST is used in bio-informatics to identify the best possible matches of an input protein or nucleotide sequence

[1] Perl and Python interpreters are examples of helper applications.

[2] An archive can contain multiple experiments as a result of using the `runnew` tool described below.

in a sequence database. The tool implements a heuristic algorithm [2] that performs matches. BLAST takes as input a file containing input sequences, a database that has to be searched, and input parameters that govern the heuristics of the search algorithm.

A typical BLAST experiment is a two step process. First, the input database is converted from the FASTA format [3] to a protein or nucleotide database format (commonly known as a BLASTable format) using a tool called `formatdb`. Then, the input sequences are "blasted" against the database to find similar sequences. A scientist who wants to reproduce the results of a BLAST experiment first has to install the BLAST tools, download the input sequences used in the original experiment, and then blast them against the original database. The scientist may not be able to reproduce the experiment because:

- The BLAST algorithm could have been updated.
- The input database could have changed (gene databases are updated periodically).
- One or more components of the experiment could have changed. For example, BLAST programs of version 2.2.9 and higher cannot use databases formatted using earlier versions
- The input parameters may have not been recorded by the person who performed the original experiment.

With Rex one can record the BLAST experiment and archive its apparatus by executing

record –IBLASTEXP blast in.seq in.db blastn

The experiment can later be reproduced by downloading the archive (`BLASTEXP`) and using Rex's `replay` tool as follows:

replay –IBLASTEXP –oBLASTOUT

Here the output of the reproduced experiment will be stored in the `BLASTOUT` folder.

One can then use the `runnew` tool to run a new experiment that is based on the one in the `BLASTEXP` archive:

runnew –IBLASTEXP

This reads the archive and pops up a GUI as shown in Figure 2. The GUI initially presents the user with information about the first experiment recorded in the archive: the name of the experiment, its command line, and its apparatus. The user can change the experiment by using the the drop-box to choose a different experiment, in which case information about that experiment will be displayed.

The user can modify the command line or apparatus to create a new experiment. Figure 3 shows how one could modify the above experiment. The command line has been changed to reflect that the new experiment should run the `blastp` program. Also, the blastall program has been changed to a new version located in a different place. The `Record` box is checked to indicate that the new experiment

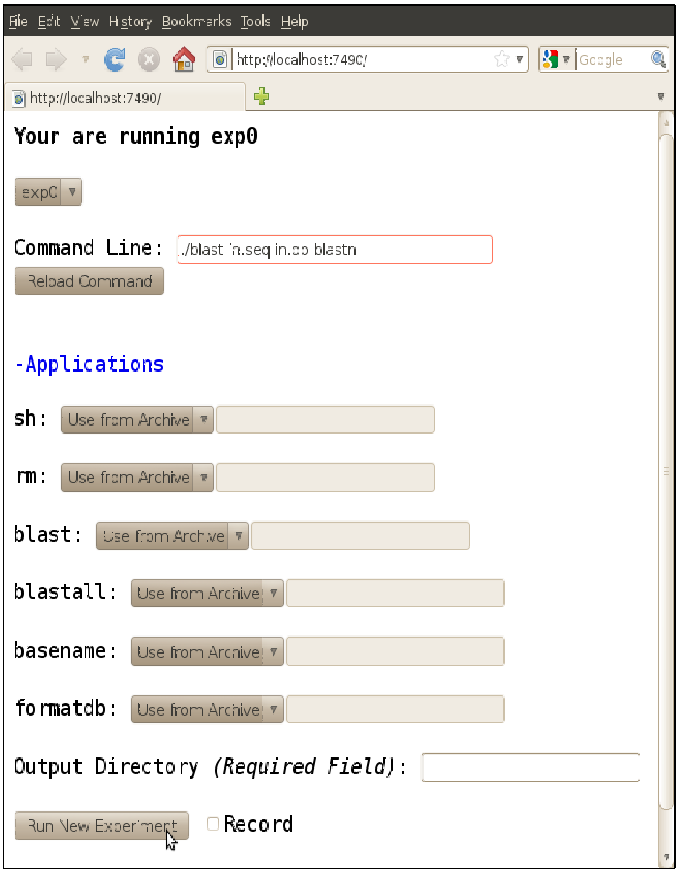

Figure 2. Running a New BLAST Experiment

should be recorded. Finally, `blastpout` has been specified as the location for the new output directory.

When the user clicks the `Runnew` button, the screen shown in Figure 4 pops up to ask the user to confirm the changes. After confirmation, the new experiment is run and and recorded into the `BLASTEXP` archive.

Since the new experiment is recorded, `expdiff` can be used to compare the original and modified experiments by executing:

expdiff BLASTEXP:exp0 BLASTEXP:exp1

Here `expdiff` will report all differences between the blast binaries, the command line arguments, the input databases, the input sequences, and the inputs read from the system.

2.2.2 Runassembly from the Roche 454 Software Suite

Runassembly is a tool that is part of the Roche 454 software suite [19]. It assembles short DNA sequences into large sequences by computing overlaps between them. The output of an assembly depends on the degree of parallelism used by the tool, which in turn depends on the workload of the system. Though the outputs may differ, both are correct because there is no unique answer to the assembly problem.

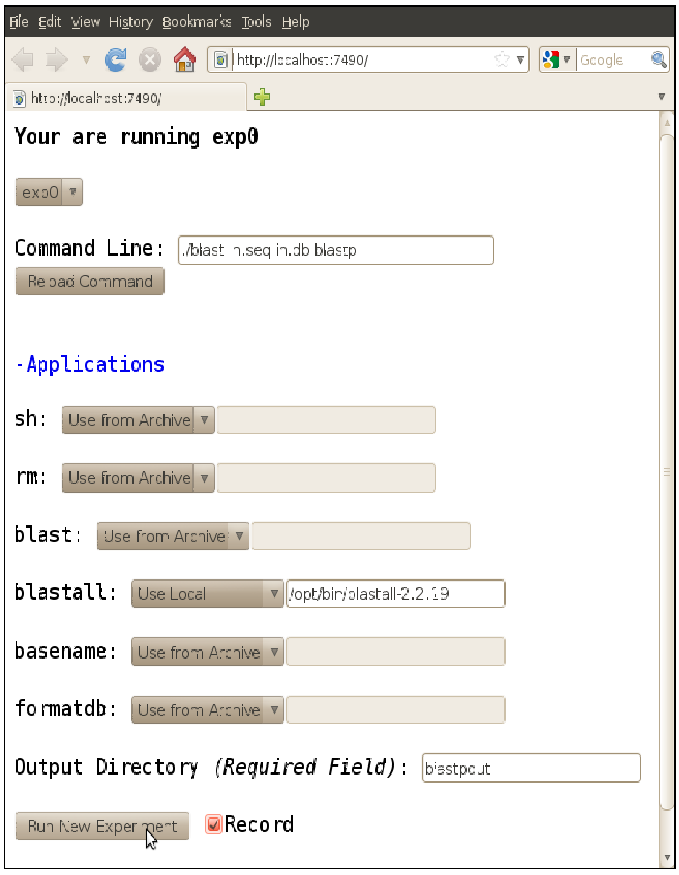

Figure 3. Making Changes to a BLAST Experiment

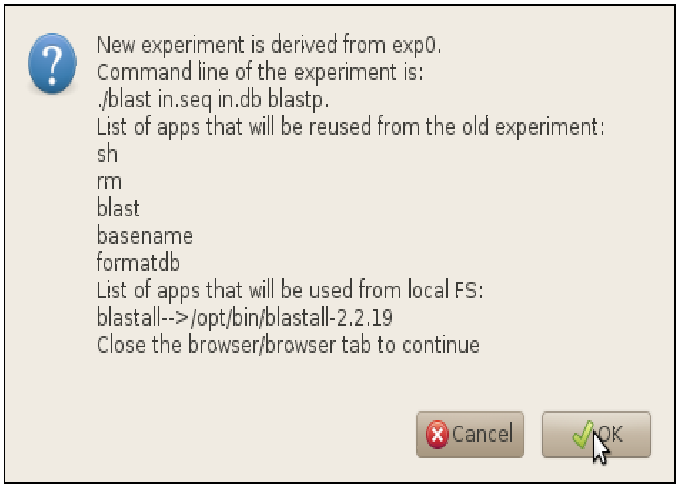

Figure 4. Summarizing the New BLAST Experiment

However, with the same degree of parallelism, the tool produces the same output.

Using Rex, runassembly can be recorded by executing:

```
record –IASSEXP runAssembly –nobig –o outdir
    in.sff
```

The recorded experiment can then be replayed by executing:

```
replay –IASSEXP
```

When the experiment is replayed, runassembly will be fed the recorded CPU workload; hence, it will use the same degree of parallelism and produce the same output as the recorded experiment.

2.2.3 Global Mobile Information System Simulation (GloMoSim): A Network Simulator

GloMoSim[20] is a scalable simulation environment for large wired and wireless communication networks. It is implemented as a parallel discrete event simulator. GloMoSim can simulate large networks that can scale up to thousands of nodes linked by heterogeneous network stacks, such as asymmetric communications using direct satellite broadcasts, multi-hop wireless communications using adhoc networking, and traditional Internet protocols.

The input parameters for a GloMoSim simulation are specified using a configuration file. The parameters include the number of nodes, the topology of network, the movement of of the nodes within the topology, the protocols, and transmission noise. The results of a simulation will vary if the topology of the network or the movements of the nodes is set to `RANDOM`. Randomness is controlled using a seed that is also specified as a part of the configuration file.

A publication reporting the results of a simulation may fail to describe all the configuration file setting such as the random number seed. With Rex, a network simulation can be recorded and archived as follows:

```
record –ISIMEXP glomosim config.in
```

Generally, network simulations that use random numbers are run multiple times with different seeds to assure consistency in results. Running a simulation with a different random number seed is equivalent to running a new experiment with the same apparatus but a different configuration file. To run a new experiment using Rex, one would edit the configuration file, and then use the `runnew` tool with an archived experiment and the new configuration file. If the new experiment is recorded, it will be added to the archive. If the archive is distributed along with published results, readers of the paper can use Rex to replay, and hence validate, the experiments contained in the archive.

3. Implementation of Rex

This section describes the implementation of Rex tools. The tools all work with an archive, which contains all

the information needed to reproduce an experiment: the apparatus of an experiment, input read, output written, environment variables, command-line arguments, and a log of all system calls made when the experiment was recorded. Rex gathers information about an experiment by monitoring the systems calls that it makes, because it is through system calls that a program reads input, writes output, and loads application programs and libraries.

When Rex records a program, it creates a system call log that is stored in the archive; it uses the log when it replays a program to determine what the program did when it was recorded. System call monitoring is implemented using the Linux `ptrace` mechanism, which does not require modifying the application programs and imposes only a small overhead.

3.1 The Archive

An archive is created when the `record` tool is used. One is augmented when a new experiment is run using an existing archive (and the experiment is recorded). The archive has repositories for the apparatus, input/output files, and system call logs, and a directory of experiments.

The apparatus repository contains the applications and libraries used by all the experiments stored within the archive. Similarly, the input/output and system call log repositories contain the I/O files and system call logs, respectively, for all the experiments in the archive. Within each repository, version numbers are used to resolve naming conflicts that would, for example, occur when the same file is used by more than one experiment.

There is a folder for every experiment in the archive. It contains the experiment's environment variables and command-line arguments as well as meta data that describes the experiment's apparatus, I/O files, and system call log and indicates where they are stored. One experiment is the default experiment; initially this is the oldest experiment, but the user can change this.

3.2 The `record` Tool

When `record` is executed, it creates an archive and then stores the environment and command-line arguments of an experiment in it. The `record` tool then executes the experiment under its control so that it can use the `ptrace` mechanism to trap the entry and exit points of the system calls made by the experiment [13]. The `ptrace` mechanism requires Rex to trap all system calls; it cannot selectively trap just those in which it is interested. Rex takes the actions described below for the system calls in which it is interested; for others, Rex simply lets the call execute as it normally would. Figure 5 gives an overview of `record`.

Rex determines the applications used by an experiment by monitoring the `exec` system call. Some applications might be dependent on other applications; for example, a perl script needs a Perl interpreter in order to execute. The `exec` system call shows only the script being executed. The

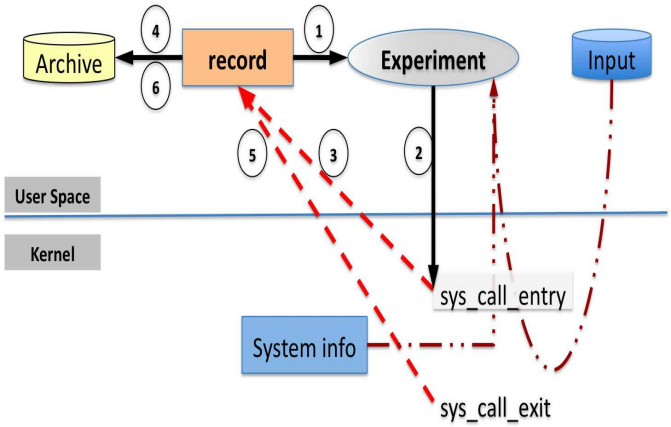

Figure 5. The `record` Tool

operating system infers that the script is not a binary and invokes the appropriate interpreter to execute the script with its input arguments. After the experiment has finished running, Rex determines that the script used a Perl interpreter (by reading file headers) and adds the interpreter to the apparatus.

While an experiment is running, Rex monitors the libraries opened by an application and archives them. This ensures that Rex archives any libraries opened using interfaces such as `dlopen` to the dynamic linking loader. After an experiment has finished running, Rex statically resolves the libraries required by all the applications that are part of the apparatus of the experiment, and it archives those that are not already in the archive. This ensures that Rex does not miss any libraries that *might* be required but that were not loaded during this run of the experiment.

The `record` tool determines the input seen by an experiment by logging all system calls that provide input. The system calls that provide input are those that read from input sources such as files, network sockets, pipes, and local system data. Some of these calls—such as `read`, `pread`, and `stat`—provide input by copying it into a user provided buffer. Others—such as `getpid` and `getuid`—provide input by means of the return value.

The file based memory map system call, `mmap`, is another one that provides input. Rex logs the call as well as the contents of the file that is mapped into a memory region. Rex maintains a tuple for each memory-mapped file in order to avoid storing duplicate copies of the same file contents. The tuple has five fields: file name, file offset, length, bytes, and log offset. Each time a file is memory-mapped, Rex checks whether an identical call for the same file was performed previously and whether the mapped contents of the file

have not changed since the last call. If the file has been mapped and the contents have not changed, Rex stores a pointer to the contents of the file within the log instead of storing the bytes again. This optimization is very helpful for applications such as `blast` that repeatedly memory map the same file and for experiments that contain several applications that all use the same set of memory-mapped libraries.

Record logs system calls that manipulate input sources, such as `open`, `close`, and `select`. The `open` and `close` calls are also logged, because they are needed during replay to keep track of file descriptors. However, the `ioctl` call is not logged because it is used to read control information from an input/output device. When the experiment is reproduced, the actual device may be different, so the `ioctl` call must be re-executed.

Rex records system calls that manipulate directories, such as `chdir` and `mkdir`, because these calls convey information about the directory structure that is needed to reproduce an experiment. Finally, Rex records system calls that create new processes or threads, such as `fork`, `execve`, and `clone`. These convey information regarding how to create a new thread of execution, and their arguments are needed in order to create identical threads.

A log entry for a system call contains a header and the arguments of that call. The header contains the identity of the application thread making the call, the system call number, the number of arguments that follow the header, the return value of the call, and the error value of the call. The log for an argument contains the argument's type, length, and value.

Rex does not log system calls that do not provide input to an experiment. Common examples of these types of calls are ones that do memory management, thread synchronization, or thread signal management.

3.3 The `replay` Tool

The `replay` tool sets up the environment of an experiment and executes it with the recorded command line parameters. When the experiment requests for inputs, recorded input is fed to the experiment. Like `record` it monitors execution and intercepts all system calls made by the experiment. For each intercepted system call, `replay` either emulates the call, executes the call and then emulates its results, or simply re-executes the call. Figure 6 gives an overview of `replay`.

Replaying System Calls. The `replay` tool emulates a system call if there is an entry in the log for the system call and if any effects of not executing the call can be masked from the experiment. Examples of such system calls are `stat`, `access`, and `read`. However, if a `read` accesses a file that can also be written, the `read` is also executed in order to advance the file pointer. System calls that returned an error during the original experiment are always emulated; they return the reason for the failure.

For some system calls, `replay` executes them and then emulates their results because it is necessary for the side-effect of the execution of the system call to happen. Examples of such system calls are an `open` call to a write-only file or a file-based `mmap` call. For a file-based `mmap` call, the side-effect necessary for correct execution of a program is the allocation of a memory region for performing file I/O. In this case, the file based `mmap` call is converted to a normal `mmap` call, and after execution of the call, the recorded buffer for that `mmap` call is copied into the memory address returned by the call.

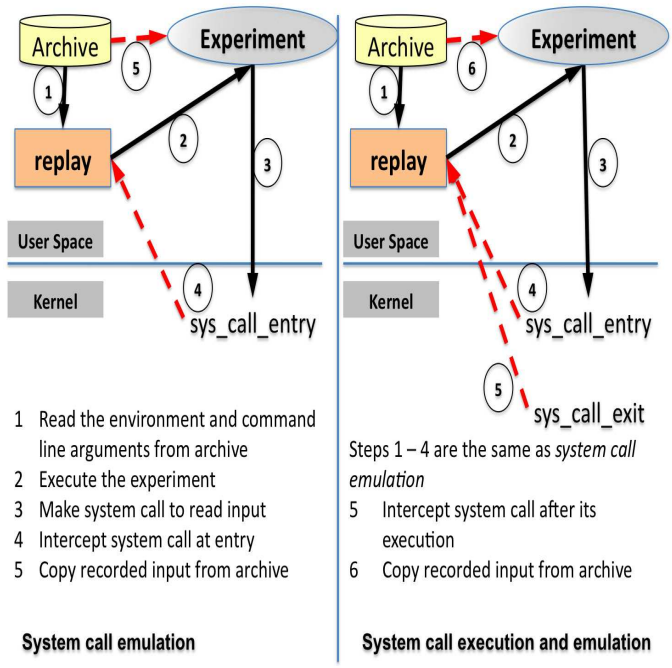

Figure 6. The `replay` Tool

System calls for which there is no entry in the log are executed normally. Examples of such system calls are `wait`, `futex`, `brk`, and `rt_sigaction`.

Emulating Directories. The directory structure of the file system is a key part of an experiment's environment. Rex could archive the directory structure during record and then replicate it during replay by using `chroot` to go to the recorded structure. However, one needs administrative privileges to execute `chroot`. So that a normal user can use `replay`, Rex emulates the directory structure for a replayed experiment by keeping track of all directories used by the experiment. Rex maintains the current working directory for each thread, which is initialized to the current working directory of its parent thread. The current working directory of the first thread is initialized to the value stored in the experiment's meta-data. Rex emulates directory-related calls such as `getcwd` and `chdir`. When replay intercepts a `chdir` call, it changes the current working directory maintained for that thread to the new directory provided by call, assuming that the call executed successfully in the original experiment. For a `getcwd` call, a copy of the thread's current working

directory is returned if the call's log entry has a successful return value.

Reproducing File Objects. File objects are the other key part of the environment that `replay` has to reproduce. When a replayed experiment accesses file objects, they have to be retrieved from one of the repositories. However, the file names used by the replayed experiment are those in the original experiment, not those in the repository. Hence, `replay` converts file names to refer to file objects within the repository. To aid in this name conversion, Rex builds a *file map* from the meta data of the experiment. The file map maps a file name and a current working directory to the location of that file object within the repository. For files that might be written, `replay` makes a copy of the file in the output directory, changes all mappings for that file object to point to the new location of the file. The `replay` tool employs two file maps: one for the apparatus and one for input/output files.

The file maps are used whenever file names are arguments in system calls. System call arguments are located in the calling thread's address space. File name parameters are encoded as a pointer (or address) to the location that contains the file name. The actual file name cannot be copied over the old file name because there might not be enough space to copy it, and the old file name could be used by the experiment later. Hence, `replay` copies the new file name to a different location in the calling thread's address space and changes the file name parameter to point to it. The new location is in a buffer that `replay` forces every thread to allocate in its address space.

Replaying Multi-threaded Programs. Rex reproduces multi-threaded programs that have externally deterministic output. Stated differently, if the execution order of the threads affects output, then the program is not reproducible because execution order is inherently non-deterministic in a multi-threaded program. On the other hand, if a program is synchronized so that it always produces the same result when given the same input, then Rex will faithfully replicate this observable behavior even if the threads execute in a different order when replayed than they did when recorded. As a specific example, suppose that when a program is recorded, thread *A* reads input from file `foo` and that thread *B* reads input from file `bar`. When the program is replayed, it may end up that thread `A` opens file `bar` and that thread `B` opens file `foo`. Each thread will blithely read a different file, which is fine as long as the output that the threads produce is the same as that produced when the experiment was recorded. (The `expdiff` tool can be used to determine this.)

3.4 The `runnew` Tool

The `runnew` tool allows a user to run a new experiment that is derived from an old experiment by:

- Changing some part of the apparatus,
- Changing some of the command-line parameters,
- Changing some of the environment variables, or
- Changing some of the input.

There are two components to the `runnew` tool: a user-interface that helps the user make changes, and a back-end tool that takes the changes as input, applies them to the archived experiment, and runs the new experiment. The user can optionally record the new experiment, in which case it is added to the same archive.

The user interface was shown earlier in Section 2. The back-end is similar to the `replay` tool. It first reads in the old experiment's meta data and sets up the new experiment. Then it takes the changes provided by the user interface component and applies them to the experiment. Changes to the command line parameters and environment variables are straight forward to apply. Changes to the apparatus and the input/output files are applied by changing the apparatus and file map. The back-end tool then executes the new experiment and intercepts the system calls that it makes. If a system call requests an item that is in the archive, Rex uses the file map to find the object in the archive, and then changes the system call parameter as described earlier for `replay`. The back-end tool records the new experiment in the same way that `record` saved the original experiment.

3.5 The `expdiff` Tool

The `expdiff` tool takes the archives for two recorded experiments and compares each pair of components: environment variables, command line arguments, apparatuses, and input and output files. For environment variables, `expdiff` reports ones that appear in one experiment but not the other, and ones that have different values in the two experiments. For command lines, `expdiff` performs a textual comparison of the entire command lines to report differences; it does not perform a semantic comparison because it knows nothing about the role of each argument. The `expdiff` tool compares apparatuses by comparing binary headers for binary components and by performing a simple `diff` for textual components, such as scripts.

The most significant task of `expdiff` is to determine differences between input files and output files. It uses the system call logs to examine the input read and output written by each program. It reports differences as follows, depending on the level of detail that is chosen by the user:

- **Level 1**: reports just the files that are different
- **Level 2**: reports the files that are different, the bytes that differ, and the total number of bytes that are different
- **Level 3**: reports all output from Level 2 plus the system calls that were skipped during comparison of the system call log

The logs are compared by comparing the system calls *process-wise* for the input they feed to the experiment and

the output they received from the experiment. This is done by comparing the buffers and return values that are used to pass data. If the system calls themselves differ, then `expdiff` follows an algorithm similar to that of the textual `diff` program to synchronize the logs as quickly as possible. First, for the current entry in the first log, find a match in the second log; and for the first entry in the second log, find a match in the first log. Then calculate the distance in each log from the current entry to the first match for the other log. Finally, choose the log with the shorter distance, and set the current system call in that log to the first entry that matches the current entry in the other log. The system calls that are skipped while synchronizing are noted and reported if the user requests Level 3 output.

The system call logs are pre-processed before comparison. First, systems calls that return an error are removed because they do not contribute to any input seen by the experiment.[3] Then, for each input/output source, the set of system calls that access that source are coalesced into a canonical form. For example, a read system call followed by another read call to the same input source by the same process is coalesced into a single read call. The contents of the buffer of the resulting call will be a concatenation of the buffers of the two read calls, and the length of the resulting buffer is the sum of the bytes read by the two calls. The rationale behind coalescing read calls is that the number of calls used to read input is irrelevant; it is the content and the length of the contents that need to be examined to see if there is a difference in the input. Write calls are coalesced in the same way and for the same reasons.

4. Experimental Results

This section presents the time and space overhead of using Rex. We measure the performance of the applications described in Section 2—BLAST, runassembly, and GloMoSim—as well as an additional bio-informatics program called Randomized Axelerated Maximum Likelihood (RAxML) [18]. RAxML calculates phylogenetic trees, which show evolutionary relationships among biological species [8]. For reference purposes, we also measure the performance of Rex on the SPECINT 2000 benchmarks.

Most benchmark programs were executed on an Intel Core 2 Duo system with a 2.0 GHz clock and 4GB of RAM and running the Linux 2.6 kernel. The runassembly and RAxML programs were benchmarked on an Intel Xeon Quad-core system with a 2.0 GHz clock and 24GB of RAM running the Linux 2.6 kernel. Execution overhead was computed by running each program six times; taking the median execution time (as reported by `gettimeofday`) for the original program, recorded program, and replayed program; and then calculating ratios. Space overheads were computed by simply measuring the sizes of the archive and logs and counting the number of system calls.

Figure 7 shows the execution overhead when using Rex to record and replay computational science and computer science experiments. The execution times for the application programs have been normalized to 1.0. The overheads for both recording and replaying `raxml` and `glomosim` are negligible because these programs do a lot of computation and have relatively few systems calls. The overhead for recording `blastall` is also negligible, but this application has a large amount of data that has to be copied into user space during replay, and writing data into user space is much more expensive than reading it from user space[4]

Recording and replaying the `runassembly` benchmark incurs execution overhead because of excessive concurrency. When runassembly starts executing, it probes for the CPU workload and spawns as many threads as there are free cores. At this point, the Rex thread doing record or replay is waiting on runassembly, so it does not contribute to the CPU workload. Thus more threads than free cores get created, and context-switching is required every time Rex traps a system call.

The `formatdb` benchmark has the most significant slow down because it makes around 58K system calls in a second, and these account for 16% of its execution time. (Most of our benchmarks spend less than 1% of their execution time on system calls.) Moreover, a large percentage of these system calls read data, which as noted above cause slowdown during replay due to data copying. Overheads similar to those in `formatdb` can only be expected in programs that make a lot of system calls and that require lots of recorded input during replay.

Figure 8 shows the execution overhead of using Rex to record and replay the SPECINT 2000 benchmarks. Almost all have very low overhead for both record and replay. The `gcc` benchmark makes a lot of `getrusage` calls that get canned input from the replay tool; this avoids the overhead that would incurred by executing the actual system call, and thus replay actually runs faster than record. The `vortex` benchmark—like `runassembly` and `formatdb`—reads a large amount of data and hence gets slowed down during replay.

Figure 9 gives space overheads for recording the various benchmark applications. For each, we recorded the number of system calls made by the benchmark, the size of the system call log, and the size of the archive (which contains the apparatus of the experiment, input and output files,

[3] For example, an experiment goes through a list of paths to search for an application. The system calls that test if the application exists on a particular path will return an error if it does not exist on that path. Two different experiments might find the application in different places. All that is relevant with respect to reproducibility is whether the application was eventually located.

[4] Data is read from and written to an experiment's address space using system calls. During record, Rex can read any amount of contiguous data from an experiment's address space using one system call. However, during replay Rex has to write data into an experiment's address space one word at a time. Thus, to write 4096 bytes during replay takes 512 system calls, assuming a word is 8 bytes.

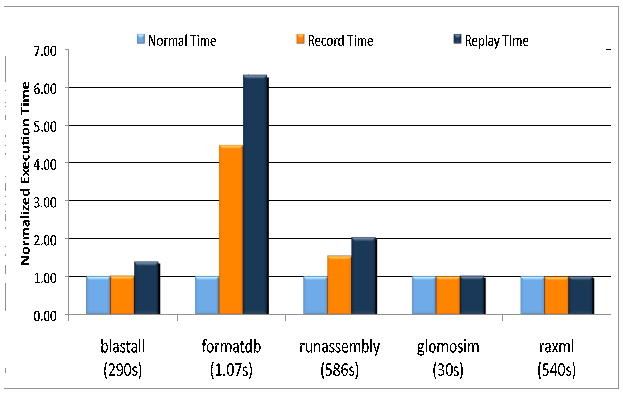

Figure 7. Science Experiments Execution Overhead

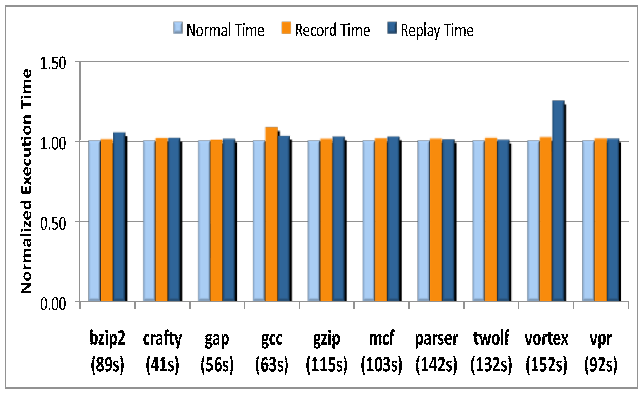

Figure 8. SPECINT 2000 Benchmark Execution Overhead

and the system call log). The size of the system call log depends on the number of system calls and the amount of data recorded for each of those calls. The `runassembly` application reads a 800 MB input file twice and thus has a huge log and archive. The `crafty` benchmark executes more system calls than `vpr`, but 99% of its calls are `ioctl` calls that require little space for logging. On the other hand, `vpr` makes very few calls, but 80% of them are reads that return a fair amount of data. Finally, `vortex` and `gap` both make about the same number of read system calls, but the total amount of data read by `vortex` is around 298M whereas `gap` reads only around 3M.

5. Related Work

There has been work in computational biology that focuses on integrating the data analysis part of a software experiment into scientific publications [17, 10, 5]. This is accomplished by embedding scripts into papers so that reported results can be independently reproduced and verified. BioConductor [11] and Madagascar [1] are two such projects. Both require that readers wanting to reproduce the results will install the appropriate software packages and the data sources that were used by the original computation. RA [15] focuses on

Benchmark	*No. System Calls*	*System Call Log (MB)*	*Archive (MB)*
Science Experiments			
blastall	33k	28	60
formatdb	58k	38	78
glomosim	792	3	7
raxml	948	3	8
runassembly	7M	3264	5270
SPEC INT 2000 Benchmarks			
bzip	334	26	53
crafty	9874	3	6
gap	13k	6	13
gcc	66k	45	89
gzip	473	39	80
mcf	65	5	11
parser	8k	5	10
twolf	467	5	10
vortex	10k	307	338
vpr	379	14	28

Figure 9. Space Overheads

reproducing experiments written specifically in Java. The experiment has to be annotated to record input and output to methods. Rex is more powerful than these tools because it automatically captures the code and data of an experiment (independent of the source language), and it does not require that the user modify anything.

Many debugging systems record executions of applications and use replay to reproduce errors [16, 9, 12, 4, 7]. Systems such as Jockey[16] and Liblog[9] inject a library at runtime into an application to record and replay it. They work on applications written in C/C++; Liblog also needs to modify the thread scheduler for multi-threaded program. Systems such as R2[12] require access to the source code of an application because the user has to add annotations in order to record and replay an application.

These systems focus on reproducing the exact states the application execution so that they reproduce any bugs. By contrast, the goal of Rex is to reproduce externally deterministic output. The internal states of an application can differ—as they will in multi-threaded programs especially—as long as the visible output is the same. Tornado [6] is similar to Rex, and it uses system call logging to record and replay programs. However, Tornado does not archive the apparatus, and to use it one has to make changes to the operating system. Rex aims to enable one to replay an experiment—or run new ones—long after the original experiment was recorded.

Systems such as Revirt[7] and iDNA [4] use virtualization to record and replay whole systems. They are heavyweight systems because they record the entire virtual machine (VM) on which an experiment runs, and replaying

an experiment requires a hypervisor that can understand that VM to perform replay. Rex is a much lighter-weight system and supports running new experiments, although it does require that replay and runnew execute on the same type of operating system on which an experiment was recorded.

6. Conclusions and Future Work

The Rex system supports recording and replaying experiments and comparing differences between recorded experiments. Because it captures the apparatus of an experiment, it also supports running new experiments on a recorded apparatus. Rex can handle a wide variety of computational science experiments—from shell scripts to multi-process workflows and multi-threaded programs—and it does not require that the user modify an experiment in any way.

Rex cannot yet handle MPI programs that run on a cluster of machines, but work is underway to do so. At present, differences between experiments are computed off-line; we are also working on extending Rex to report differences on the fly for long running programs.

There are also a few limitations of our current implementation. Rex monitors a program by trapping system calls, so it does not handle input received by using machine level instructions such as `rdtsc`, but instructions such as this have not been used in any application that we have tested. Rex also does not handle fast system calls such as `gettimeofday` that are included in the newer Linux kernels; these could be intercepted by pre-loading hooks for each application using the `LD_PRELOAD` mechanism. Finally, Rex does not archive the operating system on which a recorded experiment is run; this could be supported by using virtualization technology.

References

[1] Madagascar. `http://www.reproducibility.org/wiki/Main_Page`.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] SF Altschul, TL Madden, AA Schaffer, J. Zhang, Z. Zhang, W. Miller, and DJ Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389, 1997.

[4] S. Bhansali, W.K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drini, D. Mihoka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 2 nd international conference on Virtual execution environments*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2006.

[5] J. Buckheit and D.L. Donoho. Wavelab and reproducible research. *Wavelets and Statistics*, 103:55–81, 1995.

[6] F. Cornelis, M. Ronsse, and K. De Bosschere. Tornado: A novel input replay tool. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA*, volume 3, pages 1598–1604.

[7] 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. *ACM SIGOPS Operating Systems Review*, 36:211–224, 2002.

[8] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of molecular evolution*, 17(6):368–376, 1981.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference*, volume 2006, pages 2–3, 2006.

[10] R. Gentleman. Reproducible research: A bioinformatics case study. *Statistical Applications in Genetics and Molecular Biology*, 4(1):2, 2005.

[11] R. Gentleman, V. Carey, D. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5(10):R80, 2004.

[12] 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 Eighth Symposium on Operating Systems Design and Implementation (OSDI08)*, pages 193–208, 2008.

[13] M. Haardt and M. Coleman. ptrace (2), 1999.

[14] S.T. King, G.W. Dunlap, and P.M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference*, pages 1–15, 2005.

[15] D. Ramage and A.J. Oliner. RA: ResearchAssistant for the computational sciences. In *Proceedings of the 2007 workshop on Experimental computer science*, page 19. ACM, 2007.

[16] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging table of contents*, pages 69–76. ACM New York, NY, USA, 2005.

[17] M. Schwab, M. Karrenbach, and J. Claerbout. Making Scientific Computations Reproducible. *Computing in Science & Engineering*, pages 61–67, 2000.

[18] A. Stamatakis. The RAxML 7.0. 4 Manual. *The Exelixis Lab, LMU Munich.(April 2008)*.

[19] T.T. Torres, M. Metta, B. Ottenwälder, and C. Schlötterer. Gene expression profiling by massively parallel sequencing. *Genome research*, 18(1):172, 2008.

[20] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. *ACM SIGSIM Simulation Digest*, 28(1):154–161, 1998.