Jockey: A user-space library for record-replay debugging

Yasushi Saito
Hewlett-Packard Laboratories
Palo Alto, CA, USA
yasushi.saito@gmail.com

Abstract

Jockey is an execution record/replay tool for Linux. It records and replays invocations of system calls and CPU instructions with context-dependent effects, without requiring changes to the target program, operating system, or debugger. Jockey is implemented as a user-space library that runs as a part of the target process. While this design is the key for achieving Jockey's goal of safety and ease of use, it also poses challenges. This paper discusses some of the practical issues we needed to overcome, including low-overhead system-call capturing, resource-usage segregation between Jockey and the target process, checkpointing in the presence of the dynamic linker, and an interface for fine-grain control of Jockey's behavior. Jockey has been applied extensively to debug real-world programs. We our experiences using Jockey as well.

1 Introduction

Jockey is a record/replay tool for Linux. It logs the execution of an ordinary program and replays it deterministically later. Jockey is designed to help debug programs that communicate with the operating system or other computers in a complex fashion. We plan to make Jockey publicly available via http://www.freshmeat.net, pending legal authorization.

Jockey was originally developed as a debugging aid for *FAB* (Federated Array of Bricks) [23]. FAB is a high-availability disk array built on a cluster of commodity servers. It provides accesses to logical volumes to iSCSI clients using complex peer-to-peer-style replication and erasure-coding protocols.

Traditional debuggers, such as gdb, provide comprehensive support for debugging single-node, sequential programs. They are, however, not as useful for interactive or distributed programs such as FAB [10, 4]. We identify three key problems and discuss how Jockey alleviates them.

First, the execution of such programs is inherently non-deterministic. The behavior of a process will diverge, depending on interactions with the OS, the user, or other processes. Jockey helps debug such programs by recording every non-deterministic choice the process makes, and replaying the execution as many times as the developer wishes. Thus, debugging for a non-deterministic program is reduced to that for a sequential, repeatable program.

Second, these programs often run for a long period of time, either because they need lots of resources (e.g., scientific computation), or they are server programs (e.g., FAB and distributed hash tables), or they need substantial user interactions (e.g., spreadsheet). Simply reproducing the bug often tests a developer's patience. Jockey alleviates this problem by transparently checkpointing the process state during execution. The developer can replay from any checkpoint and easily "time-travel" through the history of execution to diagnose bugs. Checkpointing also bounds the log-space overhead, as log records older than the checkpoint can be discarded.

Third, running a distributed system such as FAB requires starting processes on multiple computers, which is cumbersome and increases the turn-around time for program development. Jockey alleviates this problem by recording and replaying each process independently—after recording the execution whole system, the developer can replay each process under a traditional debugger. This can also be a limitation; Jockey could be less useful when one wants to look at the execution of the whole system at once. We discuss our experiences in Section 5.2.

1.1 Goals and approaches

Jockey is designed with two pragmatic goals in mind. First is *ease of use*: Jockey must be easy and safe to deploy. It should work without requiring changes to the target program, the operating system, or the debugger. Second is *generality*. Jockey should be able to handle generic Linux programs, not just those written in a particular programming language or API, such as MPI or CORBA [11].

We achieve the first goal by implementing Jockey as a user-space library that runs as a part of the target process. In contrast to kernel-based approaches [26], it can be used by anyone without administrator privilege or a patched kernel. Developers can continue using their favorite debuggers without change. In addition, this design allows the developer or the target program to control or extend Jockey easily, as we discuss in Section 4. Our second goal is achieved by recording and replaying events at a fairly low level—system calls and CPU instructions.

1.2 Non-goals

We do not try to make program execution under Jockey identical to native execution (without Jockey). Because Jockey internally needs to perform mmap and file accesses, the mmap addresses and file descriptors allocated to the target process may differ between a native and a Jockey run. This usually does not cause an additional burden, because programs traced by Jockey are usually non-deterministic to begin with.

Similarly, performance is a secondary goal. Jockey is used only during testing and debugging. Some slowdown under Jockey should be acceptable, as long as it does not alter the target program's behavior qualitatively. In practice, as we show in Section 5.1, Jockey's overhead is at most 30% for I/O intensive programs, more often close to zero—well within our limit of tolerance.

1.3 Example

The meat of Jockey is `libjockey.so`, an x86 shared-object file. Figure 1 shows a simple program that reads from `/dev/random`, Linux's random number device. Figure 2 shows the most basic use of Jockey. Running a program with Jockey requires no change to the source code or the executable file. Simply loading `libjockey.so` on startup causes Jockey to take control of the process. In this example, Jockey intercepts the call to the `read` system call made via `getc`. It logs the value read during the recording phase. When replaying, Jockey reads the value from the log without actually reading from `/dev/random`. Jockey can also be invoked in several different ways, as illustrated in Figure 3.

```
// test1.c
int main() {
    FILE *f = fopen("/dev/random", "r");
    printf("%x\n", getc(f));
}
```

Figure 1: *A simple program, `test1.c`, with a non-deterministic behavior.*

```
% cc -o test1 test1.c
% LD_PRELOAD=libjockey.so \
  JOCKEYRC="replay=0" ./test1 # recording
38
% LD_PRELOAD=libjockey.so \
  JOCKEYRC="replay=1" ./test1 # replaying
38
```

Figure 2: *The most basic use of Jockey. The program outputs the same number even though it is reading from `/dev/random`. Setting `LD_PRELOAD` causes the dynamic linker to load `libjockey.so` before other object files. The `JOCKEYRC` environment variable passes parameters to `libjockey.so`.*

1.4 Challenges and limitations

Our decision to co-locate Jockey with the target process poses challenges and limitations. First, Jockey could be compromised by a seriously buggy or malicious target program—if it wishes, for example, the target program can destroy a memory region used internally by Jockey. Jockey, however, tries to prevent such problems by segregating resources used by Jockey and the target as much as possible, as discussed in Section 3.2.

The second challenge is recording and replaying events that are not directly initiated by system calls. We describe our solutions to two such types of events, signals and memory-mapped file I/Os in Sections 3.5 and 3.6. There are, however, events that are fundamentally impossible to capture. For example, memory access races that happen with kernel-based pthreads cannot be replayed, because in-kernel context switches and SMP cache-coherence protocols are out of Jockey's control. For this reason, Jockey does not support kernel multi-threading. Similarly, Jockey does not support any program or API that interacts with other processes (or devices) via shared memory or files—

```
 1  % LD_PRELOAD=libjockey.so \
 2    ./test1 --jockey=replay=0    # recording
 3  82
 4  % LD_PRELOAD=libjockey.so \
 5    ./test1 --jockey=replay=1    # replaying
 6  82

 8  % jockey --replay=0 ./test1    # recording
 9  a9
10  % jockey --replay=1 ./test1    # replaying
11  a9

13  % cc test.c -ljockey
14  % ./test1 --jockey=replay=0    # recording
15  c1
16  % ./test1 --jockey=replay=1    # replaying
17  c1
```

Figure 3: *Alternative ways of running a program under Jockey. Lines 1 to 6 shows an alternative method that passes a command-line parameter* `--jockey=`*. This parameter is parsed by* `libjockey.so`*. For this method to work, the target program must be designed to ignore a command-line string that starts with* `--jockey=`*. Lines 8 to 11 show how one can start the test program from the* `jockey` *frontend.* `jockey` *just sets the environment variables and* `execve`*'s the target program. Jockey can also be linked manually to the target program, instead of via* `LD_PRELOAD`*. Lines 13 to 17 shows how that can be done.*

e.g., uDAPL [1] for memory-mapped network I/Os. Note that Jockey does support user-space threads, such as Capriccio [30]—in fact, FAB is built on a similar package.

2 Related work

Execution record/replay has long been advocated as an effective debugging method [6, 22, 21].

Bugnet was one of the earliest record/replay tools [32]. It intercepted I/O activities by the processes and took checkpoints of the system periodically. Bugnet, however, supported only programs written to a special API, unlike Jockey that supports generic Linux programs. Flashback [26] is the most recent work along this line. It offers a similar set of functionalities as Jockey—recording and replaying system calls and fork-based checkpointing—but Flashback is offered as a kernel extension. As such, it is less easy and safe to use than Jockey.

In a slightly different approach, some systems record and replay individual memory accesses [19, 16, 25, 7]. They have several potential advantages over event-based approaches like Jockey. First, some of them enable reverse execution—stepping CPU instructions literally backward [19, 7]. Second, they could be more generic because they need not know deeply about the semantics of system calls and other interactions with OS. However, these systems require special compilers and have a large logging overhead. Even with sophisticated optimizations, they generate logs at the rate of multiple megabytes per second for CPU-intensive programs [16, 25]. Jockey, in contrast, generates only a few hundred bytes per second for such programs, as we show in Section 5.1.

2.1 Record/replay using virtual machines

Revirt is a virtual machine that records and replays low-level interrupts and device activities [2]. It has been proved to be useful for network intrusion detection and diagnosing kernel bugs [5]. Several other papers also propose distributed-system emulation using virtual machines [3, 17]. While these systems are powerful, they are also cumbersome to use—for example, one must create a full file system tree for each virtual machine. They are overkill when one is interested only in debugging user-space programs. Jockey is designed to be simpler and easier to use than these systems.

2.2 Record/replay for parallel and distributed programs

Deterministic record/replay has been most effective in parallel and distributed environments [21]. Indeed, earliest tools specifically targeted such environments [32, 19]. Since then, many theoretical improvements have been proposed for both shared-memory parallel programs [13] and message-passing programs [14, 15]. Jockey does not yet support deterministic replay of a distributed system—it can only replay processes within the system independently. As we discuss in Section 5.2, we found this limitation not to be a serious obstacle so far.

3 Implementation of Jockey

We have implemented Jockey in C++. Linux's dynamic linker (`ld.so`) invokes Jockey's initialization routine immediately after `libjockey.so` is loaded, before the target program starts execution. The initialization routine performs the following tasks.

(1) For each system call in `libc` with timing- or context-dependent effects, Jockey rewrites its first few instructions and intercepts calls to it. Jockey currently

intercepts 80 Linux system calls, including `time`, `recvfrom`, and `select`.

(2) Jockey does the same for CPU instructions with non-deterministic effects. It currently patches only `rdtsc`, the x86 instruction for reading the CPU's timestamp counter. It is used, for example, as a pseudo random-number generator in `libc`.

(3) Jockey checkpoints the process state just before returning control to the target program. In the replay mode, Jockey simply loads the checkpoint. Checkpointing is needed to ensure that the target sees the same set of environment variables and command line parameters during both record and replay.

(4) Jockey transfers control to the target program. From this moment, Jockey becomes active only when the target executes a system call or a non-deterministic CPU instruction. Upon intercepting one of these events, Jockey logs the values generated by the event during recording, and reads the value from the log during replay.

The next section describes the first two steps in more detail. Section 3.2 discusses Jockey's efforts to segregate itself from the target program to avoid unnecessary interference. Section 3.3 describes Jockey's checkpointing mechanism (step (3)), along with the challenges we had to overcome. We describe our approach to reduce the event logging overhead (step (4)) in Section 3.4.

3.1 Instruction patching

As an example, Figure 4 shows how the `time` system call is recorded and replayed. (a) shows the first few CPU instructions of `time` in `libc.so`. When Jockey starts, it writes a `jmp` instruction in the first 5 bytes of the procedure, as shown in (b). If the 5th byte is in the middle of another CPU instruction, as is the case with `time`, Jockey overwrites up to the next instruction boundary (and fills the memory with `nop` as needed). In (c), Jockey also copies the original first 5 bytes (6 bytes for `time`) of the function to a newly allocated memory so that Jockey can run the old implementation if necessary. (d) shows the entry point for the new implementation of `time`. Jockey dynamically generates this code so that it can execute on a separate stack and avoid corrupting target memory (Section 3.2). Finally, (e) shows `newtime`, Jockey's implementation of `time`. While recording, `newtime` calls the original `time` (c) and logs the returned value. Upon replaying, it simply supplies the value from the log without actually calling `time`.

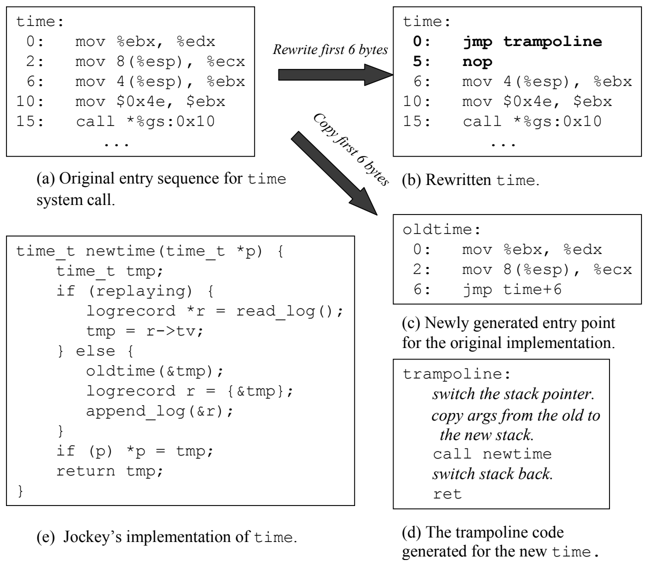
One might wonder why `libjockey.so` does not just provide a new implementation of a system call with the same name—in fact, `LD_PRELOAD` is often used for that purpose. The reason is that doing so will miss calls made inside `libc` or the dynamic linker—for example, the call to `read` via `getc` in Figure 1. These internal calls are pre-resolved by the static linker (`ld`), and cannot be overridden by redefining system call functions in `LD_PRELOAD`.

For task (2), Jockey rewrites all offending CPU instructions found in the target process. This is done in two ways, *slow* mode and *cached* mode. In slow mode, Jockey first reads the special file `/proc/`*n*`/maps` (*n* is the target process ID) that shows the virtual-memory mappings of the target process. It then reads the header of each mapped shared object file, discovers the locations of the text sections, and scans each text section. Jockey finds non-deterministic CPU instructions in the section (if any), and patches them. Jockey also intercepts invocations of `mmap` system call and does the same.

Jockey needs to parse CPU instructions during steps (1) and (2), not a trivial task given x86's complex instruction encoding. It uses a pidgin table-based parser for common instructions and consults *libdisasm* [8], an open-source x86 disassembler library, for uncommon cases. A few tables that map opcodes/operands to their instruction length let us quickly parse more than 80% of all instruction occurrences.

Even using this technique, however, parsing all CPU instructions in a typical Linux program takes about 350 milliseconds on a 1.5GHz Pentium-M processor, which may be too slow for some users. To reduce the startup latency further, Jockey also employs cached-mode instruction patching. Here, after finishing the slow mode, Jockey writes the locations of non-deterministic instructions found for each shared object in file `~/.jockey-sig`. When Jockey starts the next time, it just reads `~/.jockey-sig` without scanning the process's virtual memory, unless the timestamp of the object file has changed.

Jockey's instruction-patching approach is simpler and faster than full-program binary translation, employed by ATOM [27] or Valgrind [24]—as we show in Section 5, the Jockey's performance overhead is negligible for CPU-intensive programs.



```
time:
 0:   mov %ebx, %edx
 2:   mov 8(%esp), %ecx
 6:   mov 4(%esp), %ebx
10:   mov $0x4e, $ebx
15:   call *%gs:0x10
         ...
```

(a) Original entry sequence for `time` system call.

```
time:
 0:   jmp trampoline
 5:   nop
 6:   mov 4(%esp), %ebx
10:   mov $0x4e, $ebx
15:   call *%gs:0x10
         ...
```

(b) Rewritten `time`.

```
oldtime:
 0:   mov %ebx, %edx
 2:   mov 8(%esp), %ecx
 6:   jmp time+6
```

(c) Newly generated entry point for the original implementation.

```
trampoline:
    switch the stack pointer.
    copy args from the old to
      the new stack.
    call newtime
    switch stack back.
    ret
```

(d) The trampoline code generated for the new `time`.

```
time_t newtime(time_t *p) {
    time_t tmp;
    if (replaying) {
       logrecord *r = read_log();
       tmp = r->tv;
    } else {
       oldtime(&tmp);
       logrecord r = {&tmp};
       append_log(&r);
    }
    if (p) *p = tmp;
    return tmp;
}
```

(e) Jockey's implementation of `time`.

Figure 4: *Recording and replaying* `time`.

3.2 Segregating resource usage

Jockey and the target application run as part of the same process and share all resources. Jockey must segregate the use of resources to prevent Jockey from unnecessarily changing the target's behavior, and to minimize the chance of a misbehaving target program breaking Jockey. This section discusses Jockey's treatment of three types of resources: heap, stack, and file descriptors.

Heap: Jockey cannot use standard libc functions, such as `malloc` or `sbrk`, to manage its internal data. Doing so increases the likelihood of a misbehaving target program breaking Jockey. Moreover, it changes the memory layout of the target process between record and replay. It would thus become impossible to replay invalid memory accesses correctly—e.g., accessing `free`'ed memory, which is one of the common programming errors.

Instead, Jockey stores all its internal data in a fixed mmaped virtual address (0x63000000) that is unlikely to be accessed accidentally by the target program. Jockey uses an internal malloc-like library to carve the memory out to individual data structures and builds a custom C++ STL memory allocator on top of it. Thus, the Jockey code has full access to STL features, including hash tables and dynamic vectors. This design has simplified the development of Jockey considerably.

One restriction is that Jockey cannot make internal calls to libc functions that use `malloc`. Examples include high-level I/O functions (`fopen`, `std::fstream`) and DNS resolvers (`gethostbyname`).

Stack: Jockey also segregates the use of stack. This is necessary to replay a program that improperly accesses data beyond the stack pointer (e.g., accessing an on-stack array with a negative index). Figure 4 (d) illustrates how this is done. In the first few instructions after it intercepts the call to `time`, Jockey saves the stack pointer in an internal variable, switches the stack to an internal buffer, copies the parameters to `time` (a 4-byte pointer) from the old to the new stack, and calls `newtime`. Once the new implementation returns, Jockey restores the stack pointer. This allows for deterministic replay of even a program that abuses the stack, because Jockey never uses the target's stack.

This stack-switching code must run without touching any CPU register other than the stack pointer. Thus, all the data structures involved here are allocated statically. This makes Jockey non-reentrant, but it is not an issue because Jockey does not support multi-threading.

File descriptors: Jockey must do its own file accesses occasionally, for example, when opening a log file or dumping checkpoints. Because Jockey and the target process share the same file-descriptor table, Jockey must ensure that its file operations do not alter the descriptor allocation scheme seen by the target. For this purpose, Jockey moves file descriptors it internally opens to a fixed range not likely to be used accidentally by the target (430∼439).

Gdb (debugger) poses another problem. When starting the target process, gdb opens a few extra file descriptors in addition to the usual stdin, stdout, and stderr. Thus, if execution is recorded under a normal shell and then replayed under gdb, the files opened by the target processes will be assigned different descriptors, which makes replaying divergent.[1] We solve this problem by having Jockey open dummy files for descriptors 0 to 9 before starting the target program (descriptors inherited from the parent process are left untouched). Assuming that gdb opens at most 10 descriptors when it starts the target, we can ensure that the target has the same set of files open upon record and replay.

[1]In fact, this problem is not just specific to gdb. It happens whenever the target process inherits more than the standard number of file descriptors from the parent.

```
% jockey --checkpointfrequency=30 \
    --retaincheckpoints=5 \
    -- httpd -X
   ... later ...
% jockey --restore=.jockeylog/checkpoint-3 httpd
```

Figure 5: *Taking automatic `httpd` (Apache) checkpoints every 30 seconds. The `-X` option runs Apache in foreground. The `--retaincheckpoints=5` option causes only the last five checkpoints to be retained. The last line replays `httpd` from the third checkpoint.*

3.3 Checkpointing

Jockey allows process state to be checkpointed automatically. Figure 5 shows an example. Checkpointing serves two purposes. First, it allows the developer to time-travel through the history of execution quickly. Second, it bounds log-space consumption, because log records older than the oldest checkpoint can be deleted from disk.

Following the technique pioneered by libckpt [20] and flashback [26], Jockey first forks the target process. It then dumps the state of the child, while letting the parent continue running. Jockey reads the `/proc/`*n*`/maps` (*n* is the process ID) to obtain the virtual memory mappings of the process and dumps only those sections that are mapped privately and read-write. To restore a checkpoint, for each section recorded in the checkpoint file, Jockey unmaps the memory region if it is already occupied, and either restores the contents from the checkpoint file or remaps the file.

We discuss two particular problems we faced, both related to dynamic linking.

Preventing brain damage to the dynamic linker One of the challenges of checkpoint restoration is that Jockey needs to overwrite the memory that is potentially used by the restoration code itself. Jockey would crash if restoration is done naively. Here, two types of memory regions need to be taken care of: Jockey's internal heap (Section 3.2) and the heap used by the dynamic linker. For example, Jockey must execute the `read` system call to load checkpoint contents. If the call to `read` happens to be the first ever made by the target application or Jockey, then the dynamic linker is invoked to resolve the symbol "`read`", which involves modifying the linker's heap.

Jockey handles its internal heap by excluding it from checkpointing, but the dynamic linker poses a particular challenge—we cannot know a priori where the memory used by the dynamic linker is (the linker performs an anonymous mmap of its heap memory; all anonymous-mmaped sections look the same to Jockey). We handle this by eagerly linking all libc functions used by Jockey, by making dummy calls to functions such as `open` and `read`, before it restores any checkpoint. After restoration, the dynamic linker's heap reverts back to the state during recording.

Exec shield Exec-shield is a facility found in some Linux kernels (e.g., Red Hat, Fedora Core) to thwart buffer-overflow attacks [12]. One of its features is randomization of the loading addresses of shared-object files. This feature breaks Jockey because Jockey needs to keep data structures that are specific to the process's memory layout. We currently demand that this feature be disabled by doing the below on machine boot.

```
  echo 0 >/proc/sys/kernel/exec-shield
```

3.4 Reducing logging overhead

Jockey employs two types of logging policies, depending on the types of system calls, to reduce the log-space overhead.

- For requests to regular files or directories, Jockey performs "undo" logging [9]. That is, for system calls that update a regular file or directory, Jockey logs enough information to restore its contents *before* the modification. For example, when a `write` system call overwrites the mid-section of a file, Jockey logs the offset and the old contents of the section. Or, when `write` appends to the end of the file, Jockey just logs the old size of the file. In the replay mode, Jockey scans the log from the end to the beginning and restores the file contents. Read-only system calls (e.g., `read`) to regular files are simply executed directly.

- For all other types of events—I/Os to sockets, pipes, or `select`, `time`, or `rdtsc`—Jockey performs "redo" logging. Jockey logs the value produced by the event during recording, as illustrated in Figure 4 (e). During replay, Jockey reads the value from the log without executing the actual system call.

System calls such as `read` and `write` can operate on both types of files. We intercept calls to functions that create file descriptors—e.g., `open`, `socket`, and `accept`—remember the type of each descriptor, and dispatch based on the descriptor type. File descriptors inherited from the parent process (e.g., stdin) are always redo-logged.

Various studies have shown that majority of I/Os to regular files are reads, and that most of the write traffic is actually appends [18, 29]. For these common cases, our design allows Jockey to only log the type and the offset of the requests, not the actual contents. Thus, it drastically reduces the logging overhead for I/O system calls for regular files.

The downside of the undo-based logging is that the user cannot modify the files accessed by the target program between record and replay. So far, we have found this not to be a significant burden.

3.5 Handling signals

Signals, especially those that happen asynchronously (e.g., SIGALRM, SIGINT), present a special challenge. We handle them in a way similar to [28]. Each signal delivery is first intercepted by Jockey. Jockey's signal handler simply records the parameters to the signal (signal number and the CPU register values) and finishes. At the end of the Jockey's handler for a system call or `rdtsc` CPU instruction, Jockey checks if a signal was intercepted in the past. If so, it logs the signal (so that it can be replayed) and calls the target-defined signal handler. This way, Jockey converts asynchronous signals to synchronous upcalls that only happen immediately after a system call.

This technique may distort program behavior when the target program runs without issuing a system call (or executing non-deterministic CPU instructions) for a long period and receives signals in the meantime. However, Jockey's primary targets, I/O-oriented programs, usually do not suffer from this problem.

3.6 Handling memory-mapped I/Os

Updates to memory mapped files are handled using MMU protection mechanisms. Jockey intercepts calls to `mmap` and `mprotect`. For each file requested to be mapped read&write in a shared mode (i.e., `MAP_SHARED`),[2] Jockey makes the mapped region read-only, and takes a page fault (`SIGSEGV` signal) after the first write access to each page in the region. In the `SIGSEGV` handler, Jockey logs the current page contents (Section 3.4), makes the page writable, and returns the control to the target.

These memory pages are made read-only again just before checkpointing, so that Jockey can restore the contents of the file at the moment of each checkpoint.

4 Controlling Jockey

Jockey is designed to replay executions without requiring modification to the target source code. Sometimes, however, allowing the target program to change the behavior of Jockey would enable more efficient program execution or debugging. Jockey's library-based design makes it easy to offer such control knobs for the target. This section introduces some of them.

`jockey_set_fork_trace_mode`(*mode*): By default, upon `fork`, Jockey continues recording only the parent and disables recording the child. This function can be called by the target program to record only the child, or both.[3] It can be used, for example, for daemon-type programs that fork to detach themselves from the parent process.

`jockey_redirect_calls(`*`name`*`, `*`newproc`*`, `*`size`*`)`: This function is used to transfer the control to *newproc* whenever function *name* is called. Parameter *size* is the size of the on-stack parameters to the function. This function is implemented using instruction-patching service discussed in Section 3.1. This feature can be used, for example, to provide record/replay functionality for obscure `ioctl` commands.

User-defined invariant checker: Jockey allows an arbitrary object file to be linked and run during replay. Figure 6 shows an example. Here, assume that we ran `test2.c` under Jockey and found that procedure `foo` behaves anomalously when `i == 95999`. We could diagnose the problem by setting a breakpoint on `foo` in a debugger and waiting until it hits 95999 times, but Jockey offers a quicker alternative, as shown in Figure 7.

The implementation of this feature is tricky, because we cannot use the dynamic linker to load the object file into the target process—doing so would alter the heap structure of the target (Section 3.3). Jockey instead uses the static linker, `ld`. Jockey first discovers the addresses of all public symbols in the target process, including those exported by Jockey, by invoking `nm` for each shared object file loaded by the target. Jockey then invokes `ld`, passes the symbols'

[2]Accesses to a private mapping (`MAP_PRIVATE`) need not be intercepted, because private mapping is essentially a heap memory with particular initial contents.

[3]The behavior of `fork` can also be controlled via `JOCKEYRC` environment variable.

```
// test2.c
void foo(int i) {
   ... do something complex ...
}
void main() {
   for (int i = 0; i < 100000; i++) foo(i);
}
```

```
// checker.c
#include <jockey/jockey.h>
void foocheck(int i) {
   if (i == 95999)
      jockey_breakpoint();
}
void init() {
   jockey_interpose_calls("bar", foocheck, 4);
}
```

Figure 6: *`test2.c` is the target program. `checker.c` shows a user-defined checker that calls a breakpoint after `bar` is called 95999 times. Function `init` is called by Jockey when the checker is loaded into memory. `jockey_interpose_calls` is similar to `jockey_redirect_calls`, but it returns the control to to the target program after the callback returns. `jockey_breakpoint` is just a no-op.*

```
% cc -c checker.c
% gdb test2
(gdb) b jockey_breakpoint
(gdb) run --jockey=replay=1;\
checker=checker.o
```

Figure 7: *Running the user-defined checker. Procedure `foochecker` in `checker.c` will be called before every time `bar` executes.*

addresses, and instructs `ld` to resolve the symbols in the checker object at a fixed virtual address unlikely to be accessed by the target program (0x62000000). Jockey then reads the produced binary directly into memory and executes it.

5 Evaluation

This section reports performance and space overheads of Jockey and discusses our experiences applying Jockey to real-world programs.

5.1 Performance and log-space overheads

The evaluation was performed on a Fedora Core 3 Linux machine with a 1.5GHz Pentium-M CPU, 512MB of memory, and a 7200 rpm ATA disk drive. We ran a variety of programs under Jockey, as listed below. Stock binary executable files from the Fedora Core distribution were used, except for FAB.

Name	**Run time**			**Log size**	
	Native	**Record**	**Replay**	**#bytes**	**#records**
g++	1.33	1.51	1.49	73KB	80
xclock	N/A	180	0.4	80KB	4639
Emacs	N/A	210	5.81	1.4MB	20769
httpd	16.7	17.5	9.5	2.0MB	140180
FAB	33.7	44.1	31.1	34MB	887000

Table 1: *The performance and log-space overheads of Jockey. Run times are in seconds. "Native" is the run-time without Jockey. "Record" and "Replay" show the runtime during recording and replaying, respectively.*

g++ gcc 3.4.2 compiling a small C++ program that uses an STL `map`. The result shows the sum of the frontend (`g++`), backend (`cc1plus`), assembler (`as`), and linker (`ld`).

xclock a digital clock for the X window system with a screen update every second.

Emacs Emacs 21.3 running a program-development session, involving active typing, file reading, and saving.

httpd Apache 2.0.52 (single process, no forking), serving 100,000 HTTP GET requests for a static 0.5KB file.

FAB A four-process FAB cluster [23] serving 80000 random 1KB read and write requests.

g++ is an example of a short-running, CPU-intensive program, which is not Jockey's primary target. This example still shows that Jockey has a far lower overhead compared to approaches that involve memory-access logging [16], which could produce a few megabytes per second of logs. For g++, most of the slowdown is due to checkpointing that happens at the beginning of the execution (Section 3).

Xclock and Emacs are examples of interactive applications. Jockey exhibits reasonable log-space overheads for them. Jockey is able to replay their execution extremely quickly, because they need not wait for timeouts or user inputs during replay. This translates to more efficient debugging sessions.

Apache and FAB are examples of server programs. FAB represents the worst case for Jockey. Not only does FAB perform large amount of network I/Os, it also overwrites existing files (devices) repeatedly, resulting in a large amount of logging traffic (Section 3.4). In comparison, Apache has a far lower overhead because it only performs read-only accesses to HTML files and appends to access-log files.

5.2 Experiences

We have used Jockey regularly for FAB development. Our experiences have overall been positive. Jockey has been most useful when diagnosing bugs that happen during long regression tests. Before Jockey, we were forced to recompile and reboot the system many times, each time with a slightly different set of "printf" statements, hoping that we would eventually reproduce and catch the error. Jockey allows us to reproduce the bug reliably as often as we wish. Fixing such bugs, however, is still difficult even with Jockey. The real cause of the bug often happens minutes before the bug exhibits, often on a different machine. The developer needs to replay the execution of multiple processes repeatedly to locate the cause.

Jockey has also been surprisingly effective in diagnosing bugs that exhibit quickly, e.g., while processing the first request from the client (indeed, most real-world bugs are of this type). Jockey cuts the debugging turn-around time by allowing the developer to replay a single process quickly instead of restarting the entire cluster.

Our experiences so far suggest that deterministic distributed-system replay (Section 2.2) is not worth the complexity, at least for systems like FAB. The most important feature of a record&replay tool is the ability to replay quickly and reduce developers' turn-around time. The whole-system replay does not improve on this issue; it may indeed increase the replay latency.

There are a few Jockey features that sound useful in theory, but have turned out to be not quite so in practice. First is user-defined invariant checking (Section 4). Debugging is an ad-hoc activity—writing and compiling a program every time one wants to debug is awkward. A debugger support, such as transparently compiling and loading a user-defined watchpoint, would help. Another problem is that the checker can only do only limited things—for example, it cannot intercept calls in the middle of function execution, nor can it inspect on-stack variables in the call chain.

Second, the concept of "time travel" using periodic automatic checkpoints (Section 3.3) has turned out to be powerful but somewhat cumbersome. The developer must manually restart the process every time he or she wants to switch to a different checkpoint. The developer can easily lose track of which part of the execution he or she is replaying. An extension to debuggers, such as automatic checkpoint scanning for detecting invariant violation [5, 31], would go a long way toward make this feature truly useful.

6 Conclusion

This paper described Jockey, a Linux tool for deterministic record/replay debugging. To achieve Jockey's goals of safety and easy of use, it is implemented as a user-space library that runs as a part of the target process. It intercepts calls to non-deterministic system calls and CPU instructions, logs the effects of these operations during recording, and replays them from the log during replay. Jockey has a small performance and log-space overhead. Jockey has been extensively used to develop FAB.

References

[1] DAT collaborative. User-level direct access transport APIs (uDAPL), 2004. http://www.datcollaborative.org/udapl-.html.

[2] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[3] T. L. Harris. Dependable software needs pervasive debugging. In *10th ACM SIGOPS European Workshop*, Saint Emilion, France, September 2002.

[4] J. Huselius. Debugging parallel systems: A state of the art report. Technical Report 63, Dept. of CSE, Malardalen University, September 2002.

[5] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Tech. Conf.*, Anaheim, CA, USA, April 2005.

[6] L. C. Lam. A survey of data breakpoint and reverse execution. SUNY Stony Brook RPE report, http://www.ecsl%-.cs.sunysb.edu/tr/rpe12.ps.gz, September 2001.

[7] B. Lewis. Debugging backwards in time. In *5th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Ghent, Belgium, September 2003.

[8] libdisasm. Libdisasm: x86 disassembler library, 2004. http://bastard.sourceforge.net/libdisasm.html.

[9] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, November 1998.

[10] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[11] M. S. Meier, K. L. Miller, D. P. Pazel, J. R. Rao, and J. R. Russell. Experiences with building distributed debuggers. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 70–79, Philadelphia, PA, USA, May 1996.

[12] I. Molner. Exec shield, new Linux security feature. http%-://people.redhat.com/mingo/exec-shield/ANNOUNCE-exec-shield, 2004.

[13] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *ACM workshop on parallel and distributed debugging*, San Diego, CA, USA, May 1993.

[14] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing*, Mineapolis, MN, USA, November 1992.

[15] R. H. B. Netzer, S. Subramanian, and J. Xu. Critical-path-based message logging for incremental replay of message-passing programs. In *14th Int. Conf. on Dist. Comp. Sys. (ICDCS)*, pages 404–413, Poznan, Poland, June 1994.

[16] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incrementar reexecution for debugging long-running programs. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, USA, June 1994. Also available as Brown University Technical Report CS-94-11.

[17] O. Oppitz. A particular bug trap: Execution replay using virtual machines. In *5th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Ghent, Belgium, September 2003.

[18] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. D. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *10th Symp. on Op. Sys. Principles (SOSP)*, pages 15–24, Orcas Island, WA, USA, December 1985.

[19] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *ACM workshop on parallel and distributed debugging*, Madison, WI, USA, May 1988.

[20] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *USENIX Winter Tech. Conf.*, New Orleans, LA, USA, January 1995.

[21] M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmuller. Record/replay for nondeterminsitic program executions. *Comm. of the ACM (CACM)*, 46(9), September 2003.

[22] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay and debugging. In *4th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Munich, Germany, August 2000.

[23] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *11th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS-XI)*, Boston, MA, USA, October 2004.

[24] J. Seward et al. Valgrind: A GPL'd system for debugging and profiling x86-linux programs. http://valgrind.kde.org/, 2004.

[25] M. W. Shapiro. RDB: A system for incremental replay debugging. Master's thesis, Dept of. Computer Science, Brown University, 1997.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and determinsitic replay for software debugging. In *USENIX Annual Tech. Conf.*, Boston, MA, USA, June 2004.

[27] A. Srivastaba and A. Eustace. ATOM: a system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, FL, USA, June 1994.

[28] D. Stodolsky, B. N. Bershad, and J. B. Chen. Fast Interrupt Priority Management in Operating System Kernels. *Usenix Workshop on Microkernels*, pages 105–110, September 1993.

[29] W. Vogels. File system usage in Windows NT 4.0. In *17th Symp. on Op. Sys. Principles (SOSP)*, pages 93–109, Kiawah Island, SC, USA, December 1999.

[30] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Cappriccio: Scalable threads for Internet services. In *19th Symp. on Op. Sys. Principles (SOSP)*, Bolton Landing, NY, USA, October 2003.

[31] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *6th Symp. on Op. Sys. Design and Impl. (OSDI)*, San Francisco, CA, USA, December 2004.

[32] L. D. Wittie. Debugging distributed C programs by real time replay. In *ACM workshop on parallel and distributed debugging*, pages 57–67, Madison, WI, USA, May 1988.