Replay Debugging for Distributed Applications

Dennis Michael Geels

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-163
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-163.html

December 8, 2006

Acknowledgement

This work was supported by the Fannie and John Hertz Foundation Graduate Fellowship program.

Replay Debugging for Distributed Applications

by

Dennis Michael Geels

B.S. (Rice University) 1999
M.S. (University of California at Berkeley) 2002

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Ion Stoica, Chair
Professor Scott Shenker
Professor John Chuang

Fall 2006

The dissertation of Dennis Michael Geels is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Replay Debugging for Distributed Applications

Abstract

Replay Debugging for Distributed Applications

by

Dennis Michael Geels
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Researchers in networks and computer systems have developed exciting new distributed applications in recent years; however, adoption of real-world prototypes has been slow. The development of stable, usable services has been hindered by the tremendous effort required to debug distributed applications that are deployed across the Internet. We believe that more powerful debugging tools are needed to address this problem. This dissertation presents the progress we have made on this front, in the form of two new tools, `Liblog` and `Friday`.

The first, `Liblog`, is a *replay debugging* library for `libc`- and POSIX-based distributed applications. It logs the execution of deployed application processes and replays them deterministically, faithfully reproducing race conditions and non-deterministic failures, enabling careful offline analysis.

To our knowledge, `Liblog` is the first replay tool to address the requirements of large distributed systems: lightweight support for long-running programs, consistent replay of arbitrary subsets of application nodes, and operation in a mixed environment of logging and non-logging processes. In addition, it runs on generic Linux/x86 computers without special hardware or kernel patches and supports unmodified application executables.

The second tool, `Friday`, combines the deterministic replay provided by `Liblog` with the power of symbolic, low-level debugging and a simple language for expressing higher-level distributed conditions and actions. `Friday` allows the programmer to

understand the collective state and dynamics of a distributed collection of coordinated application components, as part of the debugging process.

This dissertation presents the design of `Liblog` and `Friday`, an evaluation of the performance overhead that they impose at runtime, and a set of case studies that illustrate the new functionality enabled for real distributed applications.

Professor Ion Stoica
Dissertation Committee Chair

For Cheryl

Contents

List of Figures **iv**

List of Tables **v**

1 Introduction **1**
1.1 Step One: `Liblog` . . . 1
1.1.1 Requirements . . . 2
1.2 A Need for *Distributed Comprehension* . . . 4
1.3 Contributions . . . 5
1.4 Are `Liblog` and `Friday` Right For You? . . . 6

2 Background **8**
2.1 Deterministic Replay . . . 8
2.2 Log-based Debugging Without Replay . . . 9
2.3 Off-line vs. On-line . . . 11

3 Liblog **12**
3.1 Design . . . 12
3.1.1 Shared Library Implementation . . . 12
3.1.2 Message Tagging and Capture . . . 13
3.1.3 Central Replay . . . 14
3.2 Challenges and Solutions . . . 15
3.2.1 Signals and Thread Replay in Userland . . . 15
3.2.2 Unsafe Memory Access . . . 16
3.2.3 Consistent Replay for TCP . . . 17
3.2.4 Finding Peers in a Mixed Environment . . . 18
3.2.5 Replaying Multiple Processes . . . 19
3.2.6 Migratable Checkpoints . . . 20
3.3 Limitations . . . 21
3.4 Evaluation . . . 23
3.4.1 Wrapper Latency . . . 24
3.4.2 Network Performance . . . 25

3.4.3 Log Bandwidth . . . 26
3.4.4 Checkpoint Overhead . . . 28
3.4.5 Evaluation Summary . . . 28
3.5 Experience . . . 30
3.5.1 Programming Errors . . . 30
3.5.2 Broken Environmental Assumptions . . . 31
3.5.3 Broken Usage Assumptions . . . 32
3.5.4 Self-Debugging . . . 33
3.5.5 Injected Bugs . . . 34
3.6 Summary . . . 35

4 Friday **36**
4.1 Design . . . 36
4.1.1 Distributed Watchpoints and Breakpoints . . . 37
4.1.2 Commands . . . 40
4.1.3 Limitations . . . 44
4.2 Case Studies . . . 46
4.2.1 Routing Consistency . . . 46
4.2.2 A Reliable Communication Toolkit . . . 53
4.3 Evaluation . . . 56
4.3.1 Micro-benchmarks . . . 57
4.3.2 Micro-benchmarks: Scaling of Commands . . . 59
4.3.3 Micro-benchmarks on Replayed Chord . . . 60
4.3.4 Case Studies . . . 62
4.4 Summary . . . 64

5 Conclusion **65**
5.1 Future Work . . . 65
5.2 Afterword . . . 67

Bibliography **69**

List of Figures

3.1 `Liblog` architecture 13
3.2 State machine for incoming TCP streams 18
3.3 Overhead: wrapper latency 24
3.4 Network performance: datagram rate 25
3.5 Network performance: datagram throughput 26
3.6 Network performance: TCP throughput 27
3.7 Network performance: message latency 28
3.8 Overhead: disk bandwidth 29
3.9 Overhead: disk space 29
3.10 Checkpoint latency 30
3.11 Checkpoint size 31

4.1 `Friday` architecture 37
4.2 Ring consistency example 48
4.3 Watchpoint latency breakdown 59
4.4 Watchpoint latency at scale 60
4.5 Slowdown on Chord 62
4.6 Case study performance 64

List of Tables

4.1 `Friday` micro-benchmarks 56
4.2 Slowdown on Chord 61
4.3 Case study performance 63

Acknowledgments

I received a great deal of help over the past seven years, for which I am grateful.

First, I thank my fellow Rice alumni, for keeping me sane and proving that graduate school is indeed finite.

I thank my fellow graduate students, both from my original research project and also from the network group, for sharing ideas, cooperating on experiments, and being good friends. I came to Cal looking for bright colleagues, and I was not disappointed.

The work presented in this dissertation includes contributions from current projectmates Gautam Altekar, Petros Maniatis, and Timothy Roscoe. Your suggestions and hard work helped immensely. I treasure our friendship and thank you all.

I thank Professor Jonathan Chuang for finding time for yet another dissertation committee. You are too generous.

I thank Professor Scott Shenker for advising a frustrated and disillusioned student who needed to start over.

I thank my advisor, Ion Stoica, for three years of good advice and unbelievable patience. The university needs more professors like you.

Most of my graduate career was funded by the Fannie and John Hertz Foundation Graduate Fellowship. The Foundation gave me much more than just putting food on the table and computers on the desk; the freedom that this fellowship gave me allowed me to restart my research when other avenues were closed. I would not have finished my degree without it.

My parents, Daniel and Joanne, gave me the education that got me into graduate school and the stubbornness that made me stay. I'm sure you meant well. Thank you.

I thank my sister and brother for being more encouraging than siblings tend.

Finally, I thank my glorious wife Cheryl for supporting me, putting up with graduate school life, and taking care of the kids while I spent too many weekends and evenings working. I don't deserve you, but I'll keep you anyway.

Chapter 1

Introduction

Over the past few years, research has produced new algorithms for routing overlays, query processing engines, byzantine fault-tolerant replication, and distributed hash tables. Popular software like peer-to-peer file sharing applications suggests that interest in distributed applications is not restricted to academic circles.

But debugging is hard. Debugging distributed applications is harder still, and debugging distributed applications deployed across the Internet is downright daunting. We believe that the development of new services has been held back by this difficulty and that *more powerful debugging tools are needed.*

1.1 Step One: `Liblog`

A distributed application is a collection of processes running on machines spread across a network (for our purposes, the Internet). The individual processes may be analyzed independently, and debugging existing tools can catch common "local" errors such as unsafe memory accesses and thread synchronization errors. Unfortunately, these tools do not address the new problems that arise when the processes are composed across an unpredictable and lossy network. Races between network messages produce non-deterministic behaviour. Message delay and failure ensure that the aggregate application state is only rarely globally consistent.

Simulation and small-scale test deployments help developers evaluate aggregate

system behaviour in a relatively easy environment. With a simulator, the developer has full power to repeat the same execution across multiple experiments, and the state of each application process is available locally for examination. Test deployments complement simulation by adding more realistic network and host machine behaviour. Using local clusters and small, or even emulated, networks, developers may carefully control the degree of realism exposed to their applications.

However, once deployed, distributed applications will reach states that were not tested, and the underlying network will fail in ways that the developer did not anticipate. Long-running services are particularly prone to the slow-developing and non-deterministic, low-probability faults that resist detection during the testing phase.

And once the application is deployed, race conditions and internal state are difficult to observe. Developers rely on application-level logging and `printf` statements, but these techniques only help if the developer chooses to expose the affected internal state before the fault manifests. These types of bugs are generally impossible to reproduce locally, where analysis would be simpler. This *limited visibility* is the core problem for debugging distributed applications. We have developed a new debugging tool, `Liblog`, to address it.

1.1.1 Requirements

We designed this tool to help fix non-deterministic failures in deployed, distributed applications. This goal imposed several requirements on our design.

Deterministic Replay: First and foremost, deployed applications need *logging and replay*. Normal debuggers monitor an application's execution synchronously, so that the process can be paused immediately when a failure, signal, or breakpoint occurs. This approach is infeasible for real, deployed systems for three reasons. First, the latency of a synchronous connection to a remote debugger would significantly slow down the application. Second, pausing the process (or processes, if the developer wished to look at global state) at breakpoints would be unacceptable for real, deployed services, which interact continuously with peer services and clients. Third, real networks are

not stable enough to maintain a persistent connection to each process.

Thus debugging must be asynchronous. Each process records its execution to a local log, with sufficient detail such that the same execution can be replayed later. We should follow the same code paths during replay, see the file and network I/O, and even reproduce signals and other IPC. The replay could run in parallel with the original execution, after the original process dies, or even on a completely different machine.

Continuous Logging: In order to record the manifestation of slow-developing and non-deterministic, low-probability faults, the logging infrastructure must remain active at all times. We must operate under the assumption that more bugs are always waiting. Also, any slight perturbations in application behaviour imposed by the debugger becomes the "normal" behaviour. Removing it then would be a perturbation that might activate so-called "heisenbugs": bugs that behave differently or disappear under observation.

If the debugging system required significant resources, the cost in performance (or faster hardware) might be prohibitive. Fortunately, many types of distributed applications consume relatively few local resources themselves. Whereas network bandwidth and latency might be precious, we often have extra CPU cycles and disk space to accommodate our logging tools. In particular, if we confine ourselves to a small processing budget, the network will remain the performance bottleneck, and the application will exhibit little slowdown.

Consistent Group Replay: We are particularly interested in finding *distributed bugs*, such as race conditions and incorrect state propagation. This kind of error may be difficult or impossible to detect from the state of any one process. For example, transient routing loops are only visible when the aggregate state of multiple routers is considered.

So we must be able to see snapshots of the state across multiple processes and to trace message propagation from machine to machine. Naturally, true snapshots are impossible without synchronized clocks (cf. [Lam78]), but we can require that each

machine is replayed to a *consistent* point, where no message is received before it has been sent.

Mixed environment: Most applications will not run our software, particularly client software and supporting services like DNS. This fact becomes a problem if we require coordination from communication peers during logging or replay, as we generally must in order to satisfy the previous requirement (consistent replay). Since we do not operate in a closed system, our tools must understand the difference between cooperating and non-cooperating peers and treat each appropriately.

1.2 A Need for *Distributed Comprehension*

Correct operation of a distributed application is frequently a function not only of single-component behaviour, but also of the global collection of states of multiple components. For instance, in a message routing application, individual routing tables may appear correct while the system as a whole exhibits routing cycles, flaps, wormholes or other inconsistencies.

To face this difficulty, ideally a programmer would be able to debug *the whole application*, inspecting the state of any component at any point during a debugging execution, or even creating custom invariant checkers on global predicates that can be *globally* evaluated continuously as the system runs. In the routing application example, a programmer would be able to program her debugger to check continuously that no routing cycles exist across the running state of the entire distributed system, with the same ease as we read the current state of program variables in typical symbolic debuggers.

`Friday`, the second system we present in this dissertation, is a first step towards realizing this vision. `Friday` builds upon `Liblog`, which can capture the distributed execution of an application and replay the captured execution trace within a symbolic debugger in a single location. But simple replay does not supply the global view of the system required to diagnose emergent misbehaviour of the application as a whole. For this *distributed comprehension*, `Friday` extends the debugger's programmability

for complex predicates that involve the *whole* state of the replayed system.

`Friday` combines the flexibility of symbolic debuggers on each replayed node with the power of a general-purpose, embedded scripting language. Bridging the two allows a single global invariant checker script to monitor and control the global execution of multiple, distinct replayed components. To our knowledge, this is the first replay-based debugging system for unmodified distributed applications that can track arbitrary global invariants at the fine granularity of source symbols.

1.3 Contributions

The first contribution of our work is the design and evaluation of a debugging tool, `Liblog`, that satisfies each of the requirements listed in section 1.1.1. Previous projects have developed logging and replay tools that focus on either low overhead or providing consistent replay, but we have addressed both. Furthermore, to the best of our knowledge, `Liblog` is the first tool that (1) provides consistent replay in a mixed environment, or (2) allows consistent replay for arbitrary subsets of application processes.

In addition, `Liblog` requires neither special hardware support nor patches to privileged system software. Also, it operates on unmodified C/C++ application binaries at runtime, without source code annotations or special compilation tools. Multithreading, shared memory, signals, and file and network I/O all work transparently.

Finally, we designed `Liblog` to be simple to use. Logging only requires running our start-up script on each machine. Our replay tools make debugging as easy as using a symbolic debugger with local applications: they automate log collection, export the traditional debugger interface to the programmer, and even extend that interface to support consistent replay of multiple processes and tracking messages across machines.

`Friday` incorporates two additional contributions. First, it provides primitives for detecting events in the replayed system based on data (watchpoints) or control flow (breakpoints). These watchpoints and breakpoints are *distributed*, coordinating detection across all nodes in the replayed system, while presenting the abstraction of

operating on the global state of the application.

Second, `Friday` enables users to attach arbitrary *commands* to distributed watchpoints and breakpoints. `Friday` gives these commands access to all application state as well as a persistent, shared store for saving debugging statistics, building behavioural models, or shadowing global state.

We have built an instance of `Friday` for the popular Gnu Project Debugger [GDB], using Python as the script language, though our techniques are equally applicable to other symbolic debuggers and interpreted scripting languages.

1.4 Are `Liblog` and `Friday` Right For You?

Many distributed applications can benefit from `Friday`'s functionality, including both fully distributed systems (e.g., overlays, protocols for replicated state machines) and centrally managed distributed systems (e.g., load balancers, cluster managers, centralized resource managers, grid job schedulers). Using `Friday`'s facilities, developers can evaluate global conditions during replay to validate a particular execution for correctness, to debug distributed problems, to catch inconsistencies between a central management component and the actual state of the distributed managed components, and to express and iterate behavioural regression tests. For example, in implementing an IP routing protocol that drops an unusual number of packets, a developer might hypothesize that the cause is a routing cycle, and use `Friday` to verify cycle existence. If the hypothesis is true, the developer can further use `Friday` to capture cycle dynamics (e.g., are they transient or long-lasting?), identify the likely events that cause them (e.g., router failures, processor overload, congestion on the control plane), and finally identify the root cause by performing step-by-step debugging and analysis on a few instances involving such events, all the time without the need for recompilation or annotation of source code.

However, `Friday` does not come without limitations. First, `Friday` inherits several limitations of `Liblog`, such as large storage requirements for logs and an inability to execute threads in parallel on multi-processor or multi-core machines.

We designed `Liblog` with lightweight distributed applications like routing overlays

in mind. We assume that the host machines have spare resources–specifically CPU, memory, network, and disk–that we can apply to our debugging efforts.

Although it can correctly log and replay general C/C++ applications, the runtime overhead imposed could outweigh the benefits for resource-intensive systems like streaming video servers or heavily multi-threaded databases. We quantify this overhead in Section 3.3.

Chapter 2

Background

Before we dive into the details of our new debugging tools, we present an overview of previous research in the area, in order to develop context for our own design decisions.

2.1 Deterministic Replay

Deterministic replay has been a reasonably active research subject for over two decades. Most of this work centered on efficient logging for multiprocessors and distributed shared memory computers; for an overview of the field we recommend an early survey by Dionne et al [DFD96] and later ones by Huselius [Hus02] and Cornelis et al [CGC+03].

None of these previous projects focused on deployed, distributed applications or addressed the technical challenges raised by that set of requirements. In particular, our support of consistent group replay in a mixed environment is unique, and we are the first to support multi-threaded applications without kernel support.

On the other hand, the core techniques of logging and replay have been explored thoroughly, and we borrowed or reinvented much from earlier projects. Specifically, Lamport clocks [Lam78] have been used for consistent replay of MPI [RBdK99] and distributed shared memory [RZ97]. Replaying context switches to enforce deterministic replay in multi-threaded apps was based on DejaVu [KSC00], which built the

technique into a Java Virtual Machine. Finally, some projects have integrated GDB and extended its interface to include replay commands [SKAZ04, KDC05], but only `Liblog` seamlessly provides consistent replay across multiple processes.

Our library-based implementation most closely resembles Jockey [Sai05]; they also have simple binary-rewriting functionality to detect use of non-deterministic applications. Flashback [SKAZ04] also has many similarities, but they chose to modify the host OS. Their modifications enable very efficient checkpoints and (potentially) simplified thread support. We chose instead to implement all of `Liblog` at user level in order to maximize its portability and to lower barriers to use on shared infrastructure. Also, our support for multiple threads, migratable checkpoints, and consistent replay across machines makes `Liblog` more appropriate for distributed applications.

Much research has also been devoted to replay debugging via virtualization, which can capture system effects below the system library level. Harris first made the case for pervasive, distributed debugging [Har02] through virtualization. Several projects have pursued that agenda since [SKAZ04, KDC05, JKDC05], albeit only for single-thread, single-process, or single-machine applications. Furthermore, symbolic debugging in such systems faces greater challenges than with `Friday`, since the "semantic gap" between application-defined symbols and the virtual machine interface must be bridged at some computational and complexity cost.

The DejaVu project [KSC00] shared our goal of replaying distributed applications. Like `Liblog`, they support multi-threaded applications and consistently replay socket-based network communication. Unlike `Liblog`, they targeted Java applications and built a modified Java Virtual Machine. Thus they addressed a very different set of implementation challenges. Also, they do not support consistent replay in a mixed environment, although they do sketch out a potential solution.

2.2 Log-based Debugging Without Replay

Moving away from replay debugging, many systems focus on extracting execution logs and then mining those logs for debugging purposes [Sno88, WSY91, AMW+03, BDIM04, CECZ05, CAK+04, HM93]. Such systems face the challenge of reconstruct-

ing meaningful data- and control-flow from low-level logged monitoring information. `Liblog` circumvents this challenge, since it can fully inspect the internal state of the nodes in the system during a replay of the traced execution and, as a result, need not guess causality (as with black-box approaches) or recompile the system (as with annotation-based systems).

Notable logging-based work in closer alignment with `Liblog` comes from the Bi-directional, Distributed BackTracker (BDB) [KMLC05] and Pip [RWM+06]. BDB tracks and logs causality among events within a distributed system. These logs allow tracing identified back-door programs backwards to the events that enabled them or forwards to their further implications. Most of the causality tracing rules used by BDB can be implemented using `Liblog`, except for those relying on kernel-level interactions, which lie beyond our library tracing granularity. However, a bidirectional distributed backtracker implemented with `Liblog` may be able to take advantage of successive replays to refine causality tracking for BDB rules that are inherently "noisy," e.g., directory listing filesystem operations.

Pip [RWM+06] works by comparing actual behaviour and expected behaviour to expose bugs. Such behaviours are defined as orderings of logged operations at participating threads and limits on the values of annotated and logged performance metrics. They can be extracted automatically from the logs, or specified by the programmer and matched against the logs. Unlike Pip, `Friday` does not learn behaviours from a running system and has a much cruder, textual interface. However, `Friday` offers programmers greater flexibility in describing and capturing system behaviours for two reasons. First, it applies not only to manually annotated events but to any source symbol—without need for manual instrumentation; this means that behaviour exploration on a trace can be refined, redefined, and extended without the need to collect new traces that include new metrics or new events logged. Second, `Friday` can encode dynamic behaviours that go beyond pattern matching against logs. Such are the parametrized link symmetry checks of Section 4.2.1, where the identities of the pairs of processes that must satisfy the symmetry pattern are unknown until runtime and change as the system evolves.

2.3 Off-line vs. On-line

Most distributed debuggers in the literature, like `Friday`, are off-line: they perform their operations on logs or traces that have been collected during an execution of the system. In contrast, the P2 debugger [SMRD06] operates on the P2 [LCH+05] system for the high-level specification and implementation of distributed systems. Like `Friday`, this debugger allows programmers to express distributed invariants in the same terms as the running system, albeit at a much higher-level of abstraction than `Friday`'s `libc`-level granularity. Unlike `Friday`, P2 targets on-line invariant checking, not replay execution. As a result, though the P2 debugger can operate in a completely distributed fashion and without need for log back-hauling, it can primarily check invariants that have efficient on-line, distributed implementations. `Friday`, however, can check expensive invariants such as the existence of disjoint paths, since it has the luxury of operating outside the normal execution of the system.

Further afield, many distributed monitoring systems can perform debugging functions, typically with a statistical bend [vRBDV02, YD04, BAS04]. Such systems employ distributed data organization and indexing to perform efficient distributed queries on the running system state, but do not capture control path information equivalent to that captured by `Friday`.

Chapter 3

Liblog

We built `Liblog` by combining existing technology in new ways and extending the state of the art as necessary. In the following sections, we will present an overview of the resulting design (Section 3.1) and then explain in more detail the new technical challenges that arose, along with our solutions (Section 3.2).

3.1 Design

In this section we present an overview of `Liblog`'s design, highlighting the decisions that we made in order to satisfy the requirements listed above.

3.1.1 Shared Library Implementation

The core of our debugging tool is a shared library (the eponym `liblog`), which intercepts calls to `libc` (e.g., `select`, `gettimeofday`) and logs their results. Our start-up scripts use the `LD_PRELOAD` linker variable to interpose `liblog` between `libc` and the application and its other libraries (see Figure 3.1). `Liblog` runs on Linux/x86 computers and supports POSIX C/C++ applications.

We chose to build a library-based tool because operating in the application's address space is efficient. Neither extra context switches nor virtualization layers are required. Alternative methods like special logging hardware [NM92, XBH03, NPC05]

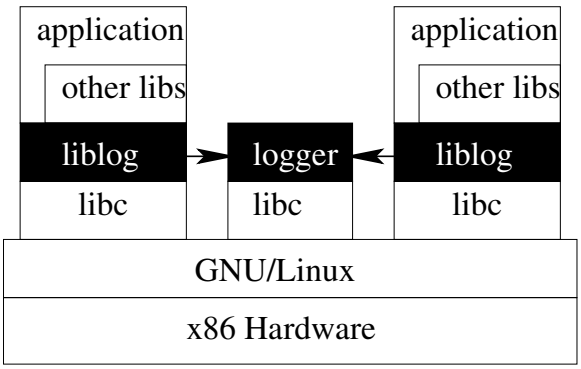

Figure 3.1: *Logging:* `liblog` intercepts calls to `libc` and sends results to `logger` process. The latter asynchronously compresses and writes the logs to local storage.

or kernel modifications [TH00, SKAZ04] can be even faster, but we found these solutions too restrictive for a tool that we hope to be widely adopted and deployed.

Another promising alternative is to run applications on a virtual machine and then to log the entire VM [KDC05, SH, HH05]. We rejected it because we believe that VM technology is still too difficult to deploy and too slow for most deployed services.

On the other hand, there are serious drawbacks of a library implementation. First, several aspects of observing and controlling applications are more difficult from within the address space, most notably supporting multiple threads and shared memory. We will discuss these challenges in Section 3.2.

Fundamentally, however, operating in the application's address space is neither complete (we cannot replay all non-determinism) nor sound (internal state may become corrupted, causing mistakes). We will discuss such limitations in Section 3.3.

Nevertheless we believe that the combined efficiency and ease of use of a library-based logging tool makes this solution the most *useful*.

3.1.2 Message Tagging and Capture

The second defining aspect of our logging tool is our approach to replaying network communication. We *log the contents* of all incoming messages so that the receiving process can be replayed independently of the sender.

This flexibility comes at the cost of significant log space (cf. Section 3.4) but is

well justified. Previous projects have tried the alternative, replaying *all* processes and regenerating message contents on the sender. We cannot do so because we operate in a mixed environment with non-logging processes. Even cooperating application logs may be unavailable for replay due to intervening disk or network failure.

So far we satisfy one requirement, but we must be able to coordinate these individual replays in order to provide another, Consistent Group Replay. For this purpose, we embed 8-byte Lamport clocks [Lam78] in all outgoing messages during execution and then use these virtual clocks to schedule replay. The clock update algorithm ensures that the timestamps in each log entry respect the "happens-before" relationship. They also provide a convenient way to correlate message transmission and reception events, so we can trace communication from machine to machine.

To make the virtual clocks more intuitive, we advance them at the rate of the local machine clock. If the machine clocks happen to be synchronized to within one network RTT, the virtual clocks will match exactly.

3.1.3 Central Replay

Our third major design decision was to enable off-site replay. Rather than restart each process *in situ*, a central console automatically downloads the necessary logs and checkpoints and instantiates each replay process locally. Local replay removes the network delay from the control loop, making it feasible to operate on distributed state and to step across processes to follow messages.

The costs are several: first, the network bandwidth consumed by transferring logs may exceed that required to control a remote debugger. Second, the hardware and system software on the replay machine must match the original host; currently we support only GNU/Linux/x86 hosts. Third, we must log data read from the local file system (as with network messages) because the files may not be available on the replay machine. This technique also obviates maintaining a versioned file system or undoing file modifications. Finally, building a migratable checkpoint system is challenging. We consider the first two costs to be acceptable and will discuss our solution to the last challenge in Section 3.2.6.

3.2 Challenges and Solutions

In this section we will discuss the technical challenges we faced when building our logging and replay system. Most are new problems caused by our user-level implementation and/or message annotations; previous projects did not address them because their focus allowed for different design choices.

3.2.1 Signals and Thread Replay in Userland

As we noted earlier, logging and replaying applications at the `libc` level assumes that they only interact with their environment through that interface and that, outside of `libc` calls, the application execution is deterministic. This assumption fails when multiple threads execute concurrently on the same address space. The value read from a shared variable depends on the order in which competing threads modify it; every write could be a race condition. The same problem arises when multiple processes share memory segments or when signal handlers (effectively another thread) access global variables.

To make replay deterministic in these cases, we must either intercept and replay the *value* of each read from shared memory, or we must replay each read and write in the same *order*, so races resolve identically. The former option is too invasive and requires log bandwidth proportional to the memory access stream. The latter is still expensive, but the cost can be reduced significantly by logging only the order and timing of thread context switches. If we assume a single processor, or artificially serialize thread operation, then identical thread schedules produce identical memory access patterns.

The challenge in our case was to record and replay thread schedules using only our user-level shared library. The task is relatively simple for kernel- or VM-based tools, but user-level libraries generally have no ability even to observe context switches among kernel threads, much less control them. We believe that `Liblog` is the first to address the problem.

Our solution effectively imposes a user-level cooperative scheduler on top of the OS

scheduler. We use a `pthread` mutex to block all but one thread at a time, ignoring conflicting context switches by the kernel. The active thread only surrenders the lock at `libc` call points, as part of our logging wrapper, and the next active thread logs the context switch before continuing. Processes that share memory are handled identically. Similarly, signals are queued and delivered at the next `libc` call.

Restricting context switches to our wrapper functions provides a convenient point to repeat the switches during replay, but the change to thread semantics is not fully transparent. In particular, we cannot support applications that intentionally use tight infinite loops, perhaps as part of a home-grown spin lock, because other threads will not have any opportunity to acquire our scheduling lock. Delaying signals may affect applications more, although we note that the kernel already tries to perform context switches and to deliver signals at syscall boundaries. We believe that the impact of our solution is negligible in most cases, but we have not quantified the degree to which the schedule that we impose differs from a normal one.

3.2.2 Unsafe Memory Access

Another potential source of non-determinism arises when an application reads from uninitialized (but allocated) heap memory or beyond the end of the stack. The contents of these memory regions are not well defined for C applications, and in practice they change between execution and replay. One could argue that accessing these regions could be considered incorrect behaviour, but it is legal, reasonably safe, and present even in robust software like OpenSSL [SSL].

Much of the change in memory between logging and replay is due to the logging tool itself, which calls different functions during replay, leaving different stack frames and allocating different memory on the heap. One can significantly minimize the tool's memory footprint, as stressed in Jockey [Sai05], but it can never be completely eliminated by a library-based debugging tool. Internal memory use by `libc` will always differ because its calls are elided during replay, so `malloc` may return different memory to the application.

Our solution is simpler: we merely zero-fill all memory returned by `malloc` (effec-

tively replacing it with `calloc`) as well as stack frames used by our `libc` wrappers. Thus, uninitialized reads replay deterministically, even if `malloc` returns a different region. This solution still fails if the application depends on the actual address, for example, as a key for a hash table.

Also, it is very difficult to protect a library-based tool from corruption by stray memory writes into the tool's heap. A virtual machine-based alternative would avoid this problem. Also, one could imagine disabling write access to the `Liblog`'s memory each time control returns to the application. Instead, we rely on dedicated memory-profiling tools like Purify [Pur] and Valgind [Val] to catch these various memory errors, so that we can focus on efficient logging.

3.2.3 Consistent Replay for TCP

As described in Section 3.1.2, we annotate all network messages between application processes with Lamport clocks so that we can replay communicating peers consistently. For datagram protocols like UDP, we use simple encapsulation: we prepend a few bytes to each packet, and remove them on reception. We pass a scatter/gather array to `sendmsg` to avoid extra copies.

Annotating byte streams like TCP is more complicated, because timestamps must be added throughout the stream, at the boundary of each sent data chunk. But the receiver need not consume bytes in the same batches; it often will read all available data, be it more or less than the contents of a single `send` payload.

Our solution is a small (3-state) state machine for each incoming TCP connections (see Figure 3.2). Once the stream has been verified as containing annotations, the state machine alternates reading annotations and reading application data until the calling function has enough data or the socket is drained. Each state transition requires a separate call to read the underlying stream; we cannot simply read extra bytes and extract the annotations, because we cannot anticipate how far to read. We do not know the frequency of future annotations, and attempting to read more data than necessary may cause the application to block needlessly. It is always possible that more bytes will not arrive.

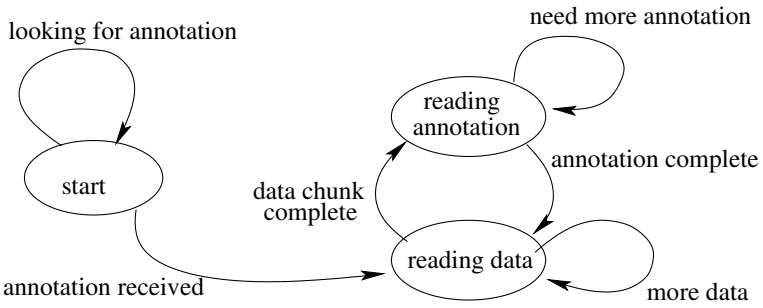

Figure 3.2: *Receiving Annotated TCP:* Detecting and extracting Lamport clocks from incoming byte streams requires additional bookkeeping.

If multiple annotations are consumed by a single `read` call, we log the most recent timestamp, as it supersedes the others. Naturally, we remember the stream state between calls so that we may continue even if the last read attempt ended in the middle of an annotation.

3.2.4 Finding Peers in a Mixed Environment

Embedding annotations in messages also complicates interaction with non-logging processes such as third-party clients, DNS and database servers, or, if `Liblog` is only partially deployed, even fellow application processes. These non-loggers do not expect the annotations and would reject or (worse yet) misinterpret the message. We believe that this problem is the reason that no previous logging tool has supported consistent replay in a mixed environment.

We must either send annotations that will be safely ignored by non-logging processes or discover whether a remote peer expects annotations and omit them when appropriate. The former option could be implemented using either IP options[1] or the out-of-band (OOB) channel for TCP connections, but either method would conflict with networks that already used these paths. Also, we have seen evidence that adding IP options has a negative impact on application traffic, and OOB does not help UDP traffic (nor incompatible TCP implementations).

We opted for a safer, but slower, solution. The `logger` on each machine tracks

[1]See RFC 791

the local ports opened by logging processes and listens on a globally well-known port (currently 5485). This approach fails to fully support applications hidden behind NAT-enabled firewalls, but it could easily replaced by a more sophisticated discovery mechanism. Each `Liblog`-enabled process then queries the remote `logger` (via TCP) before sending the first datagram or opening a TCP connection. The query contains the destination port and protocol of interest and asks whether that port is currently assigned to a logging process.

If the application receives a negative reply, or none at all, that packet flow will not be annotated. Replies are cached for the duration of a stream, or 30 seconds for datagram sockets, to amortize the query latency overhead. Currently, we wait a maximum of 2 seconds for a query, but that maximum is only reached if the remote machine has no `logger` and does not reset our TCP request. But this case does happen frequently for firewall-protected machines, so we cache information on dropped queries for up to 5 minutes.

3.2.5 Replaying Multiple Processes

The real power of replay debugging depends on the ability to set breakpoints, to pause execution, and to observe internal application state, just as one can in normal symbolic debuggers. Rather than develop new technology with its own interface, we decided to adapt the GNU Project Debugger [GDB]. GDB provides a powerful and familiar interface for controlling application execution and accessing internal state by symbolic names.

Unfortunately, GDB, like many debuggers, can only control a single process. Replaying multiple processes, or even children created with `fork`, requires multiple instances of the debugger. Our challenge was to coordinate them, multiplexing their input and output to the programmer and scheduling the application execution so that replay is consistent.

We use a two-tiered approach to controlling the replay processes. Threads within a process group are multiplexed by the same scheduling locks used during logging (cf. Section 3.2.1), always choosing the next thread based on the schedule stored in

the log. These locks also block a newly `fork`-ed process until we attach a new GDB instance to it.

Across process groups, consistent replay is enforced by our replay console, a small Python [Py] application. For each application process, the console uses GDB to set breakpoints in key `libreplay` functions. These pause execution at each `libc` call, allowing us to schedule the next process or to download the next set of logs.

The replay console provides a single interface to the programmer, passing commands through to GDB and adding syntax for broadcasting commands to multiple processes. It also allows advanced programmability by interacting directly with the underlying Python interpreter.

3.2.6 Migratable Checkpoints

Replaying application processes centrally, offline, makes the debugger more responsive and makes it feasible to operate on distributed application state. But restarting processes on a new machine is tricky. The two main challenges are first, to copy the state of the original application into a live process on the new machine, and second, to reconcile this new process with the debugger (GDB).

Our checkpoint mechanism is based on the `ckpt` [Ckp] library from the University of Wisconsin. This library reads the `/proc/` filesystem to build a list of allocated memory regions for the application and then writes all such memory to a checkpoint file. For replay, a small bootstrap application reads that file and overwrites its own memory contents, adjusting memory allocations as necessary.

First we extended `ckpt` to handle the kernel-level thread state for multi-threaded applications, which was simplified by our user-level scheduler. A thread saves its state before relinquishing the CPU, so at any time we have the state of all inactive threads stored in our tables.

Next we added support for shared memory regions: each process in a group checkpoints its private memory, and one "master" process writes and restores the shared memory for everyone.

Integrating checkpoint support to GDB required additional work. Starting the

process within GDB is problematic because the symbol tables of the bootstrap program and the restored application do not generally agree, or even necessarily overlap, and GDB does not support symbol tables moving during runtime. Even if we use the original application to bootstrap the process, GDB becomes confused when shared libraries are restored at new locations.

To solve this problem, we added a new method for finding the in-memory symbol table of a running application (by reading the `r_debug.r_brk` field), ignoring the conflicting information from the local executable file. It is then sufficient to attach to the restored application and to invoke this new symbol discovery method.

Our modifications required adding approximately 50 lines of code, including comments, to one source file in GDB. Most of those lines comprise the new function for locating the symbol table.

3.3 Limitations

There are several limitations to our debugging tool, both fundamental and mundane.

Log storage The biggest reason for a developer to *not* use `Liblog` with an application is the large amount of log data that must be written to local disk. Log storage is a fundamental problem for any deterministic replay system, but our approach to handling I/O (cf. Section 3.1) renders `Liblog` infeasible for high-throughput applications. Every Megabyte read from the network or disk must be logged (compressed) to the local disk, consuming space and disk bandwidth. This approach is acceptable for relatively lightweight applications like routing overlays, consuming only a few megabytes per hour, but is probably unrealistic for streaming video or database applications. We will quantify the problem in Section 3.4.

Host requirements Our basic logging strategy only addresses POSIX applications and operating systems that support run-time library interposition. In practice, our OS options are restricted even further, to recent Linux/x86 kernels (2.6.10+) and GNU

system software (only `libc` 2.3.5 has been tested). These limitations are imposed by our borrowed checkpointing code and compatibility issues with our modified version of GDB.

Scheduling semantics As explained in Section 3.2.1, `Liblog`'s user-level scheduler only permits signal delivery and context switches at `libc` function calls. The OS generally tries to do the same, so most applications will not notice a significant difference.

However, we are assuming that applications make these calls fairly regularly. If one thread enters a long computation period, or a home-grown spin lock implemented with an infinite loop, `Liblog` will never force that thread to surrender the lock, and signals will never be delivered. We have designed a solution to this problem, but implementation remains future work.

Network overhead Our network annotations consume approximately 16 bytes per message, which may be significant for some applications. The first 4 bytes constitute a "magic number" that helps us detect incoming annotations, but this technique is not perfect. Thus another limitation is that streams or datagrams that randomly begin with the same sequence of 4 bytes may be incorrectly classified by `Liblog` and have several bytes removed. This probability is low (1 in 2^{32} for random messages), and is further mitigated by additional validity checks and information remembered from previous messages in a flow, but false positives are still possible.

Limited consistency Fundamentally, consistent replay in a mixed environment is not guaranteed to be perfectly consistent. A message flow between two application processes loses its timing information if the flow is relayed by a non-logging third party. Then, if the virtual clocks for the two processes are sufficiently skewed, it is possible to replay message transmission *after* its reception. The probability of this scenario decreases rapidly as the application's internal traffic patterns increase in density, which keeps the virtual clocks loosely synchronized.

Completeness Finally, as mentioned earlier, library-based tools are neither complete nor sound, in the logical sense of the words. They are incomplete because they cannot reproduce every possible source of non-determinism. `Liblog` addresses non-determinism from system calls, from thread interaction, and, to a lesser extent, from unsafe memory accesses. Jockey [Sai05] focuses on a different set of sources, reducing changes to the heap and adding binary instrumentation for intercepting non-deterministic x86 instructions like `rdtsc` and, potentially, `int`.

Unfortunately, logging libraries will never succeed in making the replay environment exactly identical to the original environment because they operate inside the application's address space. The libraries run different code during logging and during replay, so their stack and heap differ. Theoretically, an unlucky or determined application could detect the difference and alter its behaviour.

Soundness We say logging libraries are *unsound* because, as part of the application, they may be corrupted. We hope that applications have been checked for memory bugs that could cause stray writes to `Liblog`'s internal memory, but C is inherently unsafe and mistakes may happen. We do assume the application is imperfect, after all.

Furthermore, libraries are susceptible to mistakes or crashes by the operating system, unlike hardware solutions or virtual machines (although even virtual machines generally rely on the correctness of a *host* OS).

Fortunately, these theoretical limitations have little practical impact. Most applications are simple enough for `Liblog` to capture all sources of non-determinism, and simple precautions to segregate internal state from the application's heap are usually sufficiently safe. Indeed, most debuggers (including GDB) are neither sound nor complete, but they are still considered *useful*.

3.4 Evaluation

We designed `Liblog` to be sufficiently lightweight so that developers would leave it permanently enabled on their applications. In this section, we attempt to quantify

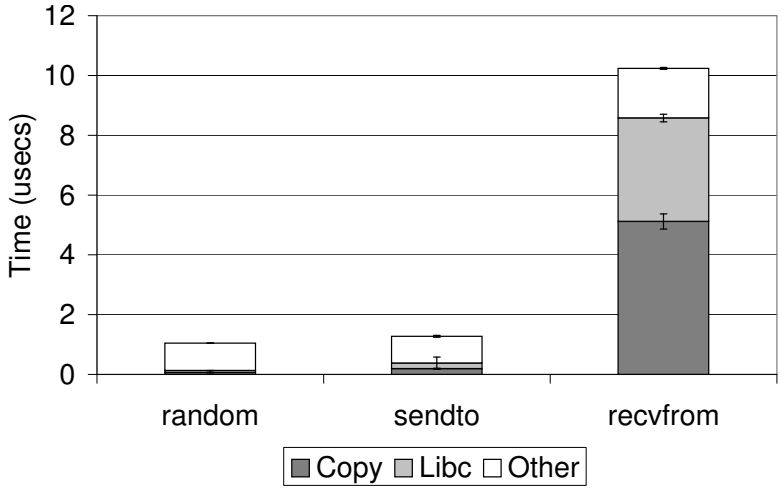

Figure 3.3: *Wrapper Overhead:* time required to intercept and log `libc` functions. The *copy* region measures the time taken to write the bytes to a shared memory region monitored by the logger, and *other* includes the overhead of intercepting the calls an our internal bookkeeping. The *libc* region measures the time taken for the underlying library call to complete.

the overhead imposed by `Liblog`, both to see whether we reached this goal and to help potential users estimate the impact they might see on their own applications.

We start by measuring the runtime latency added by our `libc` wrappers and its effect on network performance. We find that each system call requires a few microseconds, which reduces peak UDP send rate by %18 but has negligible effect on TCP throughput. A second set of experiments measures the storage overhead consumed by checkpoints and logs. This overhead depends greatly on the behaviour of the application being logged, and averages 3–6 MB/hour for our own programs.

All experiments were performed on a Dual 3.06GHz Pentium 4 Xeon (533Mhz FSB) with 512K L2 cache, 2GB of RAM, 80GB 7500 rpm ATA/100 disk, and Broadcom 1000TX gigabit Ethernet.

3.4.1 Wrapper Latency

To measure the processing overhead of `Liblog`, we first analyzed the latency added to each `libc` call. Figure 3.3 shows the latency for a few representative wrappers.

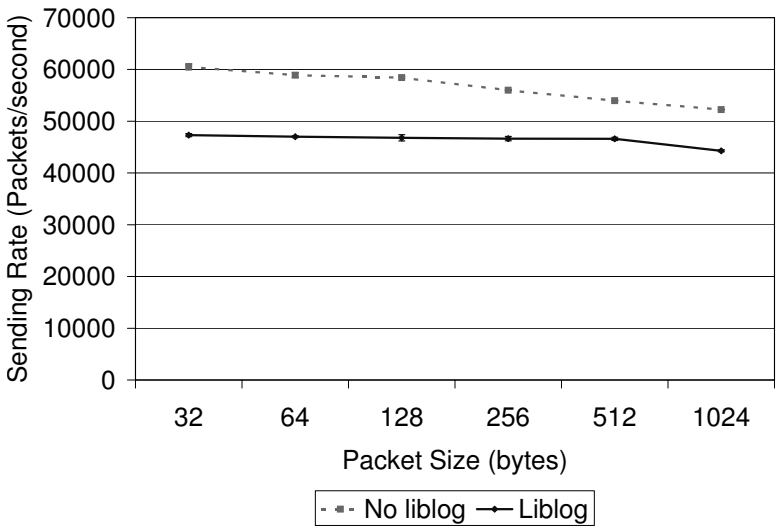

Figure 3.4: *Packet rate reduction:* Maximum UDP send rate for various datagram sizes. The maximum standard deviation over all points is 1.3 percent of the mean.

The wrappers add approximately 1 microsecond to the function `random`, which shows the minimum amount of work each wrapper must do to intercept the call and to write a log entry. The `sendto` wrapper is slightly slower as it includes the amortized cost of querying the destination to determine whether to send annotations (cf. Section 3.2.4). The "copy" phase is also longer, because we store the outgoing message address and port to facilitate message tracing. The `recvfrom` overhead is higher still because it must extract the Lamport clock annotation from the payload and copy the message data to the logs.

3.4.2 Network Performance

Next we measured the impact of `Liblog` on network performance. First we wrote a small test application that sends UDP datagrams as fast as possible. Figures 3.4 and 3.5 show the maximum packet rate and throughput for increasing datagram sizes. With `Liblog` enabled, each rate was reduced by approximately 18%.

For TCP throughput, we measured the time required for `wget` to download a 484 MB binary executable from various web servers. Figure 3.6 shows that `Liblog` hinders `wget` when downloading the file over a gigabit ethernet link, but the reduction

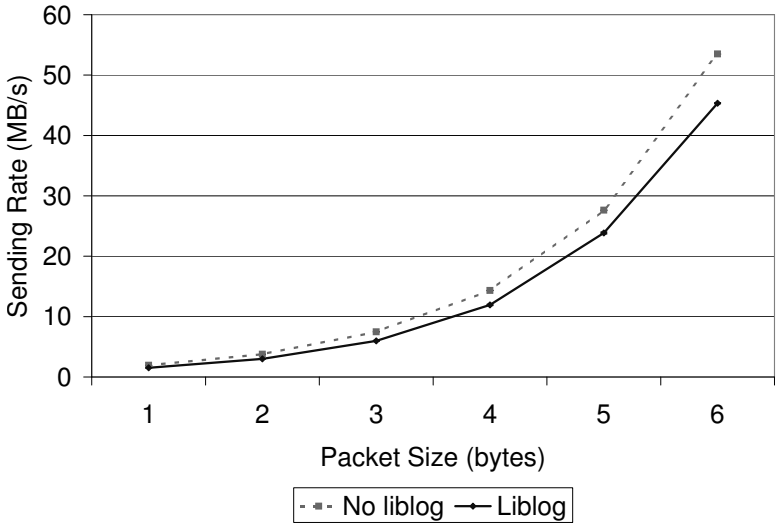

Figure 3.5: *UDP bandwidth:* Maximum UDP send throughput for various datagram sizes. The maximum standard deviation over all points is 1.3 percent of the mean.

in throughput is negligible when the maximum available throughput is lowered. Even the relatively fast 100 Mbps link to our departmental web server can be filled using `Liblog`.

Finally, Figure 3.7 shows the round-trip time (RTT) measured by `lmbench` to the local host and to a machine on a nearby network. The gigabit ethernet test shows that `Liblog` adds a few wrappers worth of latency to each RTT, as expected. On a LAN, the RTT overhead is so small that the difference is hard to discern from the graph.

3.4.3 Log Bandwidth

The amount of log space required depends greatly on the frequency of `libc` calls made by an application, as well as on the throughput and content of its network traffic, because incoming message contents are saved.

To give an idea of the storage rates one might expect, we first measured the average log growth rate of the applications we use ourselves: I3/Chord [SAZ+02b] and the OCALA proxy [JKK+06]. For this experiment, we started a small I3 network on PlanetLab [PL] and attached a single local proxy. No additional workload was applied,

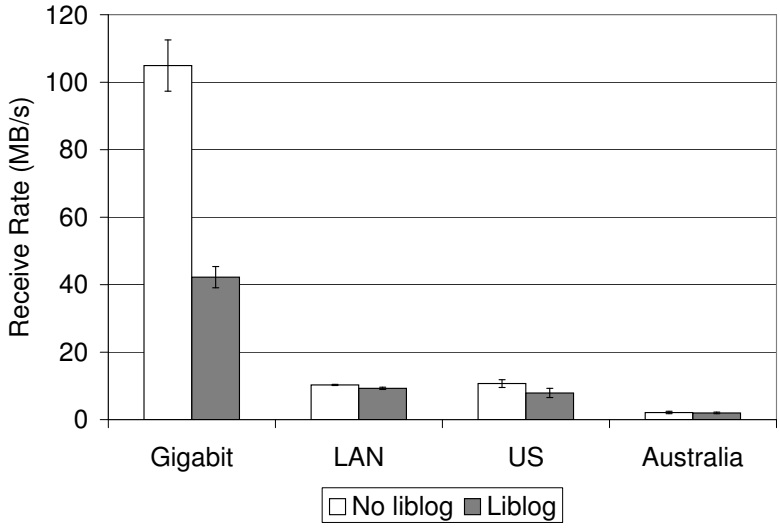

Figure 3.6: *TCP throughput* for `wget` downloading a 484MB file. Each pair of bars represents a different web server location.

so the processes were only sending their basic background traffic. We also show the logging rates for `wget` downloading an executable file when we artificially limit its download rate to simulate applications with various network throughput. Figure 3.8 shows the (compressed) log space required per hour for each application. This rate varies widely across applications and correlates directly with network throughput. We have found the 3–6 MB/hour produced by our own applications to be quite manageable.

Figure 3.9 illustrates the degree to which message contents affect the total log size. We limited `wget` to a 1 KB/s download rate and downloaded files of various entropy. The first file was zero-filled to maximize compressibility. Then we chose two real files: File A is a binary executable and File B is a `Liblog` checkpoint. Finally, we try a file filled with random numbers, which, presumably, is incompressible. The difference between zero and full entropy is over an order of magnitude, although most payloads are presumably somewhere in the middle.

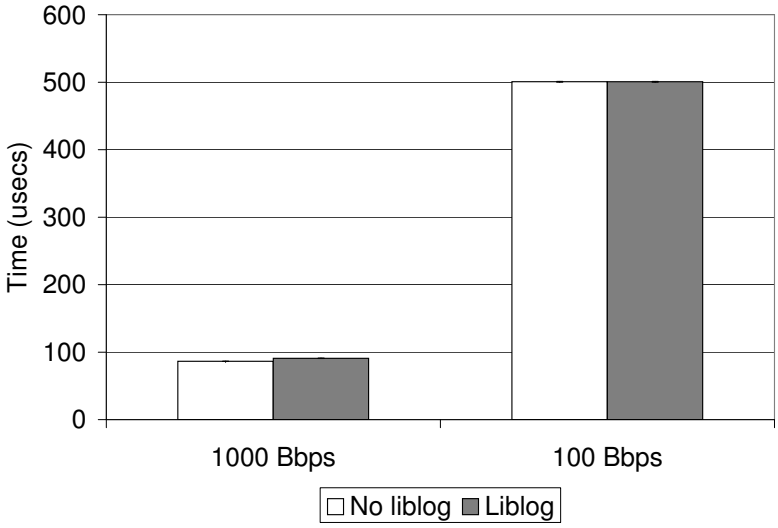

Figure 3.7: *RTT overhead:* measured by `lmbench`. The error bars cannot be seen in these graphs because the standard deviation is negligible.

3.4.4 Checkpoint Overhead

Finally, we measured the checkpoint latency (Figure 3.10) and size (Figure 3.11) for a few of our test applications. The checkpoint size depends on the amount of the application's address space that is in use. The checkpoint latency is dominated by the time required to copy the address space to file system buffers, which is directly proportional to the (uncompressed) checkpoint size. These costs can be amortized over time by tuning the checkpoint frequency. The trade-off for checkpoint efficiency is slower replay, because more execution must be replayed on average before reaching the point of interest.

3.4.5 Evaluation Summary

These experiments suggest that the CPU overhead imposed by `Liblog` is sufficiently small for many environments and has little affect on network performance. Logging could consume considerable disk space (and disk bandwidth), but the distributed applications we are familiar with (I3/Chord and OCALA) could store logs for a week or two, given 1GB of storage. Checkpoints also consume a noticeable

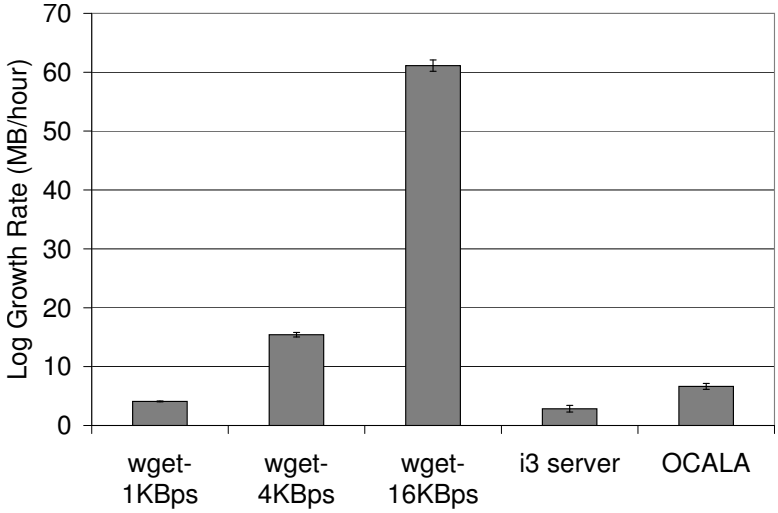

Figure 3.8: *Log bandwidth:* Log size written per hour for various applications. The bottom three columns correspond to `wget` with the specified cap on its download rate.

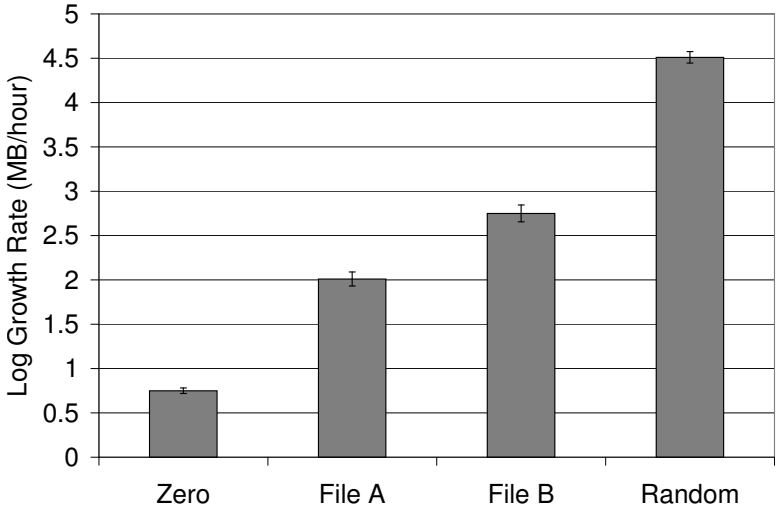

Figure 3.9: *Log entropy:* Log size written by `wget` depends on compressibility of incoming data.

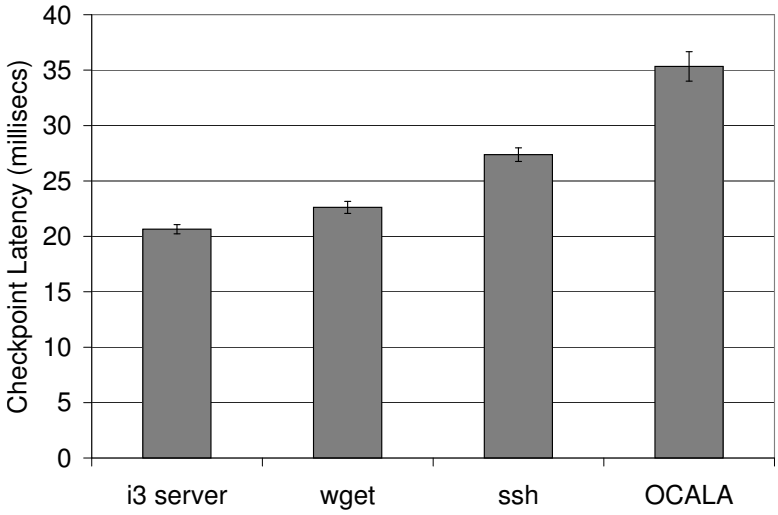

Figure 3.10: *Checkpoint Latency:* time taken to dump memory to checkpoint file for various applications.

amount of space, but writing one once an hour is probably sufficient for most cases.

3.5 Experience

We have been developing `Liblog` for over two years, and although the library itself has undergone constant change, the basic form of the prototype described in this dissertation has remained somewhat stable for the past year. We have used the tool on distributed applications with which we are familiar, namely I3/Chord [SAZ+02b] and the OCALA proxy [JKK+06]. We have already discovered several errors in these applications. In this section, we will describe how `Liblog` helped in these cases, along with a few stories from earlier prototypes and work debugging `Liblog` itself.

3.5.1 Programming Errors

To start, we found a few simple mistakes that had escaped detection for months. The first, inserted accidentally by one of the author over a year ago, involved checking Chord timeouts by calling `gettimeofday` within a "MAX" macro that evaluated its arguments twice. The time changed between calls, so the value returned was not

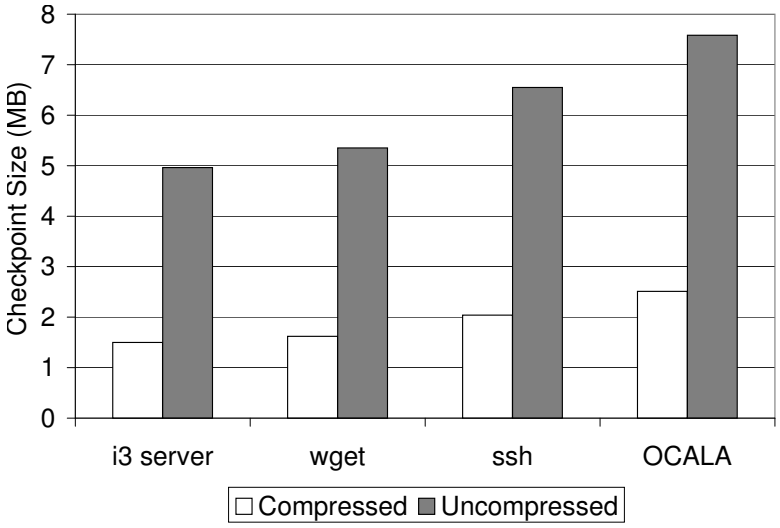

Figure 3.11: *Checkpoint Size:* total and compressed size of checkpoints for various applications.

always still the maximum.

We also found an off-by-one error in code that assumed 1-based arrays and timer initialization code that did not add `struct timeval` microseconds properly, both in OCALA's I3 library.

The off-by-one error normally had no visible effect but occasionally caused the proxy to choose a distant, high-latency gateway. The two timer-related errors only manifested occasionally but would cause internal events to trigger too late, or too early, respectively.

These bugs had escaped earlier testing because they were non-deterministic and relatively infrequent. But once we noticed the problems, `Liblog` was able to deterministically replay the exact execution paths so that we could step through the offending code in GDB and watch the problem unfold.

3.5.2 Broken Environmental Assumptions

Perhaps more interesting are bugs caused not by programmer mistakes but rather by correct implementation based on faulty assumptions. To illustrate, here are two problems in Chord we had found with an earlier `Liblog` prototype.

The first problem is common in peer-to-peer systems, and was discussed along with solutions in a later paper [FLRS05]. Basically, many network overlays like Chord assume that the underlying IP network is fully connected, modulo transient link failures. In practice, some machine pairs remain permanently disconnected due to routing policy restrictions and some links experience unexpected partial failure modes, such as transient asymmetry. Both problems cause routing inconsistencies in Chord, and both were witnessed by `Liblog` in a network deployed across PlanetLab [PL].

Rather than finding a coding error in the application, replay showed us code that worked as designed. Our project is focused on application debugging, and we do not attempt to debug the underlying network; nevertheless, our logs clearly showed the unexpected message-loss patterns. Of course the problem had not been detected using simulation, because the simulator made the same assumptions about the network as the application.

A second assumption we had made was that our application processes would respond to keep-alive messages promptly. Chord includes RTT estimation and timeout code based on TCP, which expects a reasonable amount variance. On PlanetLab, however, high CPU load occasionally causes processes to freeze for several seconds, long enough for several successive pings to time out. Chord then incorrectly declared peers offline and potentially misrouted messages.

Upon inspection, `Liblog` showed us that the timeout code was operating correctly, and the message tracing facilities detected the keep-alive responses arriving at the correct machines, although long after they had been considered lost. The virtual clock timestamps let us correlate otherwise-identical messages, as well as detect the long delay in between system calls on the pinged machine.

3.5.3 Broken Usage Assumptions

We found two problems with the OCALA proxy's overlay client initialization code, both caused by sensitivity to the bootstrap gateway list. Like those of the previous section, these "bugs" were not programming errors per se, but rather user errors (providing an imperfect configuration list) or design flaws (not tolerating user error).

One phase of startup involves pinging these gateways and triangulating the local machine's latitude and longitude based on the response times. We noticed that the proxy occasionally made a very poor estimate of local coordinates, which then caused a poor (high latency) choice of primary gateways.

We investigated the phenomenon by setting breakpoints in the relevant methods and stepping through the replay. We noticed first that very few points were used for triangulation. We then moved *backwards* in the execution to find that only a small number of pings were sent and that the proxy did not wait long enough for most the replies. If care is taken to nominate only lightly loaded gateways, triangulation works fine. If not, as in our case, performance suffers until periodic maintenance routines manage to choose a better gateway, which could take hours.

We also discovered that the proxy client is very trusting of liveness information contained in the initial gateway list. Normally this list is continually updated by an independent process so that only active gateways are included. If the list becomes stale, as we unintentionally allowed, the proxy could waste minutes trying to contact dead I3 servers before finally connecting.

We diagnosed the problem by replaying and comparing the paths taken by two executions: one which exhibited the interminable timeouts and one which lucked upon a good subset of gateways immediately. This problem could easily be dismissed as invalid usage. Nevertheless, solving it relied on our ability to deterministically replay the random choices made during the gateway selection process.

3.5.4 Self-Debugging

The program we have spent the most time debugging recently is `Liblog` itself. Because the tools run as shared libraries in the application address space, we are able to use GDB to set breakpoints and to step through our own code during replay, just like the supposed target application. We used this ability to fix programming errors in our message annotation layer and our remote discovery service. Deterministic replay also made it easy to find faults in our replay console because each log provided a repeatable test case.

Some bugs in `Liblog`, such as incomplete `libc` wrappers, manifest as non-determinism during replay. Ironically, this non-determinism made them easy to detect because we could step through the execution at the point where the original execution and replay diverged in order to isolate the failure. This approach also led us to realize the problem of applications accessing undefined heap and stack memory.

3.5.5 Injected Bugs

Our tool is interactive, aiding a human programmer but requiring their domain knowledge and expertise. We find it difficult to quantify the benefit `Liblog` provides because the user injects a large amount of variability into the process. Ideally, we will be able to compile a large library of "real" bugs that exist in tested and used applications for some time before being fixed with `Liblog`. But this process is slow and unpredictable.

Projects that develop automated analytic techniques often pull known errors from bug databases and CVS histories in order to quantify how many of the problems can be re-fixed with their tools. This path is also available to use, but the results would be somewhat suspect as the human tester may have some prior knowledge of old bugs. Similar doubts may arise if one set of programmers manually introduces errors into a current application code base for testing by an independent second group. This trick has the benefit of testing our tools on bugs that are arbitrarily complex or slow to develop.

While we wait for our library of real bugs to grow, we have decided to try both of these somewhat-artificial testing methods. So far we have only started on the latter, with one author injecting an error into the I3/Chord code base while the other uses `Liblog` to isolate and fix it. Our experience suggests that the task is equivalent to debugging Chord in a local simulator.

3.6 Summary

We have designed and built `Liblog`, a new logging and replay tool for deployed, distributed applications. We have plans for a few additional improvements to `Liblog`, both to reduce its runtime overhead and to remove some of the limitations listed in Section 3.3. Meanwhile, we hope to receive feedback from the community that will help us improve its usability.

Our ongoing research plan views `Liblog` as a platform for building further analysis and failure detection tools. Specifically, replaying multiple processes together provides a convenient arena for analyzing distributed state. We see great potential for consistency checking and distributed predicate evaluation tools. In the next chapter we will present one such tool, `Friday`.

Chapter 4

Friday

While `Liblog` provides the programmer with the basic information and tools for debugging distributed applications, the process of tracking down the root cause of a particular problem remains a daunting task. The information presented by `Liblog` can overwhelm the programmer, who is put, more often than not, in the position of finding a "needle in the haystack." `Friday` enables the programmer to prune the problem search space by expressing complex global conditions on the state of the whole distributed application.

In this chapter, we present in detail the two key facilities provided by `Friday`: (1) distributed watchpoints and breakpoints that operate on the *global* state of the application, and (2) commands that allow one to associate arbitrary code with breakpoints and watchpoints, that operate on the application's global state.

4.1 Design

`Friday` presents to users a central debugging console, which is connected to replayed node processes, each of which runs an instance of a traditional symbolic debugger such as GDB (see Figure 4.1). The console includes an embedded script language interpreter, which interprets actions and can maintain central state for the debugging session. Most user input is passed directly to the underlying debugger processes, allowing full access to the debugger's data analysis and control functions. `Friday` ex-

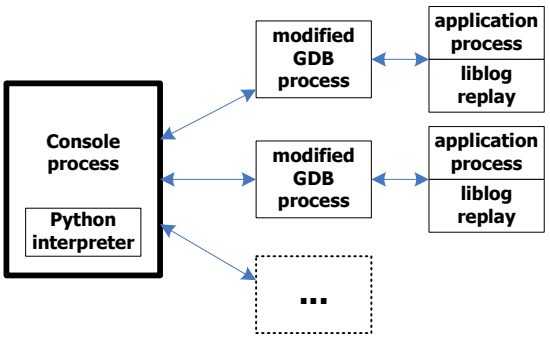

Figure 4.1: Overall architecture of `Friday`

tends the debugger's commands to handle distributed breakpoints and watchpoints, and to show status information about the whole system of debugged processes.

4.1.1 Distributed Watchpoints and Breakpoints

Traditional watchpoints allow a symbolic debugger to react—stop execution, display values, or evaluate a predicate on the running state—when the debugged process updates a particular variable location. Though watchpoints are defined in terms of writes to a specific set of memory addresses, debuggers allow these addresses to be specified via the symbolic names defined by the application's source code.

In addition to this traditional functionality, `Friday`'s distributed watchpoints can specify variables and expressions that belong to multiple nodes in the replayed distributed application. For example, a programmer debugging a ring network can use `Friday` to watch a variable called `successor` on all machines by specifying "`watch successor`" or for a single machine (here, #4) from the replay group "`4 watch successor`".

A command of the form "`<node number>, ... watch <variable>, ...`" specifies both a set of nodes on which to watch variables, and a set of variables to watch. When no list of nodes is indicated, a watch expression refers to all nodes. The node numbering used is private to `Friday`; to identify a particular node by its application-specific identifier such as an IP address or an overlay ID, an appropriate mapping watchpoint can be provided—see Section 4.1.2 for an example.

Distributed breakpoints in `Friday` have a similar flavor. Like traditional breakpoints, they allow the debugger to react when the debugged process executes a particular instruction, specified symbolically as a source line number or a function name. `Friday` allows the installation of such breakpoints on one, several, or all replayed nodes.

Implementation

`Friday` implements distributed watchpoints and breakpoints by setting local instances on each replay process and mapping the individual watch- or breakpoint numbers and addresses to a global identifier for easier operation from the replay console. These map tables are used to re-write and forward requests to disable or re-enable a distributed watch- or breakpoint, and also to map local events back to the global index in order to notify the user and find any attached commands to execute.

Local breakpoints simply use GDB breakpoints, which internally either use debugging registers on the processor or inject trap instructions into the code text. In contrast, `Friday` implements its own mechanism for local watchpoints. `Friday` uses the familiar technique of write-protecting the memory page where the value corresponding to a given symbol is stored [Wah92]. When a memory write to the page containing the watched variable occurs, the ensuing `SEGV` signal is captured by `Friday`, which unprotects the page and completes the write before passing control to any state manipulation scripts attached to the watchpoint.

This implementation can of course give rise to *false positives*, that is, trapping into `Friday` for unwatched data changes, since writes to any variable sharing a page with a watchpoint will cause a trap. The more densely populated a memory page, the more such false positives occur. Furthermore, if the watched variable shares a page with an unwatched, but frequently updated, value, the overhead can become significant. Nevertheless, we decided that protection-based watchpoints are preferable to alternative implementations, as explained next.

Why a New Watchpoint Mechanism?

We explored but rejected four other alternatives to implement watchpoints: hardware watchpoints, single stepping, implementation via breakpoints, and time-based sampling.

Hardware watchpoints are offered by many processor architectures. They are extremely efficient, causing essentially no runtime overhead, but most common processors have small, hard limits on the number of hardware watchpoint registers (a typical value is 8, and these are shared with breakpoints), as well as on the width of the watched variable (typically, a single machine word). These limits are too restrictive for the flexible predicates that we wanted to support; however, we have planned a hybrid system that uses the fast hardware watchpoints as a cache for our more flexible mechanism.

Single-stepping, or *software watchpoints*, can implement watchpoints by executing one machine instruction at a time and checking for variable modifications at each step. Unfortunately, single-stepping is prohibitively slow—we compare it to our method in Section 4.3.4 and demonstrate that it is a few thousand times slower.

Local breakpoints can emulate watchpoints by identifying the points where the watched variable could be modified and only checking for changes there. When this identification step is accurate the technique is highly efficient, but unfortunately it requires comprehensive knowledge of the program code. It is more work for the programmer, and prone to *false negatives*, that is, missed data changes for watched variables.

Periodic sampling of watched variable values (e.g., every k logical time ticks) to check for modifications enables a trade-off between replay speedup and watchpoint accuracy: it is potentially faster than all the techniques described above, but it may be difficult to identify precisely when the value was changed. Combined with replay checkpointing and backtracking, it might prove a valuable but not complete alternative.

Implementation Complexity

Building a new watchpoint mechanism in `Friday` required reconstructing some functionality normally provided by the underlying symbolic debugger, GDB. Namely, debuggers maintain state for each watched expression, including the stack frame where the variable is located (for local variables) and any mutable subexpressions whose modification might affect the expression's value. For example, a watchpoint on `srv->successor->addr` should trigger if the pointers `srv` or `srv->successor` change, pointing the expression to a new value. Because GDB does not expose this functionality cleanly, we replicated it in `Friday`.

Also, the new watchpoint mechanism conflicts with GDB's stack maintenance algorithms. When `Friday` removes write permissions from a page of memory on the stack, which is later modified, `Friday` will catch the segmentation fault and attempt to restore permissions, as described in Section 4.1.1. This operation should succeed, because GDB creates a new stack for calling functions in the target application's address space. Unfortunately, GDB performs a small amount of initialization that touches the main application stack, which is still unwritable, so the call to restore permissions (via `mprotect`) fails. We have solved this problem by avoiding GDB's normal calling method and creating our own, manipulating the application's `PC` directly. However, this solution conflicts with GDB's breakpoint maintenance routines if the application is stopped at a breakpoint when we modify the application `PC`. We are working on alleviating this adverse interaction between `Friday` and GDB, but we have not encountered the problem in our use of the system, including our case studies presented later in this dissertation.

4.1.2 Commands

The second crucial feature of `Friday` is the ability to view and manipulate the distributed state of replayed nodes. These actions can either be performed interactively or triggered automatically by watchpoints or breakpoints. Interactive commands such as `backtrace` and `set` are simply passed directly to the named set of debugger processes. They are useful for exploring the distributed state of a paused system.

In contrast, automated commands are written in a scripting language for greater expressiveness. These commands are typically used to maintain additional views of the running system to facilitate statistics gathering or to reveal complex distributed (mis)behaviours.

`Friday` commands can maintain their own arbitrary debugging state, in order to gather statistics or build models of global application state. In the examples below, `emptySuccessors` and `nodesByID` are debugging state, declared in `Friday` via the `python` statement; e.g., `python emptySuccessors = 0`. This state is shared among commands and is persistent across command executions.

`Friday` commands can also read and write variables in the state of any replayed process, referring to symbolic names exposed by the local GDB instances. To simplify this access, `Friday` embeds into the scripting language appropriate syntax for calling functions and referencing variables from replayed processes. For example, the statement "`@4(srv.successor) == @6(srv.predecessor)`" compares the successor variable on node 4 to the predecessor variable on node 6. By omitting the node specifier, the programmer refers to the state on the node where a particular watchpoint or breakpoint was triggered. For example, the following command associated with a watchpoint on `srv.successor` increments the debugging variable `emptySuccessors` whenever a successor pointer is set to `null`, and continues execution:

```
if not @(srv.successor):
   emptySuccessors++
cont
```

For convenience, the node where a watchpoint or breakpoint was triggered is also accessible within command scripts via the `__NODE__` metavariable, and all nodes are available in the list `__ALL__`. For example, the following command, triggered when a node updates its application-specific identifier variable `srv.node.id`, maintains the global associative array `nodesByID`:

```
nodesByID[@(srv.node.id)] = __NODE__
cont
```

Furthermore, `Friday` provides commands with access to the logical time kept by the Lamport clock exported by `Liblog`, as well as the "real" time recorded at each log

event. Because `Liblog` builds a logical clock that is closely correlated with wall clock during trace acquisition, these two clocks are usually closely synchronized. `Friday` exposes the global logical clock as the `__LOGICALCLOCK__` metavariable and node i's real clock at the time of trace capture as `@i(__REALCLOCK__)`.

Similarly to GDB commands, our language allows setting and resetting distributed watchpoints and breakpoints from within a command script. Such *nested* watchpoints and breakpoints can be invaluable in selectively picking features of the execution to monitor in reaction to current state, for instance to watch a variable only in between two breakpoints in an execution. This can significantly reduce the impact of false positives. It can also enable powerful debugging usage patterns efficiently, such as observing whether the distributed execution of an application follows a parametric global state machine—for example, observing that after variable `@nodeA(neighbor)` is set then variable `@neighbor(X)` should be set. Nested watchpoints allow us to watch `@neighbor(X)` only after `@nodeA(neighbor)` has been set, reducing the overhead significantly.

Language Choice

The `Friday` commands triggered by watchpoints and breakpoints are written in Python, with extensions for interacting with application state, which we describe in the next section.

Evaluating Python inside `Friday` is straightforward, because the console is itself a Python application, and dynamic evaluation is well supported. We chose to develop `Friday` in Python for its high-level language features and ease of prototyping; these benefits also apply when writing watchpoint command scripts.

We could have used a compiled command language instead, as C is used in Intro-Virt [JKDC05]. Such an approach might provide better performance, and it allows the debugging predicates to share the application's namespace. Unfortunately this option requires recompiling a shared library and loading it into the application each time the user thinks of a new predicate; we wanted to support a more dynamic, interactive model.

We could have avoided the compilation step by leveraging GDB's "command list" functionality, which lets the user attach a series of normal GDB commands and simple conditional expressions to a watchpoint or breakpoint. Unfortunately these commands lack the high-level language expressiveness of Python, like the ability to construct new data structures. Furthermore, that would require execution of `Friday` commands on individual nodes' GDB instances, which would reprise the problem of local, partial knowledge of application state. Using a general-purpose language like Python running at the console was a more flexible choice.

Syntax

When a distributed command is entered, `Friday` examines every statement to identify references to the target application state. These references are specified with the syntax `@<node>(<symbol>[=<value>])` where the `node` defaults to that which triggered the breakpoint or watchpoint. These references are replaced with calls to internal functions that read from or write to the application using GDB commands `print` and `set`, respectively. Metavariables such as `__LOGICALCLOCK__` are interpolated similarly. Furthermore, `Friday` allows commands to refer to application objects on the heap whose symbolic names are not within scope, especially when stopped by a watchpoint outside the scope within which the watchpoint was defined. Such pointers to heap objects that are not always nameable can be passed to watchpoint handlers as parameters at the time of watchpoint definition, much like continuations (see Section 4.2.2 for a detailed example). The resulting statements are compiled, saved, and later executed within the global `Friday` namespace and persistent command local namespace.

If the value specified in an embedded assignment includes keyed printf placeholders, i.e., `%(<name>)<fmt>`, the value of the named Python variable will be interpolated at assignment time. For example, the command

```
tempX = @(x)
tempY = @other(y)
@(x=%(tempY)d)
@other(y=%(tempX)d)
```

swaps the values of integer variables `x` at the current node and `y` at the node whose number is held in the python variable `other`.

Commands may call application functions using similar syntax:

```
@<node>(<function>(<arg>,....))
```

These functions would fail if they attempted to write to a memory page protected by `Friday`'s watchpoint mechanism, so `Friday` conservatively disables all watchpoints for that replay process for the duration of the function call. Unfortunately that precaution may be very costly (see Section 4.3). If the user is confident that a function will not modify any protected memory, she may start the command with the `safe` keyword, which instructs `Friday` to leave all watchpoints enabled. This option is helpful, for example, if the invoked function only modifies the stack, and watchpoints are only set on global variables.

The value returned by GDB using the `@()` operator must be converted to a Python value for use by the command script. `Friday` understands strings (type `char*` or `char[]`), and coerces pointers and all integer types to Python `long` integers. Any other type, including any structs and class instances, are extracted as a tuple containing their raw bytes. This solution allows simple identity comparisons, which was sufficient for all useful case studies we have explored so far.

Finally, our extensions had to resolve some keyword conflicts between GDB and Python, such as `cont` and `break`. For example, within commands `continue` refers to the Python keyword whereas `cont` to GDB's keyword. In the general case, we can prefix the keyword `gdb` in front of GDB keywords within commands.

4.1.3 Limitations

We have found `Friday` to be a powerful and useful tool; however, it has several limitations that potential users should consider.

We start with limitations that are inherent to `Friday`. First, false positives can slow down application replay. False positive rates depend on application structure and dynamic behaviour, which vary widely. In particular, watching variables on the stack can slow `Friday` down significantly. In practice we have circumvented

this limitation by recompiling the application with directives that spread the stack across many independent pages of memory. Though this runs at odds with our goal of avoiding recompilation, it is only required once per application, as opposed to requiring recompilations every time a monitored predicate or metric must change. Section 4.3 has more details on `Friday` performance.

The second `Friday`-specific limitation involves replaying from a checkpoint, as opposed to from the beginning of a replay trace. Since some `Friday` predicates build up their debugging state by observing the dynamic execution of a replayed application, when starting from a checkpoint these predicates must rebuild that state through observation of a static snapshot of the application at that checkpoint. While such rebuilding of debugging state is straightforward for the applications we study in Section 4.2, it may be more involved for applications with less clean, complex data structures. We are currently working on a method for checkpointing and storing debugging state along with `Liblog` checkpoints at debug time, to simplify further the predicate complexity required of programmers for quick replays.

Thirdly, we have found that `Friday`'s centralized and type-safe programming model makes predicates considerably simpler than the distributed algorithms they verify. Nevertheless, most `Friday` predicates do require some debugging themselves. For example, Python's dynamic type system allowed us to refer to application variables that were not in dynamic scope, causing runtime errors. These issues can be addressed by using a statically-typed language like OCaml.

Beyond `Friday`'s inherent limitations, the system inherits certain limitations from the components on which it depends. First, an application may copy a watched variable and modify the copy instead of the original, which GDB is unable to track. This pattern is common, for example, in STL collection templates, and requires the user of GDB (and consequently `Friday`) to understand the program well enough to place watchpoints on all such copies. The problem is exacerbated by the difficulty of accessing these copies, mostly due to GDB's inability to place watchpoints on STL's many inlined accessor functions.

A second inherited limitation is unique to stack-based variables. As with most common debuggers, we have no solution for watching stack variables in functions

that have not yet been invoked. To illustrate, it is difficult to set up ahead of time a watchpoint on the command line argument variable `argv` of the `main` function across all nodes before we have entered the `main` at all nodes. Nested watchpoints are a useful tool in that regard.

Finally, `Friday` inherits from `Liblog` its non-trivial storage requirements for logs and an inability to log or replay threads in parallel on multi-processor or multi-core machines. The latter may be a feature disguised as a limitation, since most human programmers are not quite as proficient at debugging in parallel as computers are.

4.2 Case Studies

In this section, we present use cases for the new distributed debugging primitives presented above. First, we look into the problem of consistent routing in the i3/Chord DHT [SAZ+02a], which has occupied networking and distributed research literature extensively. Then we turn to debugging `Tk`, a reliable communication toolkit [Sub05], and demonstrate sanity checking of disjoint path computation over the distributed topology, an integral part of many secure-routing protocols. For brevity, most examples shown omit error handling, which typically adds a few more lines of Python script.

4.2.1 Routing Consistency

In this section, we describe `Friday` predicates to demonstrate debugging of routing inconsistencies in i3/Chord. In such a distributed lookup service, routing consistency is the property of answering the same lookup with the same result at the same time, regardless of who is asking. All examples refer to the `srv` data structure, which contains a node's `successor` and `predecessor` pointers in Chord's ring topology, and a node's application-specific identifier `srv.node.id` and IP address `srv.node.addr`.

We show examples that detect link reciprocity, extract consistency statistics, and detect routing state oscillation, a common misbehaviour of routing protocols that might result in route flaps, wormholes, or even black holes.

Node Indexing

An important invariant in many distributed systems is that naming assigns unique nodes to each distinct system-specific identifier. For example, nodes on the same subnet cannot have the same IP address, and nodes on a distributed hash table cannot have the same ID.

One simple use of our distributed comprehension infrastructure allows us to index nodes according to an application-specific node identifier, as introduced in Section 4.1.2. We extend the earlier utilitarian indexing example to check the uniqueness of names in the population as well:

```
break chord.c:58
python nodesByID = {}
command
  id = @(srv.node.id)
  if id in nodesByID :
    print __NODE__, "is stealing identifier", id
  else
    nodesByID[id] = __NODE__
    cont
end
```

These distributed commands are triggered by a breakpoint on a source code line, not shown here, where a node's application-specific ID is set. At that time, `Friday` will update its index with the current node number for the triggering node, and will stop execution if an ID collision is detected. This example harnesses application-specific knowledge that a node sets its `id` only once, so a breakpoint, rather than the more expensive watchpoint, can be used. To check even that assumption, one could also set a distributed watchpoint and command on `srv.node.id`, ensuring that it is a write-once variable.

This mechanism can be extended to index replayed machines according to any choice of one or more application-specific literals. For example, we could index replayed machines according to the multicast groups to which they participate or according to the IDs of the resources they hold, etc.

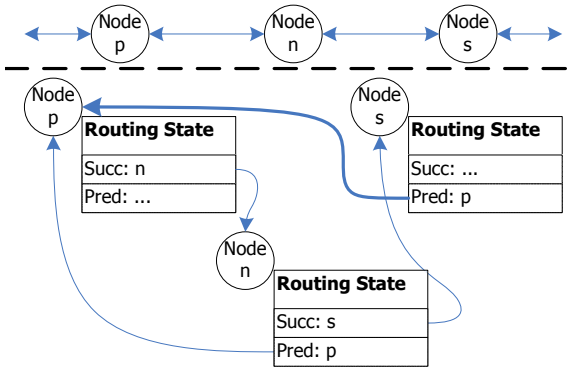

Figure 4.2: At the top, we show what node n hopes the ring topology looks like around it. At the bottom, we see the relevant state as stored by the involved nodes n, s and p. The thick routing entry from node s to its predecessor, which points to p instead of node n illustrates a possible inconsistency with node p and s's successor pointers.

Ring Consistency

Routing inconsistency can result when basic assumptions on the connectivity graph are violated. For example, in overlays that organize members in a bidirectional ring topology, the symmetry of ring links is such an assumption. The specific invariant in terms of individual nodes' states is that every node is its immediate ring successor's predecessor and its immediate ring predecessor's successor. Figure 4.2 provides an illustration.

Checking that successor/predecessor consistency conditions hold at all times is unnecessary. Instead, it is enough to check the conditions only when a successor or predecessor pointer changes, and only check those specific conditions in which the changed pointers participate. We can encode these two symmetric checks in `Friday` as follows:

```
watch srv.successor
command
  successor_id = @(srv.successor->id)
  if @(srv.node.id) != @nodesByID[successor_id](srv.predecessor->id):
    print __NODE__, "'s successor link is asymmetric."
end
```

and symmetrically for the predecessor's successor. Recall that in the absence of a

node specifier, a variable in a distributed command applies to the node that triggered the watchpoint. Also, the index `nodesByID` is maintained as described in Section 4.1.2.

Such graph invariants are especially straightforward to write given `Friday`'s ability to manipulate the local state of all nodes in a distributed execution. For instance, in more complex overlays that enhance rings with long links (such as distributed hash tables), a programmer can trivially maintain an inverted index of all nodes' local views of the identifier space and that space's partitioning among peers, pinpointing collisions. The complexity of such examples is marginally greater than the link symmetry example shown above, and we omit them here for brevity.

Ring Consistency Statistics

The techniques of Section 4.2.1 will typically issue inconsistency warnings many times during any system execution. Such inconsistencies occur transiently even when the system operates perfectly while an update occurs, e.g., when a new node is inserted into the ring. Without transactional semantics across all involved nodes in which checks are performed only before or after a transition, such warnings are unavoidable. Therefore, an interesting question might be "how long do such inconsistencies last?" Given this measure, a programmer can conclude whether an inconsistency warning is a transient or a pathological one.

In `Friday`, we can characterize the execution of the system by computing the fraction of time during which the ring topology lies in an inconsistent state. Specifically, by augmenting the monitoring statements from Section 4.2.1, one can instrument transitions from consistent to inconsistent state and back, to keep track of the time when those transitions occur, and averaging over the whole system.

```
watch srv.successor, srv.predecessor
command
  myID = @(srv.node.id)
  successorID = @(srv.successor->id)
  predecessorID = @(srv.predecessor->id)
  if not (myID == @nodesByID[successorID](srv.predecessor->id)
              == @nodesByID[predecessorID](srv.successor->id) ):
```

```
    # inconsistent now
    if consistent[myID]:
      consistentTimes += (@(__REALCLOCK__) - lastEventTime[myID])
      consistent[myID] = False
      lastEventTime[myID] = @(__REALCLOCK__)
  else:
    # converse: consistent now
    if not consistent[myID]:
      inconsistentTimes += @((__REALCLOCK__) - lastEventTime[myID])
      consistent[myID] = True
      lastEventTime[myID] = @(__REALCLOCK__)
  cont
end

py consistent = {}
py lastEventTime = {}
py consistentTimes = inconsistentTimes = 0
```

This example illustrates how to keep track of how much time each replayed machine is in the consistent or inconsistent state, with regards to its ring links. The monitoring specification keeps track of the amounts of time node i is consistent or inconsistent in the debugging counters `consistentTimes` and `inconsistentTimes`, respectively. Also, it remembers when the last time a node switched to consistency or inconsistency in the debugging hash tables `consistent` and `inconsistent`, respectively. When the distributed commands are triggered, if the node is now inconsistent but was not before (the last time of turning consistent is non-empty), the length of the just-ended period of consistency is computed and added to the thus-far sum of consistency periods. The case for inconsistency periods is symmetric and computed in the "else" clause.

Periodically, or eventually, the relevant ratios can be computed as the ratio of inconsistent interval sums over the total time spent in the experiment, and the whole system might be characterized taking an average or median of those ratios.

State Oscillation

The previous case studies have focused on detecting routing consistency. Consider a scenario in which a system operator has used those tools to note a large amount of inconsistency. She would next like to determine the reason.

One common cause of routing inconsistency is a network link that, whether due to high loss rates or intermittent hardware failure, makes a machine repeatedly disappear and reappear to its neighbor across the link. This oscillation may cause routes through the nodes to flap to backup links, or even create routing wormholes and black holes. The system operator could analyze the degree of oscillation in her network with the following simple `Friday` breakpoint commands.

```
break remove_finger
command
  finger = @(f->node.addr)
  # 'f' is parameter to remove_finger()
  eventTable = routeEvents[@(srv.node.addr)]
  if finger not in eventTable:
    eventTable[finger] = []
  eventTable[finger].append(("DOWN",__LOGICALCLOCK__))
  cont
end

break insert_finger
command
  finger = @(addr)
  # 'addr' is parameter to insert_finger()
  eventTable = routeEvents[@(srv.node.addr)]
  if finger in eventTable:
    lastEvent,time = eventTable[finger][-1]
    if lastEvent == "DOWN":
      eventTable[finger].append(("UP",__LOGICALCLOCK__))
  cont
end
```

The first command adds a log entry to the debugging table `routeEvents` (initialized

elsewhere) each time a routing peer, or *finger*, is discarded from the routing table. The second command adds a complementary log entry if the node is reinserted. The two commands are slightly asymmetric because `insert_finger` may be called redundantly for existing fingers, and also because we wish to ignore the initial insertion for each finger. The use of virtual clocks here allows us to correlate log entries across neighbors.

Mis-delivered Packets

Our last study moves beyond calculating the frequency and duration of routing inconsistencies to check an execution for actual occurrences of packets being mis-delivered. To do so, we first build a table containing the application-specific ID for each node. We intentionally extract the ID as a string, rather than the internal binary representation, so that we can use the application function `atoid` to regenerate this binary representation on demand. This approach is less efficient than simply copying the internal ID but allows us to demonstrate the ability in `Friday` to pass debugger data back into application functions.

```
py ids =
py failures = []

break chord.c:58
command
  # 'id' is string from configuration file
  ids[__NODE__] = @((char*)id)
  cont
end

break process.c:63
command
  failures.append( ("ALONE",ids[__NODE__], __LOGICALCLOCK__) )
  cont
end

break process.c:69
command
```

```
  for peer in __ALL__:
    @((chordID)_liblog_workspace = atoid("%(ids[peer])s"))
    if @(is_between((chordID*)&_liblog_workspace, packet_id, &successor->id)) :
      failures.append( ("MISSING",ids[__NODE__], ids[peer], __LOGICALCLOCK__) )
      break
  cont
end
```

We use two breakpoints in the packet-delivery method to detect mis-delivered packets. The first is located on the clause that handles empty networks. Because we are running these tests on non-trivial networks, a node should never believe that it is alone.

The second breakpoint checks every packet that the process believes has reached its best destination. We iterate across all peers in the network, using the `atoid` function to load the peer's ID into application scratch space and then to check ownership of the packet's ID using the application logic found in the `is_between` function[1]. If a better destination can be found, we log the packet ID as a failure. Construction of such a global index of all nodes is a powerful technique of catching inconsistencies that is virtually impossible in a distributed and efficient fashion.

It would be more efficient to build a sorted shadow ring and check for gaps in the Python command, but this method does not require the operator to understand the routing logic encoded in `is_between`. And, more importantly for us, this method showcases the ability of `Friday` to call application functions and provides a resource-intensive case study for evaluation in the following section.

4.2.2 A Reliable Communication Toolkit

In the second scenario, we investigate `Tk` [Sub05], a toolkit that allows nodes in a distributed system to communicate reliably in the presence of k adversaries. The only requirement for reliability is the existence of at least k disjoint paths between communicating nodes. To ensure this requirement is met, each node pieces together

[1]The `_liblog_workspace`, linked into the application's address space by `Liblog`, provides that scratch space for passing large arguments by reference.

a global graph of the distributed system based on path-vector messages and then computes the number of disjoint paths from itself to every other node using the max-flow algorithm. A bug in the disjoint path computation or path-vector propagation that mistakenly registers k or more disjoint paths would seriously undermine the security of the protocol. Here we show how to detect such a bug.

Maintaining a Connectivity Graph

When performing any global computation, including disjoint-path computation, a graph of the distributed system is a prerequisite. The predicate below constructs such a graph by keeping track of the connection status of each node's neighbors.

```
py graph = zero_matrix(10, 10)

break server.cpp:355
command
  neighbor_pointer = "(*(i->_M_node))"
  neighbor_status_addr = @(&(%(neighbor_pointer)s->status))

  # Set watchpoint at memory location neighbor_status_addr
  # with parameter neighbor_pointer and associated command.
  watchpoint(["*%d"%neighbor_status_addr], np=@(%(neighbor_pointer)s))
  command
    status = @((((Neighbor*)(%(np)d))->status))
    neighbor_id = @((((Neighbor*)(%(np)d))->id))
    my_id = @(server->id)

    if status > 0:
      graph[my_id][neighbor_id] = 1
      compute_disjoint_paths() # Explained below.
    cont
  end
  cont
end
```

This example showcases the use of nested watchpoints, which are necessary when a watchpoint must be set at a specific program location. In this application, a neigh-

bor's connection status variable is available only when the neighbor's object is in scope. Thus, we place a breakpoint at a location where all neighbor objects are enumerated, and as they are enumerated, we place a watchpoint on each neighbor object's connection status variable. When a watchpoint fires, we set the corresponding flag in an adjacency matrix.

A connection status watchpoint can be triggered from many programs locations, making it hard to determine what variables will be in scope for use within the watchpoint handler. In our example, we bind a watchpoint handler's `np` argument to the corresponding neighbor object pointer, thereby allowing the handler to access the neighbor object's state even though a pointer to it may not be in the application's dynamic scope.

Computing Disjoint Paths

The following example checks the toolkit's disjoint path computation by running a centralized version of the disjoint path algorithm on the global graph created in the previous example. The predicate records the time at which the k-path requirement was met, if ever. This timing information can then be used to detect disagreement between `Friday` and the application or to determine node convergence time, among other things.

```
py time_friday_found_k_paths = zero_matrix(10, 10)

def compute_disjoint_paths():
  my_id = @(server->id)
  k = @(server->k)

  for sink in range(len(graph)):
    friday_num_disjoint_paths = len(vertex_disjoint_paths(graph, my_id, sink))

    if friday_num_disjoint_paths >= k:
      time_friday_found_k_paths[my_id][sink] = __VCLOCK__
```

The disjoint path algorithm we implemented in `vertex_disjoint_paths`, not shown here, employs a brute force approach—it examines all k combinations of paths

Benchmark	Latency (ms)
False Positive	13.2
Null Command	15.6
Value Read	15.9
Value Write	15.9
Function Call	26.1
Safe Call	16.5

Table 4.1: Micro-benchmarks - single watchpoint

between source and destination nodes. A more efficient approach may be possible; however, since predicates are run offline, `Friday` affords us the luxury of using an easy-to-implement, albeit slow, algorithm.

4.3 Evaluation

In this section, we evaluate the performance of `Friday`, by reporting its overhead on fundamental operations (micro-benchmarks) and its impact on the replay of large distributed applications. Specifically, we evaluate the effects of false positives, of debugging computations, and of state manipulations in isolation, and then within replays of a routing overlay.

For our experiments we gathered logs from a 62-node i3/Chord overlay running on PlanetLab [BBC+04]. After the overlay had reached steady state, we manually restarted several nodes each minute for ten minutes, in order to force activity in the Chord maintenance routines. No additional lookup traffic was applied to the overlay. All measurements were taken from a 6 minute stretch in the middle of this turbulent period. The logs were replayed in `Friday` on a single workstation with a Pentium D 2.8GHz dual-core x86 processor and 2GB RAM, running the Fedora Core 4 OS with version 2.6.16 of the Linux kernel.

4.3.1 Micro-benchmarks

Here we evaluate `Friday` on six micro-benchmarks that illustrate the latency overhead required to watch data values and execute code on replayed process state. Table 4.1 contains latency measurements for the following operations:

- *False Positive*: A false positive occurs when a variable watchpoint is triggered by the modification of an unwatched variable that happens to occupy the same memory page as the watched variable.

- *Null Command*: A null command is the simplest command we can execute once a watchpoint has passed control to `Friday`. The overhead includes reading the new value (8 bytes) of the watched variable and evaluating a simple compiled Python object.

- *Value Read*: This is a single fetch of a variable from the state of one of the replayed processes for reading. The overhead involves contacting the appropriate GDB process and obtaining the requested variable's contents.

- *Value Write*: A value write updates the contents of a single variable in a single replayed process.

- *Function Call*: The command calls an application function that returns immediately. All watchpoints (only one in this experiment) must be disabled before, and re-enabled after the function call.

- *Safe Call*: The command is marked "safe" to obviate the extra watchpoint management.

These measurements indicate that the latency of handling the segmentation faults dominates the cost of processing a watchpoint. Our implementation of watchpoints is therefore sensitive to the false positive rate, and we could expect watchpoints that share memory pages with popular variables to slow replay significantly.

Fortunately, the same data suggests that executing the user commands attached to a watchpoint is inexpensive. Reading or writing variables or calling a safe function

adds less than a millisecond of latency over a null command, which is only a few milliseconds slower than a false positive. The safe function call is slightly slower than simple variable access, presumably due to the extra work by GDB to set up a temporary stack, marshal data, and clean up afterward.

A normal "unsafe" function call, on the other hand, is 50% slower than a safe one. The difference (9.6 ms) is attributed directly to the cost of temporarily disabling the watchpoint before invoking the function.

Next we break down the processing latency by major phases:

- *Unprotect*: Temporarily disable memory protection on the watched variable's page, so that the faulting instruction can complete. This step requires calling `mprotect` for the application, through GDB.

- *Step*: Re-execute the faulting instruction, potentially modifying a watched variable. Also requires setting and triggering one temporary breakpoint, used to return to the instruction from the segmentation fault handler.

- *Reprotect*: Re-enable memory protection with `mprotect`.

- *Check and Execute*: If the faulting address falls in a watched variable (as opposed to a false positive), its new value is extracted from GDB. If the value has changed, any attached command is evaluated by the Python interpreter. The command may interact with GDB further.

- *Other*: Miscellaneous tasks, including reading the faulting address and PC from the signal's user context structure.

Figure 4.3 highlights how the steps required to process a false positive also consume the same amount of time for any type of watchpoint hit. The dark segments in the middle of each bar show the portion required to execute the user command. It is small and approximately equal for each case except the unsafe function call, where it dominates.

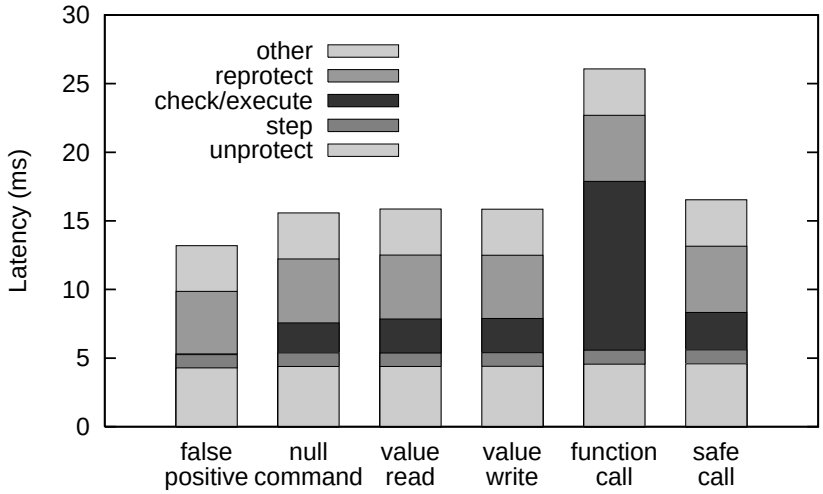

Figure 4.3: Latency breakdown for various watchpoint events.

4.3.2 Micro-benchmarks: Scaling of Commands

Next we explored the scaling behaviour of the four command micro-benchmarks: *value read*, *value write*, *function call*, and *safe call*. Figure 4.4 shows the cost of processing a watchpoint as the command reads, writes, or calls a function in an increasing number of nodes. All data points for each graph are averaged over the same number of watchpoints; the latency increases because more GDB instances must be contacted.

The figure includes the best-fit slope for each curve, which approximates the overhead added for each additional node that the command reads, writes, or calls. For most of the curves this amount closely matches the difference between a null command and the corresponding single-node reference. In contrast, the unsafe function call benchmark increases at a faster rate—almost double—and with higher variance than predicted by the single node overhead. We attribute both phenomena to greater contention in the replay host's memory hierarchy due to the extra memory protection operations.

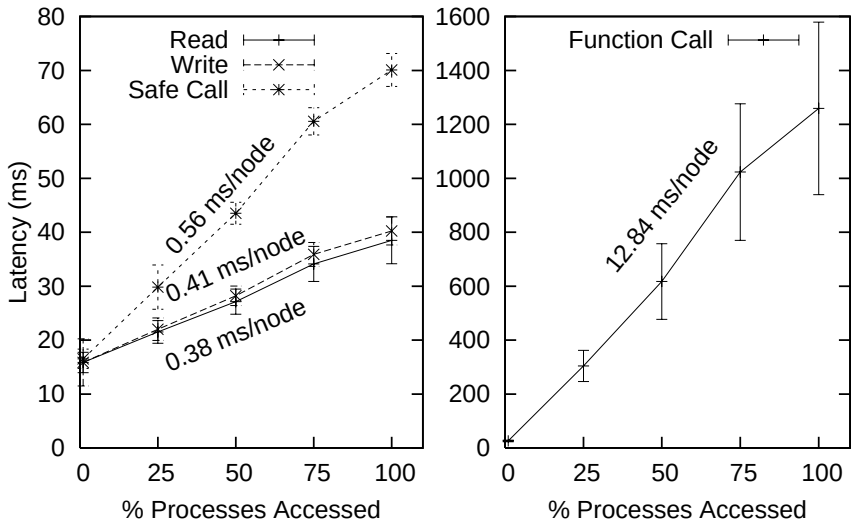

Figure 4.4: Microbenchmarks indicating latency and first standard deviation (y axis), as a function of the percentage of nodes involved in the operation (x axis). The population contains 62 nodes.

4.3.3 Micro-benchmarks on Replayed Chord

We continue by evaluating how the same primitive operations described in the previous section affect a baseline replay of a distributed application. For each benchmark, we average across 6 consecutive minute-long periods from the i3/Chord overlay logs described above.

We establish a replay baseline by replaying all 62 traced nodes in `Liblog` without additional debugging tasks. Average replay slowdown is $3.12x$, with a standard deviation of $.08x$ over the 6 samples. `Liblog` achieves a slowdown less than the expected $62x$ by skipping idle periods in each process. By comparison, simply replaying the logs in GDB, but without `Liblog`, ran 11 times faster, for a replay *speedup* of $3.5x$. The difference between GDB and `Liblog` is due to the scheduling overhead required to keep the 62 processes replaying consistently. `Liblog` must continually stop the running process, check its progress, and swap in a new process to keep their virtual clocks synchronized. Conversely, we let GDB replay each log fully before moving on to the next.

To measure false positives, we add an otherwise inconsequential watchpoint on a variable inhabiting a memory page that is written about 4.7 times per second

Benchmark	Slowdown	(dev)	Relative
No Watchpoints	3.12	(.08)	1
False Positives Only	7.95	(0.22)	2.55
Null Command	8.24	(0.24)	2.64
Value Read	8.25	(0.17)	2.65
Value Write	8.26	(0.21)	2.65
Function Call	9.01	(0.27)	2.89
Safe Call	8.45	(0.26)	2.71

Table 4.2: Micro-benchmarks: slowdown of Chord replay for watchpoints with different commands.

per replayed node; the total average replay slowdown goes up to $7.95x$ ($0.2x$ standard deviation), or $2.55x$ slower than baseline replay. This is greater than what our microbenchmarks predict: 4.7 triggered watchpoints per second should expand every replayed second from the baseline 3.12 second by an additional $4.7 \times 62 \times 0.0132 = 3.87$ seconds for a slowdown of $4.87x$. We conjecture that this difference is caused by cache contention on the replay machine, though further testing will be required to validate this.

To measure `Friday`'s slowdown for the various types of watchpoint commands, we set a watchpoint on a variable that is modified once a second on each node. This watchpoint falls on the same memory page as in the previous experiment, so we now see one watchpoint hit and 3.7 false positives per second. The slowdown for each type of command is listed in Table 4.2.

The same basic trends from the micro-benchmarks appear here: function calls are more expensive than other commands, which are only slightly slower than null commands. Significantly, the relative cost of the commands is dwarfed by the cost of handling false positives. This is expected, because the latency of processing a false positive is almost as large as a watchpoint hit, and because the number of false positives is much greater than the number of hits for this experiment. We examine different workloads later, in Section 4.3.4.

First, we scale the number of replayed nodes on whose state we place watchpoints, to verify that replay performance scales with the number of watchpoints. These experiments complement the earlier set which verified the scalability of the commands.

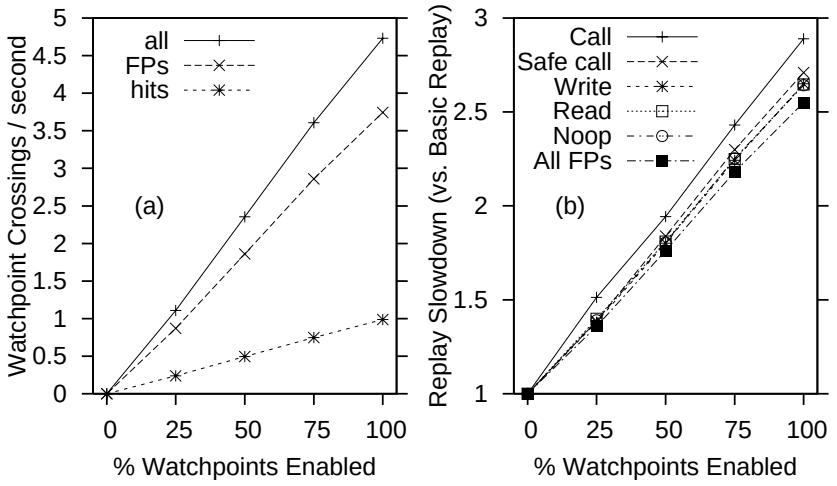

Figure 4.5: (a) Number of watchpoints crossed vs. percentage of nodes with watchpoints enabled (i3/Chord logs). Approximately linear. (b) Replay slowdown vs. percentage of nodes with watchpoints enabled, relative to baseline replay (i3/Chord logs).

As expected, as the number of memory pages incurring false positives grows, replay slows down relative to the baseline. Figure 4.5(a) shows that the rate at which watchpoints are crossed—both hits and false positives—increases as more processes enable watchpoints. The correlation is not perfectly linear, because some nodes were more active in the i3/Chord overlay and executed the watched inner loop more often than others.

Figure 4.5(b) plots the relative slowdown caused by the different types of commands as the watchpoint rate increases. These lines suggest that `Friday` does indeed scale with the number of watchpoints enabled and false positives triggered.

4.3.4 Case Studies

Finally, we turn to the performance overheads incurred by `Friday` in the case studies from Section 4.2. Unlike the experiments up to this point, these case studies include realistic and useful commands. They exhibit a range of performance, and two of them employ distributed breakpoints instead of watchpoints.

We used `Friday` to replay the same logs used in earlier experiments with the predicates for Ring Consistency Statistics, (Section 4.2.1), State Oscillation (Section 4.2.1),

Predicate	Slowdown
None	1.00
Ring Consistency Stat.	2.53
State Oscillation	1.48
Mis-delivered Packets	9.05
Software Watchpoints	8470.0

Table 4.3: Normalized replay slowdown under three different case studies. The last row gives the slowdown for the Ring Consistency Statistics predicate when implemented in GDB with single-stepping.

and Mis-delivered Packets (Section 4.2.1). Figure 4.6 plots the relative replay speed vs. baseline replay against the percentage of nodes on which the predicates are enabled. Table 4.3 summarizes the results. Results with the examples of Section 4.2.2 were comparable, giving a 100%-coverage slowdown of about 14 with a population of 10 nodes.

Looking at the table first, we see that the three case studies range from 1.5 to 9 times slower than baseline replay. For comparison, we modified `Friday` to use software watchpoints in GDB instead of our memory protection-based system, and reran the Ring Consistency Statistics predicate. As the table shows, that experiment took over 8000 times longer than basic replay, or about 3000 times slower than `Friday`'s watchpoints. GDB's software watchpoints are implemented by single-stepping through the execution, which consumes thousands of instructions per step. The individual memory protection operations used by `Friday` are even more expensive but their cost can be amortized across thousands of non-faulting instructions.

Turning to Figure 4.6, the performance of the Ring Consistency Statistics predicate closely matches that of the micro-benchmarks in the previous section (cf., Figure 4.5(b)). This fact is not surprising: performance here is dominated by the false positive rate, because these predicates perform little computation when triggered. Furthermore the predicate measured here and the micro-benchmarks in Figure 4.5(b) all watch variables located on the same page of memory, due to the internal structure of the i3/Chord application, so their false positive rates are the same.

The figure shows that the State Oscillation predicate encounters more breakpoints than the Ring Consistency predicate does watchpoints. However, handling a break-

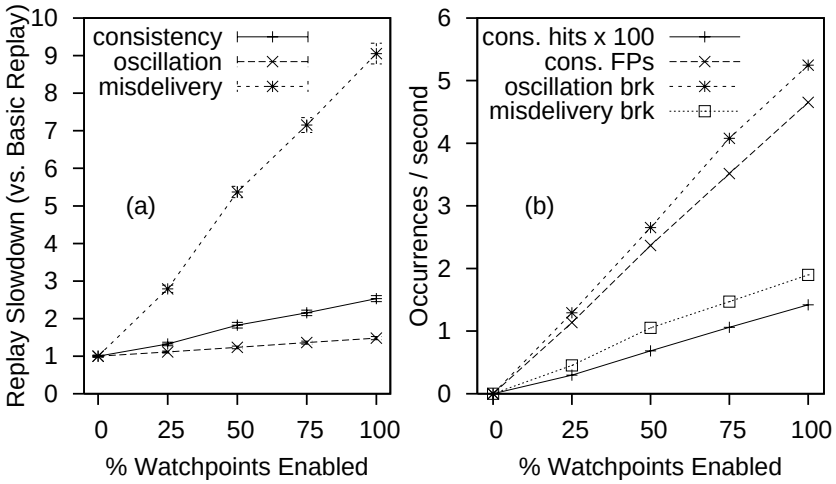

Figure 4.6: (a) Replay slowdown statistics for case study predicate performance vs. percentage of nodes with watchpoints enabled. (b) Watchpoint, breakpoint, and false positive rates vs. percentage of nodes with watchpoints/breakpoints enabled.

point is almost free, and the commands are similar in complexity, so `Friday` runs much faster for State Oscillation predicates.

The last case study, which checks for Mis-delivered Packets, hit even fewer breakpoints, and ran the fewest number of commands. Those commands were very resource-intensive, however, requiring dozens of (safe) function calls each time. Overall performance, as shown in Figure 4.6(a), is the slowest of the three predicates.

4.4 Summary

`Friday` is a replay-based symbolic debugger for distributed applications that enables the developer to maintain global, comprehensive views of the system state. It extends the GDB debugger and `Liblog` replay library with distributed watchpoints, distributed breakpoints, and actions on distributed state. `Friday` provides programmers with sophisticated facilities for checking global invariants—such as routing consistency—on distributed executions. We have described the design, implementation, usage cases, and performance evaluation for `Friday`, showing it to be powerful and efficient for demanding distributed debugging tasks that were, thus far, underserved by commercial or research debugging tools.

Chapter 5

Conclusion

In this dissertation we have presented our efforts to improve the state of the art for debugging support for distributed applications. Our work produced two novel tools: first, `Liblog`, a system for deterministically replaying POSIX applications; and second, `Friday`, the next step towards distributed comprehension. Together, these tools provide a level of visibility into the runtime behaviour of real distributed applications that was previously infeasible.

5.1 Future Work

`Liblog` and `Friday` are only the first steps in our distributed replay research. Now that the foundation has been laid, there are dozens of fascinating debugging and analytical tools that could be built on top with relatively little effort. We outline a few here.

First, could the information provided by `Liblog` help a programmer trace corrupted state back to its source? If we have a distributed application propagating garbage or triggering remote failures, we can search through the execution history by finding all messages that could have triggered a local fault, replaying the processes that sent those messages, and then finding all the messages arriving on those nodes that could have imported the problem. Iterating this simple algorithm will eventually find the origin of the problem, but the fan-out is huge. We could build a system using

`Friday`'s support for predicate evaluation to verify valid state and prune the search space significantly. The result could be tremendously useful for debugging routing applications in particular.

A related research direction would be to expose messages as first-class objects for `Friday`'s distributed predicates. This would allow the user to easily correlate transmission and reception, detect dropped messages, and trace message paths across a network.

These abilities would also enable new visualization tools that provide a higher granularity and accuracy than currently possible, and for an entire distributed application. Beyond drawing routes and packet transmissions, `Friday`-enabled visualizers could display statistics computed from global state, with no prior planning and predicate-specific instrumentation. The user could even search for arbitrary application state.

Another opportunity is to combine the power of `Liblog` with ideas from Delta Debugging [CZ05]. First, we would add the ability to `libreplay` to intentionally alter history during replay. For example, it could pretend a message was not received, or swap the arrival order of two messages, or change the return value of a system call. This selective non-determinism would let us experiment with alternate executions and search for changes that avoid non-deterministic failures. The goal of Delta Debugging is to automate that search.

An automated tool that could detect even a few types of bugs would be a huge boon for developers of distributed applications. The ability to replay execution repeatedly and evaluate predicates on application state is already provided by `Liblog` and `Friday`. We need now to develop a good notion of variations on an execution log and the search algorithm itself. And, of course, we must explore the soundness of replay in an alternate history. How do we adjust for new, unlogged behaviour from the application? How long can the alternate execution remain stable? Could we simply let the application run "live", without the log, in an emulated environment? Many questions remain.

In addition to these follow-up projects, we also have several improvements to the core `Liblog` and `Friday` tools that merit investigation. One immediate idea is to

reduce watchpoint overheads via the reimplementation of the `malloc` library call and memory page fragmentation, or through intermediate binary representations, such as those provided by the Valgrind [Val] tool. Building a hybrid system that leverages the limited hardware watchpoints, yet gracefully degrades to slower methods, would also be rewarding.

Another feature we seek to include in the near future is the ability to checkpoint `Friday` state during replay. This would allow a programmer to replay in `Friday` a traced session with its predicates from its beginning, constructing any debugging state along the way, but only restarting further debugging runs from intermediate checkpoints, without the need for reconstruction of debugging state. This would also make it easier to debug the debugger on those occasions when `Friday` predicates themselves become complicated and/or buggy.

We are considering better support for thread-level parallelism in `Friday` and `Liblog`. Currently threads execute serially with a cooperative threading model, to order operations on shared memory. We have also designed a mechanism that supports preemptive scheduling in userland, and we are also exploring techniques for allowing full parallelism in controlled situations.

Further down the road, we are very interested in improving the ability of the system operator to reason about time. Perhaps our virtual clocks could be optimized to track real or average time more closely when the distributed clocks are poorly synchronized. Better yet, it could be helpful to make stronger statements in the face of concurrency and race conditions. For example, could `Friday` guarantee that an invariant always held for an execution, given all possible interleavings of concurrent events?

5.2 Afterword

Thus ends this dissertation: a small book but a huge chapter in my life.

My advisor, Ion, first suggesting building a replay tool for Chord in Fall 2003. I initially rejected the idea, suspecting that the details of capturing non-determinism and integrating with GDB would be devilish, and that only a fool would tackle such

a wide-open research problem with so much technical grunt work.

A few months later I returned to the topic, because we needed better debugging tools. This time I was drawn to the wide-open nature of the research problem, and foolishly optimistic about the technical details and the devil therein. That was almost three years ago, and by now we have a rather powerful and mostly bug-free prototype. I hope that it may help others in their own work.

I came to graduate school for the challenge. I leave feeling that I have met the challenge and succeeded. Or maybe we'll call it a draw. In any case, I now move on to the next chapter. Thank you for reading.

Bibliography

[AMW+03] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, October 2003.

[BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *Proceedings of ACM Special Interest Group On Data Communications (SIGCOMM)*, pages 353–366, Portland, OR, USA, 2004.

[BBC+04] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, pages 253–266, March 2004.

[BDIM04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, December 2004.

[CAK+04] Mike Y. Chen, Anthony Accardi, Emre Kıcıman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based Failure and Evolution

Management. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, March 2004.

[CECZ05] Anupam Chanda, Khaled Elmeleegy, Alan Cox, and Willy Zwaenepoel. Causeway: System Support for Controlling and Analyzing the Execution of Distributed Programs. In *Proceedings of USENIX Hot Topics in Operating System (HotOS)*, Santa Fe, NM, USA, June 2005.

[CGC+03] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[Ckp] Ckpt project website. http://www.cs.wisc.edu/ zandy/ckpt/.

[CZ05] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, May 2005.

[DFD96] Carl Dionne, Marc Feeley, and Jocelyn Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, CA, August 1996.

[FLRS05] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and dhts. In *Proceedings of WORLDS*, December 2005.

[GDB] Gnu debugger website. http://gnu.org/software/gdb/.

[Har02] Timothy L. Harris. Dependable Software Needs Pervasive Debugging (Extended Abstract). In *Proceedings of ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[HH05] Alex Ho and Steven Hand. On the design of a pervasive debugger. In *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, September 2005.

[HM93] Jefferely Hollingsworth and Barton Miller. Dynamic Control of Performance Monitoring of Large Scale Parallel Systems. In *Proceedings of Super Computing (SC)*, Tokyo, Japan, July 1993.

[Hus02] Joel Huselius. Debugging parallel systems: A state of the art report. Technical Report MDH-MRTC-63/2002-1-SE, Maelardalen Real-Time Research Centre, September 2002.

[JKDC05] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through VulnerabilitySpecific Predicates. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 91–104, Brighton, UK, October 2005.

[JKK+06] Dilip Joseph, Jayanthkumar Kannan, Ayumu Kubota, Karthik Lakshminarayanan, Ion Stoica, and Klaus Wehrle. Ocala: An architecture for supporting legacy applications over overlays. In *Proceedings of NSDI*, May 2006.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, June 2005.

[KMLC05] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2005.

[KSC00] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed java applications. In *Proceedings of International Parallel and Distributed Processing Symposium*, May 2000.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LCH+05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Brighton, UK, October 2005.

[NM92] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the International Conference on Supercomputing*, November 1992.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture*, 2005.

[PL] Planet-lab project website. http://planet-lab.org/.

[Pur] Purify website. http://ibm.com/software/awdtools/purify/.

[Py] Python project website. http://python.org/.

[RBdK99] Michiel Ronsse, Koenraad De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay for an mpi-based multi-threaded runtime system. In *Proceedings of the International Conference Parallel Computing*, 1999.

[RWM+06] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, May 2006.

[RZ97] Michiel Ronsse and Willy Zwaenepoel. Execution replay for treadmarks. In *Proceedings of EUROMICRO Workshop on Parallel and Distributed Processing*, January 1997.

[Sai05] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, September 2005.

[SAZ+02a] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of ACM Special Interest Group On Data Communications (SIGCOMM)*, 2002.

[SAZ+02b] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, August 2002.

[SH] Simics hindsight. http://www.virtutech.com/products/simics-hindsight.html.

[SKAZ04] Sudarshan M. Srinivashan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference*, June 2004.

[SMRD06] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Drushel. Using Queries for Distributed Monitoring and Forensics. In *Proceedings of EuroSys*, pages 389–402, Leuven, Belgium, April 2006.

[Sno88] Richard Snodgrass. A Relations Approach to Monitoring Complex Systems. *IEEE Transactions on Computer Systems*, 6(2):157–196, 1988.

[SSL] Openssl project website. http://openssl.org/.

[Sub05] Lakshminarayanan Subramanian. *Decentralized Security Mechanisms for Routing Protocols*. PhD thesis, University of California at Berkeley, 2005.

[TH00] Henrik Thane and Hans Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of 12th Euromicro Conference on Real-Time Systems*, June 2000.

[Val] Valgrind project website. http://valgrind.org/.

[vRBDV02] Robbert van Renesse, Kenneth P. Birman, Dan Dumitriu, and Werner Vogel. Scalable management and data mining using Astrolabe. In *Proc. First International Workshop on Peer-to-Peer Systems (IPTPS '01)*, Cambridge, MA, March 2002.

[Wah92] Robert Wahbe. Efficient data breakpoints. 1992.

[WSY91] Ouri Wolfson, Soumitra Sengupta, and Yechiam Yemini. Managing Communication Networks by Monitoring Databases. *IEEE Transactions on Software Engineering*, 17(9):944–953, 1991.

[XBH03] Min Xu, Rastislav Bodik, and Mark Hill. A flight data recorder for enabling fullsystem multiprocessor deterministic replay. In *30th International Symposium on Computer Architecture*, 2003.

[YD04] Praveen Yalagandula and Mike Dahlin. A Scalable Distributed Information Management System. In *Proceedings of ACM Special Interest Group On Data Communications (SIGCOMM)*, Portland, OR, USA, September 2004.