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

Yasushi Saito
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2005-46
March 7, 2005*

E-mail: ysaito@hpl.hp.com

Jockey, debugging, Linux

This paper describes *Jockey*, an execution record/replay tool for debugging distributed Linux programs. Jockey is implemented as a shared object file that is linked to the target program. Jockey rewrites those system calls and CPU instructions with non-deterministic effects and records and replays their invocations. It supports diagnosing long-running programs by taking periodic process checkpoints and enabling "time travel" to an arbitrary moment of execution. Jockey's performance overhead is negligible for CPU intensive applications, and 30% in the worst case of I/O intensive network servers. The implementation of Jockey is complicated by the fact that Jockey and the target program runs as a single process, sharing all the resources including the memory and the file-descriptor table. This paper describes techniques we developed to ensure deterministic execution record and replay of even buggy programs.

* Internal Accession Date Only
Approved for External Publication

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

Yasushi Saito
Hewlett-Packard Laboratories
ysaito@hpl.hp.com

Abstract

This paper describes *Jockey*, an execution record/replay tool for debugging Linux programs. Jockey is implemented as a shared object file that is linked to the target program. Jockey rewrites system calls and CPU instructions with non-deterministic effects and records and replays their invocations. It supports diagnosing long-running programs by taking periodic process checkpoints and enabling "time travel" to an arbitrary moment of execution. Jockey's performance overhead is negligible for CPU intensive applications, and 30% in the worst case of I/O intensive network servers. The implementation of Jockey is complicated by the fact that Jockey and the target program run as a single process, sharing all the resources including the memory and the file-descriptor table. This paper describes techniques we developed to ensure deterministic execution record and replay of even buggy programs.

1 Introduction

Jockey is a record/replay tool for Linux. It records the execution of an ordinary Linux program and later replays it deterministically. Jockey is designed to help debug interactive or distributed programs, such as FAB [28], that communicate with the operating system or other computers in a complex fashion. Jockey will be publicly available through http://www.freshmeat.net.

Traditional debuggers, such as gdb, provide comprehensive support for debugging single-node, sequential programs [24], but they are not as useful for distributed or interactive programs [13, 7]. We identify three key problems and discuss how Jockey alleviates them.

First, the execution of such programs is inherently non-deterministic. The behavior of each process will diverge, depending on interactions with the OS, the user, or other processes. As trivial examples, the `time` system call returns values depending on the time of day, and the `select` system call returns values depending on the state of the kernel network buffer at the moment. Jockey helps debug such non-deterministic programs by recording the execution of a process, logging every non-deterministic choice made by the program, and replaying it later 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., distributed hash tables), or they need substantial user interactions (e.g., spreadsheet). Simply reproducing the bug in such a system often tests a developer's patience. Jockey solves this problem by allowing checkpointing of the process state during execution. The developer can replay execution from any checkpoint and easily "time-travel" through executions to diagnose what exactly is wrong with the program.

Third, running a distributed system requires starting multiple processes on multiple computers, which is cumbersome and increases the turn-around time for program development. Jockey alleviates this problem because Jockey can record and replay an individual process—after running and recording the whole system, the developer can replay each process using a traditional debugger. Of course, this is also a limitation; Jockey could be less useful when one wants to look at the execution of multiple processes at once. We discuss this issue further in Section 5.2.

1.1 Jockey: goals and approaches

Jockey is designed with two particular goals in mind. First is *ease of use*. Jockey must be easy and safe to install and deploy; it should work without requiring modifications to the target source code or the debugger. Second is *generality*.

Jockey should be able to handle generic server programs, not just those written in a particular programming language or API, such as MPI or CORBA [14].

We achieve the first goal by implementing Jockey as a user-space library that runs as a part of the target program. In contrast to kernel-based approaches [31], Jockey's approach is safer and easier to use—it can be used by anyone without administrator privilege, a specially patched kernel, or debugger. It also allows the target program to control or extend Jockey easily. We discuss some of these features in Section 4. The second goal is achieved by recording and replaying events at a fairly low level—system calls and CPU instructions.

User-space implementation poses challenges and limitations. First, it cannot handle events such as thread context switching or shared-memory communications. Second, because Jockey and the target program run in the same protection domain, Jockey fundamentally cannot prevent a malicious program from destroying Jockey. We discuss these issues in more detail in Section 1.3.

Efficiency is a secondary goal for Jockey. Jockey is intended for use only during testing and debugging. Thus, as long as the slowdown by Jockey does not qualitatively change the program's behavior, developers should be able to live with some performance degradation.[1] We evaluate the overhead of Jockey in Section 5. It shows that the performance overhead is at most 30%, more often close to zero—well within our limit of tolerance.

1.2 Overview

We have implemented Jockey on Linux in C++. 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` upon startup causes Jockey to take control of the process. In this example, Jockey intercepts the call to the `read` system call made from `getc`. It logs the value read during the recording phase. When replaying, it reads the value from the log without actually reading from the random device. Jockey can also be invoked in several different ways, as shown in Figure 3.

[1] For example, the Valgrind dynamic binary instrumentation tool [29], as useful as it is, slows down the target tenfold. To run a program under Valgrind, one often needs to extend timeout parameters to prevent them from expiring prematurely.

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

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

```
% cc test.c
% LD_PRELOAD=libjockey.so \
  JOCKEYRC="replay=0" ./a.out # recording
38
% LD_PRELOAD=libjockey.so \
  JOCKEYRC="replay=1" ./a.out # replaying
38
```

Figure 2: *The most basic use of Jockey. The program `a.out` 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`.*

1.3 Limitations

Most of the limitations of Jockey stem from the fact that it runs as a part of the target process. The most serious problem is that Jockey could be broken by a seriously buggy or malicious target program—if it wishes, for example, the target program can destroy the heap memory used internally by Jockey. Jockey, however, also tries to avoid such problems by segregating resources used by Jockey and the target as much as possible, as discussed in Section 3.2.

Jockey cannot capture events caused by external entities. For example, memory access races that happen in multi-threaded programs cannot be replayed, because in-kernel context switches cannot be caught by Jockey. For this reason, Jockey does not support multi-threaded programs.[2] Similarly, Jockey does not support any program or API that interacts with other processes or the OS via shared memory; an example includes uDAPL [2] for memory-mapped network I/O.

Finally, ioctl commands needs to be handled individually. Jockey only offers supports for common ioctl commands. Support for other obscure ioctl commands needs to be added

[2] Jockey does support user-space threading packages, such as Capriccio [34].

```
1  % jockey --replay=0 ./a.out    # recording
2  a9
3  % jockey --replay=1 ./a.out    # replaying
4  a9
5
6  % LD_PRELOAD=libjockey.so \
7    ./a.out --jockey=replay=0    # recording
8  82
9  % LD_PRELOAD=libjockey.so \
10   ./a.out --jockey=replay=1    # replaying
11 82
12
13 % cc test.c -ljockey
14 % ./a.out --jockey=replay=0    # recording
15 c1
16 % ./a.out --jockey=replay=1    # replaying
17 c1
```

Figure 3: *Alternative ways of running a program under Jockey control. Lines 1 to 4 show how one can start the test program from the* `jockey` *frontend.* `jockey` *just sets environment variables* `LD_PRELOAD`*,* `JOCKYRC` *and* `execve`*'s the target program. Lines 6 to 11 shows an alternative method that passes a command-line parameter* `--jockey=`*. This parameter is parsed by* `libjockey.so` *when it loads. For this method to work, the target program must be designed to ignore a command-line string that starts with* `--jockey=`*. 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.*

by extending Jockey. Jockey provides support for doing that from the target program, as discussed in Section 4.

2 Related work

Execution record/replay has long been advocated as an effective debugging method [9, 27, 26].

2.1 Record/replay for sequential programs

Bugnet [35] was one of the earliest deterministic record/replay tools. It replayed parallel programs by recording external events such as I/O, and taking checkpoints of all processes periodically (by assuming loosely synchronized clocks). Bugnet supported only a special API, however, unlike Jockey which supports generic UNIX programs. Flashback [31] is the most recent work along this line. It offers virtually the same set of functionalities as Jockey—recording and replaying system calls and fork-based checkpointing—but Flashback is offered as a kernel module. As such, Flashback is less easy and safe to use than Jockey.

In a slightly different approach, some systems record and replay individual memory accesses [23, 20, 30, 10]. They have several advantages over event-based approaches like Jockey or Bugnet. First is that some of them enable *reverse execution*—literally stepping CPU instructions backward [23, 10]. They are also more generic because they need not know deeply about the semantics of system calls and other interactions with the environment. However, they require extensive program modification and have a large logging overhead. Netzer and others have proposed smart algorithms for determining the minimum set of load/store instructions that require tracing [20, 30], but these systems still require 1.5MB/second of log generation for a CPU-intensive program [20]. In contrast, Jockey only requires a few hundred bytes/second, as we show in Section 5.

2.2 Record/replay using virtual machines

Revirt is a virtual machine that records and replays the entire operating system [4]. Based on User-mode Linux [3], it records and replays low-level interrupts. Revirt has been proved to be useful for network intrusion detection and diagnosing kernel bugs [8]. Several other papers also propose distributed system emulation using virtual machines [5, 21]. While these systems are powerful, they are also cumbersome to use—for example, one needs to create a separate file system tree for each virtual machine. They are overkill when one is interested only in debugging user-space programs. Jockey is simpler and easier-to-use than these systems.

2.3 Record/replay for parallel and distributed programs

Deterministic record/replay has been most effective in parallel and distributed environments [26]. Indeed, two of the earliest tools, Bugnet [35] and Pan & Linton [23] expressly targeted such environments. Since then, many theoretical improvements have been proposed for both shared-memory parallel programs [17] and message-passing programs [18, 19]. Jockey does not yet support the replay of a distributed program—it can record and replay only an individual process within the system. As discussed in Section 5.2, we are not yet convinced that full-distributed-system record and replay is the worth the cost. That is perhaps because our main target (FAB) is essentially a "small" system that contains only a few tens of nodes. When we face

a truly large-scale system, e.g., when emulating a wide-area network, we might change our mind.

3 Implementation of Jockey

Jockey performs the following tasks upon program startup.

(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 78 Linux system calls, including `gettimeofday`, `recvfrom`, and `select`. Jockey logs the values generated by these calls during recording, and replays the log without actually executing the calls.

(2) Jockey does the same for CPU instructions with non-deterministic effects. We currently only patch `rdtsc`, the x86 instruction for reading the CPU's timestamp counter (cycle 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. We discuss checkpointing in more detail in Section 3.3.

(4) Jockey transfers control to the target program. The target runs as if Jockey does not exist. Jockey becomes active only when the target executes a system call or a non-deterministic CPU instruction.

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 function (step (3)), along with the challenges we had to overcome.

3.1 Instruction patching

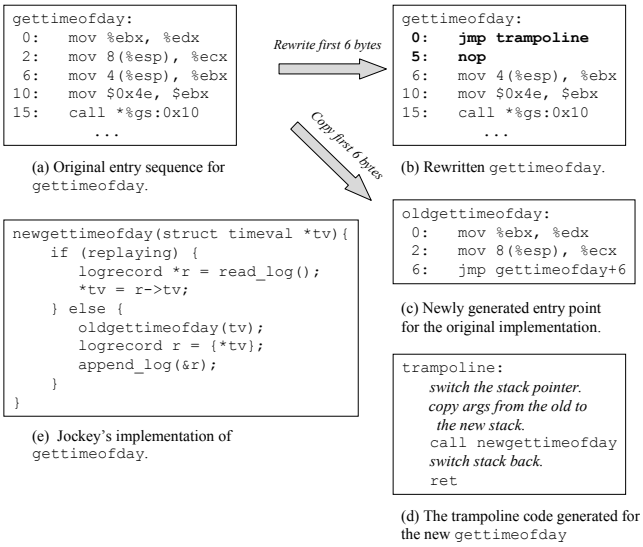
Figure 4 shows how the `gettimeofday` system call is recorded and replayed. (a) shows the first few instructions of the `gettimeofday` function in `libc.so`. When Jockey starts, it writes a `jmp` instruction in the first 5 bytes of the function, as shown in (b). If the 5th byte is in the middle of another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

One might wonder why `libjockey.so` does not just provide a new implementation of a system call with the same name. — `LD_PRELOAD` is frequently used to do exactly that. The reason is that doing so will miss system calls made inside `libc` or `ld.so`, for example, a call to `read` that would be made by `getc`. These internal calls are pre-resolved by the static linker (`ld`), and cannot be overridden by redefining 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 ELF header of each mapped shared object, 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 calls to `mmap` 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 operands and consults *libdisasm* [11], 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 shorten the startup latency, Jockey also employs cached-mode instruction patching. Here, after finishing the slow mode, Jockey writes the locations of non-deterministic instructions for each shared



(a) Original entry sequence for `gettimeofday`.

(b) Rewritten `gettimeofday`.

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

(d) The trampoline code generated for the new `gettimeofday`

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

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

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 parsing/rewriting module can also be used by the target program or a programmer-defined object file to monitor the execution of the target. We discuss this feature in more detail in Section 4.

3.2 Segregating resource usage

Jockey and the target application run as part of the same process. They share resources, including memory and file descriptors. Jockey must segregate the use of such resources to prevent Jockey from unnecessarily changing the target's behavior, and to minimize the chance of a misbehaving target program from breaking Jockey. We discuss Jockey's handling of three such resources, heap, stack, and file descriptors.

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

Instead, Jockey stores all its internal data in a `mmap`ed region at a virtual address (0x63000000) that is unlikely to be used accidentally by the target program. Jockey uses its own 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 maps and dynamic vectors. This design has considerably simplified the development of Jockey.

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

Stack: Jockey also segregates the use of stack. This is necessary to properly replay programs that access data beyond the stack pointer (e.g., accessing an on-stack array with a negative index). Figure 4, step (d) shows how this is done. In the first few instructions after it intercepts the call to `trampoline`, Jockey saves the stack pointer to an internal static variable, switches the stack pointer register to its internal memory, copies the parameters to gettimeofday (a pointer of 4 bytes) from the old to the new stack, and calls `newgettimeofday`. Once the new implementation returns, it then restores the stack pointer. This allows for deterministic replay even for a buggy program because the target's stack is not used above the original stack pointer.

File descriptors: Jockey must do its own file accesses every so often, for example, when opening an event log file, displaying a message, or taking 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, each file descriptor opened by Jockey is moved to a range that is not likely to be used by the application (430 and up). This is done by performing `dup2` immediately after `open`.

Gdb poses another problem. When starting the target process, gdb, for some reason, 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. We solve this problem by letting Jockey open dummy files for descriptors 0 to 9 on startup before returning control to the target program (it leaves descriptors in-

```
% jockey --checkpointfrequency=30 \
    --retaincheckpoints=5 \
    -- httpd -X
  ... later ...
% jockey --restore=log/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.*

herited from the parent process 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. This technique is valid for other situations when the target program is started with more than the standard number (three) of inherited file descriptors.

3.3 Checkpointing

Jockey allows a user to checkpoint the process state automatically. Figure 5 shows an example of taking a checkpoint of httpd (Apache) every 30 seconds.

Following the technique pioneered by libckpt [25] and flashback [31], Jockey checkpoints by forking the process and dumping the state of the child, while letting the parent continue running. Jockey reads the special file `/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 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 if the section is marked read-only.

We faced two particular problems for checkpointing, both related to dynamic linking:

Preventing ld.so brain damage One of the challenges during 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 read the checkpoint contents. If that 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 internal data structures.

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.

Exec shield Exec-shield is a facility found in some Linux kernels (e.g., Red Hat, Fedora Core) to thwart buffer-overflow attacks [15]. One feature that it provides 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 have no solution to this problem yet; we currently just demand that this feature be disabled by doing following:

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

3.4 Handling signals

Signals are handled in a way similar to [32]. Each signal delivery is first intercepted by Jockey. Jockey's signal handler simply records the parameters to the signal (signal number and the register context) 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.

3.5 Reducing logging overhead

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

- For requests to regular files or directories, Jockey performs "undo" logging. That is, for system calls that update a file, 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. For read-only system calls (such as `read`), it just reads directly from the file. A similar approach was used in the Discount checking system [12].

- For all other types of events—I/Os to sockets, pipes, fifos, devices, or `select`, `gettimeofday`, or `rdtsc`—Jockey performs "redo" logging. Jockey logs the value produced by the event during recording. During replay, Jockey just reads the values from the log without executing the actual system call. Thus, system calls that update a file become no-op during replay.

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, stdout, stderr) are 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 [22, 1, 33]. 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 file I/O system calls [12].

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

3.6 Handling memory-mapped I/Os

Jockey currently requires manual efforts to handle I/Os done through memory-mapped files. To replay such I/Os, the programmer must instrument the program and notify Jockey every time an I/O is made. Figure 6 shows an example. `jockey_prepare_mmaped_write` logs the original image of the specified region, so that Jockey can restore the image before replaying.

This approach is admittedly error-prone. As an alternative, we could apply "page diffing" by write-protecting mmaped pages and trapping accesses to them [6]. Implementing such an approach is future work.

```
1  int fd = open("blah", O_RDWR);
2  char *base = mmap(NULL, size,
3      PROT_READ|PROT_WRITE, MAP_SHARED,
4      fd, 0);
5  ...
6  char *p = base + offset
7  char *str = "Hello, World";
8  jockey_prepare_mmaped_write(fd, offset,
9      p, strlen(str) + 1);
10 strcpy(p, str);
```

Figure 6: *An example of memory mapped I/O. Before the program modifies a memory mapped region, it must notify Jockey (line 6) the file descriptor (fd), the file offset (offset), the mmap pointer (p), and the size of modification.*

4 Controlling Jockey

One of the virtues of Jockey being a part of the target process is that it easily allows the user or the target program to customize Jockey. This section lists some of the knobs that are available to change the behavior of Jockey.

`jockey_set_fork_trace_mode`: By default, upon `fork`, Jockey continues tracing only the parent and disables tracing the child. This function, when called by the target program, changes that behavior—whether to trace only the parent, the child, or both after forking. This function can be used, for example, for daemon-type programs that fork to detach themselves from the parent process.

`jockey_checkpoint(`*`path`*`)`: This function can be called by the target program to take a checkpoint manually.

`jockey_redirect_calls(`*`name, newproc, argsize`*`)`: This function is used to transfer the control to *newproc* whenever function *name* is called. Parameter *argsize* is the size of the on-stack parameters to function *name*. This feature can be used, for example, to provide record/replay functionality for an obscure `ioctl` command used only by the target application. Function `jockey_interpose_calls(`*`name, newproc, argsize`*`)` is similar, but it transfers the control back to the original function after *newproc* finishes execution.

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

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

Figure 7: *The upper listing, `testprogram.c` is a simple program that calls function `bar` many times. The lower listing, `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_breakpoint` is a function, defined in Jockey, that does nothing.*

```
% cc -c checker.c
% gdb testprogram
(gdb) set env LD_PRELOAD=libjockey.so
(gdb) b jockey_breakpoint
(gdb) run --jockey=replay=1;\
checker=checker.o
```

Figure 8: *Running the user-defined checker. `checker.c` is the name of the user-defined checker program. Function `bar_checker` is called every time `bar` executes. It examines the status of the program without affecting the target execution and calls the breakpoint at the designated point.*

User-defined invariant checker: Jockey allows an arbitrary object file to be executed during replay. Figure 7 shows an example. Here, let's assume that we ran the upper program under Jockey and found that procedure `bar` behaves anomalously when `i == 95999`. We could diagnose this situation by setting a breakpoint on `bar` in gdb and waiting until it hits 95999 times, but Jockey offers a quicker alternative, as shown in Figure 8.

The implementation of this feature is complicated, because we cannot rely on the dynamic linker to link the object into the target program—doing so would alter the heap structure of the target (Section 3.3). Jockey uses a static linker (`ld`) instead to resolve symbols in the checker object. Jockey first picks a virtual address unlikely to be used by the target application (0x62000000). It then discovers the addresses of all public symbols in the process (including those exported by Jockey) by invoking `nm` for each shared object. This information is then passed to `ld`, which generates a binary file with all symbols resolved. Jockey then reads the file contents into memory and executes it.

5 Evaluation

This section reports performance and space overheads of Jockey and discusses experiences using Jockey to debug real programs.

5.1 Performance and log-space overheads

We tried Jockey on a variety of programs, listed below. The evaluation was performed on a Linux computer running Fedora Core 3 (kernel 2.6.9) with a 1.5GHz Pentium-M CPU, 512MB of memory, a 7200 rpm ATA hard disk, and a 1Gb/s Ethernet interface.

g++ g++ 3.4.2 compiling a small C++ program that uses an STL `map`. The overhead is the sum for the frontend (`g++`), the compiler itself (`cc1plus`), and the 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, without forking), serving 100000 HTTP GET requests for a 0.5KB file from httperf [16].

FAB A four-brick FAB cluster [28] serving 80000 random 1KB I/O requests. FAB is a distributed disk array system running on a cluster of PCs.

g++ is an example of a short-running, CPU-intensive program, which is not Jockey's primary targets. This example still shows that Jockey has a very low log-space overhead compared to approaches that involve memory-access logging [20], which could consume up to a few megabytes per second for logging. For gcc, most of the performance overhead is due to process checkpointing that happens at the beginning of the execution (Section 3).

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 space overhead 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.*

Xclock and emacs are examples of interactive applications. Jockey shows reasonable log-space overheads. Jockey is able to replay their execution extremely fast, because it need not wait for timeouts or user inputs during replay. This translates to more efficient debugging activities.

Apache and FAB are examples of network servers. FAB represents the worst case for Jockey. Not only does FAB perform large amount of network I/O, it also overwrites existing files repeatedly, resulting in a large amount of logging traffic (Section 3.5). In comparison, Apache has a far lower overhead because it only performs read-only accesses to `index.html` and appends to log files.

5.2 Experiences

Jockey has been regularly used for FAB development. It has been surprisingly effective in diagnosing bugs that exhibit quickly, e.g., during the first I/O request from the client. Jockey reduces the debugging "turn-around" time, because it allows the developer to replay a single process instead of restarting the entire cluster. This makes the developer more productive.

Jockey has been most useful when debugging non-deterministic bugs that happen during stress or regression tests. Before we had Jockey, we had to restart the cluster many times, each time with a slightly different set of "printf" statements, with the hope that we would eventually reproduce and catch the error. Jockey allows the developer to reproduce the bug reliably as often as he wishes. Diagnosing such bugs, however, is still difficult even with Jockey. The real "cause" of the bug often happens a few hundred requests before it exhibits, often in a different process. The programmer is forced to replay the execution of multiple processes repeatedly to locate the cause. People have argued for deterministic distributed-system replay for diagnosing such bugs (Section 2), but we currently doubt if it is worth the effort. Debugging a long-running distributed system requires a deep understanding of the program anyway, and single-process independent replay as provided by Jockey seems to achieve most of the benefits.

There are a few Jockey features that sound useful in theory, but have turned out to be not quite so in practice. The concept of "time travel" using periodic automatic checkpoints is one such idea (Section 3.3). It is a powerful, but difficult-to-use tool. Without proper debugger support [8], the developer needs to restart the debugger every time he or she wants to switch to a different checkpoint. This manual effort quickly exhausts the developer. Another feature is user-defined invariant checking (Section 4). The problem is twofold. First is the lack of proper debugger support—writing and compiling a program every time one wants to debug is cumbersome. Second is that the kind of things that the checker can do is rather limited—for example, it cannot intercept calls in the middle of function execution, nor can it inspect the local variable values in the call chain.

6 Conclusion

This paper described Jockey, a user-space library for deterministic record/replay debugging. Jockey runs as a part of the target program and intercepts calls to non-deterministic system calls and CPU instructions. It logs the effects of these operations during recording, and replays them from the log during replay. It has a small performance and log-space overhead. Jockey has been extensively used to develop FAB, a distributed disk-array system running on a cluster of PCs.

References

[1] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *13th Symp. on Op. Sys. Principles (SOSP)*, pages 198–212, Pacific Grove, CA, USA, October 1991.

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

[3] Jeff Dike et al. User-mode Linux home page. http:/-/user-mode-linux% -.sourceforge.net/, 2005.

[4] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter 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.

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

[6] Antony L. Hosking, Eric W. Brown, and J. Eliot B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In *19th Int. Conf. on Very Large Data Bases (VLDB)*, pages 429–440, Dublin, Ireland, August 1993.

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

[8] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Tech. Conf.*, Anaheim, CA, USA, April 2005.

[9] Lap Chung 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.

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

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

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

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

[14] Michael S. Meier, Kevan L. Miller, Donald P. Pazel, Josyula R. Rao, and James R. Russell. Experiences with building distributed debuggers. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 70–79, Philadelphia, PA, USA, May 1996.

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

[16] David Mosberger. httperf—a tool for measuring web server performance. http://www.hpl.hp.com/personal-/David_Mosberger/httperf.html, 2001.

[17] Robert 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.

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

[19] Robert H. B. Netzer, Sairam Subramanian, and Jian 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.

[20] Robert H. B. Netzer and Mark 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.

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

[22] John K. Ousterhout, Herv Da Costa, David Harrison, John A. Kunze, Michael D. Kupfer, and James 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.

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

[24] Vern Paxson. A survey of support for implementing debuggers. http://citeseer.ist.psu.edu-/paxson90survey.html, 1990.

[25] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. In *USENIX Winter Tech. Conf.*, New Orleans, LA, USA, January 1995.

[26] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmuller. Record/replay for non-determinsitic program executions. *Comm. of the ACM (CACM)*, 46(9), September 2003.

[27] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *4th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Munich, Germany, August 2000.

[28] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan 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.

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

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

[31] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and determinsitic replay for software debugging. In *USENIX Annual Tech. Conf.*, Boston, MA, USA, June 2004.

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

[33] Werner 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.

[34] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Cappriccio: Scalable threads for Internet services. In *19th Symp. on Op. Sys. Principles (SOSP)*, Bolton Landing, NY, USA, October 2003.

[35] Larry 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.