`Friday`: Global Comprehension for Distributed Replay

Dennis Geels[*], Gautam Altekar[‡], Petros Maniatis[φ], Timothy Roscoe[†], Ion Stoica[‡]

[*]*Google, Inc.,*[‡]*University of California at Berkeley, *[φ]*Intel Research Berkeley, *[†]*ETH Zürich*

Abstract

Debugging and profiling large-scale distributed applications is a daunting task. We present `Friday`, a system for debugging distributed applications that combines deterministic replay of components 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.

To evaluate `Friday`, we consider several distributed problems, including routing consistency in overlay networks, and temporal state abnormalities caused by route flaps. We show via micro-benchmarks and larger-scale application measurement that `Friday` can be used interactively to debug large distributed applications under replay on common hardware.

1 Introduction

Distributed applications are complex, hard to design and implement, and harder to validate once deployed. The difficulty derives from the distribution of application state across many distinct execution environments, which can fail individually or in concert, span large geographic areas, be connected by brittle network channels, and operate at varying speeds and capabilities. Correct operation is frequently a function not only of single-component behavior, 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 de-

bugger to check continuously that no routing cycles exist across the running state of the entire distributed system, as easily as she can read the current state of program variables in typical symbolic debuggers.

`Friday`, the system we present in this paper, is a first step towards realizing this vision. `Friday` (1) captures the distributed execution of a system, (2) replays the captured execution trace within a symbolic debugger, and (3) extends the debugger's programmability for complex predicates that involve the *whole* state of the replayed system. 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.

Capture and replay in `Friday` are performed using `liblog` [8], which can record distributed executions and then replay them consistently. Replay takes place under the control of a symbolic debugger, which provides access to internal application state. But simple replay does not supply the global system view required to diagnose emergent misbehavior of the application as a whole.

For global predicate monitoring or replayed applications (the subject of this paper), `Friday` combines the flexibility of symbolic debuggers on each replayed node, with the power of a general-purpose, embedded scripting language, bridging the two to allow a single global invariant checker script to monitor and control the global execution of multiple, distinct replayed components.

Contributions: `Friday` makes two 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 de-

bugging statistics, building behavioral models, or shadowing global state.

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

Applicability: 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, grid job schedulers). Developers can evaluate global conditions during replay to validate a particular execution for correctness, to catch inconsistencies between a central management component and the actual state of the distributed managed components, and to express and iterate behavioral regression tests. For example, with 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 holds 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 or congestion), and finally identify the root cause by performing step-by-step debugging and analysis on a few instances involving such events, all without recompiling or annotating the source code.

Structure: We start with background on `liblog` in Section 2. Section 3 presents the design and implementation of `Friday`, and also discusses the limitations of the system. We then present in Section 4 concrete usage examples in the context of two distributed applications: the Chord DHT [25], and a reliable communication toolkit for Byzantine network faults [26]. We evaluate `Friday` both in terms of its primitives and these case studies in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2 Background: `liblog`

`Friday` leverages `liblog` [8] to deterministically and consistently replay the execution of a distributed application. We give a brief overview here.

`liblog` is a replay debugging tool for distributed `libc-` and POSIX C/C++-based applications on Linux/x86 computers. To achieve deterministic replay, each application thread records the side-effects of all nondeterministic system calls (e.g., `recvfrom()`, `select()`, etc.) to a local log. This is sufficient to replay the same execution, reproducing race conditions and non-deterministic failures, following the same code paths during replay, as well as the same file and network I/O, signals, and other IPC. `liblog` ensures causally consistent group replay, by maintaining Lamport clocks [16] during logging.
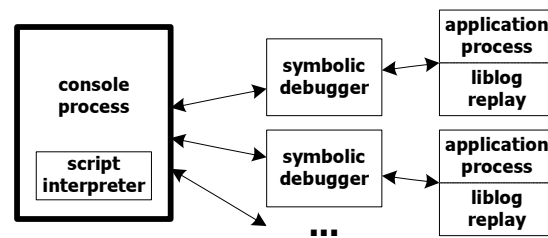

Figure 1: Overall architecture of `Friday`

`liblog` is incrementally deployable—it allows instrumented applications to communicate with applications that are not instrumented (e.g., DNS). `liblog` also supports replaying a subset of nodes without having to gather the logs of all nodes in the distributed system. Both incremental deployment and partial replay call for logging all incoming network traffic.

Finally, `liblog`'s library-based implementation requires neither virtualization nor kernel additions, resulting in a small per-process CPU and memory footprint. It is lightweight enough to comfortably replay 60 nodes on a Pentium D 2.8GHz machine with 2GB of RAM. We have also built a proof-of-concept cluster-replay mechanism that can scale this number with the size of the replay cluster to thousands of nodes.

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.

3 Design

`Friday` presents to users a central debugging console, which is connected to replay processes, each of which runs an instance of a traditional symbolic debugger such as GDB (see Figure 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, allowing full access to data analysis and control functions. `Friday` extends the debugger's commands to handle distributed breakpoints and watchpoints, and to inspect the whole system of debugged processes.

3.1 Distributed Watchpoints and Breakpoints

Traditional watchpoints allow a symbolic debugger to react—stop execution, display values, or evaluate a pred-

icate on the running state—when the process updates a particular variable location, named via a memory address or a symbolic name from the application's source.

`Friday`'s distributed watchpoints extend this functionality for variables and expressions from 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 on a single machine (here, #4) with "`4 watch successor`". The command "`[<node number>, ...]  watch <variable> [...]`" specifies both a set of nodes on which to watch variables (all by default), and a set of variables to watch. The node numbering is private to `Friday`; to identify a particular node by another identifier such as an IP address, an appropriate mapping can be provided (Section 3.2).

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 as a source line number or a function name. `Friday` allows the installation of such breakpoints on one, several, or all replayed nodes.

3.1.1 Implementation

`Friday` implements distributed watchpoints and breakpoints by setting local instances on each replay process and mapping their individual numbers and addresses to a global identifier. These maps are used to rewrite and forward disable/enable requests to a local instance, and also to map local events back to the global index when executing attached commands.

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 [29]. When a write to the variable's page occurs, the ensuing `SEGV` signal is intercepted, leading `Friday` to unprotect the page and completes the write, before evaluating any state manipulation scripts attached to the watchpoint.

This implementation can give rise to *false positives* when an unwatched variable sharing the page with a watchpoint is written. The more densely populated a memory page, the more such false positives occur. We decided that protection-based watchpoints are preferable to alternative implementations.

We explored but rejected four alternatives: 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 processors have small, hard limits on the number of watchpoint registers (a typical value is 8), as well as on the width of the watched variable (typically, a single machine word). For instance, watching for changes to a variable across tens of replayed nodes would not be possible if the replay machine has only 8 watchpoint registers. These limits are too restrictive for distributed predicates; however, we have planned a hybrid system that uses hardware watchpoints as a cache for our more flexible mechanism.

Single-stepping, or *software watchpoints*, execute one machine instruction and check variable modifications at each step. Unfortunately, single-stepping is prohibitively slow—we compare it to our method in Section 5.4 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 and is prone to mistakes.

Periodic sampling of watched variables (e.g., every k logical time ticks) 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 pinpoint value changes. Combined with replay checkpointing and backtracking, it might prove a valuable but not complete alternative.

3.1.2 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. `Friday`'s manipulation of memory page protection on the stack (Section 3.1.1) conflicts with GDB's initialization tasks when calling application functions, causing `mprotect` failures. To resolve the conflict, we replace GDB's calling facilities with our own, manipulating the application's `PC` directly, thereby complicating GDB's breakpoint maintenance. Thankfully, these complications are not triggered by any of our case studies presented in this paper.

3.2 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)behaviors.

`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 `nodes` 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 `nodes`:

```
nodes[@(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, by enabling watchpoints only when they are relevant.

3.2.1 Language Choice

The `Friday` commands triggered by watchpoints and breakpoints are written in Python, with extensions for interacting with distributed application state.

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 the application's native language (C/C++), in much the same way that C is used in IntroVirt [11]. Such an approach would allow the programmer to inline predicate code in the language of the application, thereby simplifying the interface between C/C++ constructs and a higher-level language. It would also eliminate the need to rely on GDB and Python for breakpoint/watchpoint detection and predicate evaluation, thereby reducing IPC-related overhead during replay. Unfortunately, this option calls for duplicating much of the introspection functionality (e.g., inspection of stack variables) already offered by GDB, and requires recompiling/reloading a C/C++ predicate library each time the user changes a predicate; we wanted to support a more interactive usage model.

At the opposite end of the spectrum, we could have used GDB's "command list" functionality to express distributed watchpoints and breakpoints. Unfortunately GDB commands lack the expressiveness of Python, such as its ability to construct new data structures, as well as the wealth of useful libraries. Using a general-purpose scripting framework like Python running at the console afforded us much more flexibility.

3.2.2 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 watch-

point. 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 4.2.1 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 during the function call. Unfortunately that precaution may be very costly (see Section 5). If the user is confident that a function will not modify protected memory, she may start the command with the `safe` keyword, which instructs `Friday` to leave 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.

3.3 Limitations

We have used `Friday` to debug large distributed applications. Though still a research prototype with rough edges and only a rudimentary user interface, 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 behavior, 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 5 has more details on `Friday` performance.

The second `Friday`-specific limitation involves replaying from the middle of a replay trace. Some `Friday` predicates build up their debugging state by observing the dynamic execution of a replayed application, and when starting from a checkpoint these predicates must rebuild that state through observation of a static snapshot of the application at that checkpoint. This process is straightforward for the applications we study in Section 4, but it may be more involved for applications with more complex data structures. We are working on a method for adding debugging state to `liblog` checkpoints at debug time, to avoid this complexity.

Thirdly, although we have found that `Friday`'s centralized and type-safe programming model makes predicates considerably simpler than the distributed algorithms they verify, `Friday` predicates often require some debugging themselves. For example, Python's dynamic type system allows us to refer to application variables that are not in dynamic scope, causing runtime errors.

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 the collection templates of the C++ Standard Template Library, 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 breakpoints 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 `liblog`'s large storage requirements for logs and an inability to log or replay threads in parallel on multi-processor machines.

4 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 [24], which has occupied networking and distributed research literature extensively. Then we turn to debugging `Tk`, a reliable communication toolkit [26], 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.

4.1 Routing Consistency

In this section, we describe a usage scenario in which `Friday` helps a programmer to drill down on a reported bug with i3/Chord. The symptom is the loss of a value stored within a distributed hash table: a user who did a `put(key,value)`, doing a `get(key)` later did not receive the `value` she put into the system before. We describe a debugging session for this scenario and outline specific uses of `Friday`'s facilities.

Our programmer, Frieda, starts with a set of logs given to her, and with the knowledge that two requests, a `put` and a `get` that should be consistent with each other appear to be inconsistent: the `get` returns something other than what the `put` placed into the system.

4.1.1 Identifying a distributed bug

Frieda would probably eliminate non-distributed kinds of bugs first, by establishing for instance that a node-local store does not leak data. To do that, she can monitor that the two requests are handled by the same node, and that the node did not lose the key-value pair between the two requests.

```
py getNode = None
py putNode = None
break process_data
command
  if @(packet_id) == failing_id:
    if is_get(@(packet_header)):
      getNode = @(srv.node.id)
    else:
      putNode = @(srv.node.id)
end
```

This breakpoint triggers every time a request is forwarded towards its final destination. Frieda will interactively store the appropriate message identifier in the Python variable `failing_id` and define the Python method `is_get`. At the end of this replay session, the variables `getNode` and `putNode` have the identifiers of the nodes that last serviced the two requests, and Frieda can read them through the `Friday` command line. If they are the same, then she would proceed to debug the sequence of operations executed at the common node between the `put` and the `get`. However, for the purposes of our scenario we assume that Frieda was surprised to find that the `put` and the `get` were serviced by different nodes. This leads her to believe that the system experienced *routing inconsistency*, a common problem in distributed lookup services where the same lookup posed by different clients at the same time receives different responses.

4.1.2 Validating a Hypothesis

The natural next step for Frieda to take is to build a map of the consistent hashing offered by the system: which part of the identifier space does each node think it is responsible for? If the same parts of the identifier space are claimed by different nodes, that might explain why the same key was serviced by different nodes for the `put` and the `get` requests. Typically, a node believes that its immediate successor owns the range of the identifier space between its own identifier and that of its successor.

The following breakpoint is set at the point that a node sets its identifier (which does not change subsequently). It uses the `ids` associative array to map `Friday` nodes to Chord IDs.

```
py ids = {}
break chord.c:58
command
  ids[__NODE__] = @((char*)id)
  cont
end
```

Now Frieda can use this information to check the correct delivery of requests for a given key as follows:

```
break process.c:69
command
  if @(packet_id) != failing_id:
    cont
  for peer in __ALL__:
    @((chordID)_liblog_workspace =
        atoid("%(ids[peer])s"))
    if @(is_between((chordID*)&_liblog_workspace,
            @(packet_id), &successor->id)):
      print "Request %s misdelivered to %s" %
                  (@(packet_id), @(successor->id))
      break
  cont
end
```

This breakpoint triggers whenever a node with ID `srv.node.id` believes it is delivering a packet with destination ID `packet_id` to its rightful destination: the node's
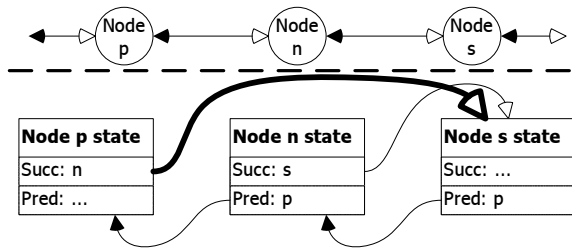
Figure 2: At the top, we show what node n believes the ring topology to be around it. At the bottom, we see the relevant state as stored by the involved nodes n, s and p. The thick routing entry from node p to s is inconsistent with n's view of the ring, indicating a source of routing consistency problems.

immediate successor with ID `srv.successor->id`, such that `packet_id` is in between identifiers `srv.node.id` and `srv.successor->id`. When that happens, this command figures out if the request in question is one of Frieda's problem requests, and if so, it finds out if there is a node that should be receiving the packet instead.

This check uses the native Chord function `atoid` to load the peer's ID into application scratch space (`_liblog_workspace`) and then invokes the Chord function `is_between` to perform the range check. Both of these functionalities could have been duplicated instead in Python, if Frieda suspected their implementation as the source of the problem.

This breakpoint command is a simple instance of a very powerful construct: a global index of all nodes in the system is very easy for `Friday` to construct at replay time but difficult or impossible to collect reliably and efficiently at runtime. Doing so would require transmitting each update (node insertion, etc.) to all nodes, presumably while all other communication is disabled to avoid inconsistencies. These updates would be expensive for large networks and could fail due to transient network conditions. Conversely, `Friday` can maintain a global view of the whole logged population, even if the nodes themselves could not talk to each other at runtime.

4.1.3 Searching for a Root Cause

The identified inconsistency told Frieda that she has a problem. Most likely, it tells her that part of her purported ring topology looks like Figure 2, in which the culprit node, p, believes its successor to be node s and delivers anything between identifiers p and s to s for processing, where instead all requests for identifiers between p and n belong to node n instead.

To dig deeper, Frieda can monitor ring consistency more closely, for instance by ensuring that ring edges are symmetric. Checking that successor/predecessor consistency conditions hold at all times is unnecessary. Instead, it is enough to check the conditions when a successor or predecessor pointer changes, and only check those spe-

cific conditions in which the changed pointers participate. Frieda can encode this in `Friday` as follows:

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

and symmetrically for the predecessor's successor. This would catch, for instance, the problem illustrated in Figure 2, which caused Frieda's problem.

4.1.4 How Often Is The Bad Thing Happening?

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. Frieda must figure out whether this inconsistency she uncovered occurs most of the time or infrequently; that knowledge can help her decide whether this is a behavior she should mask in her Chord implementation (e.g., by sending redundant `put` and `get` requests) or fix (e.g., by ensuring that nodes have agreed on their topological neighborhood before acting on topology changes).

In `Friday`, Frieda can compute the fraction of time during which the ring topology lies in an inconsistent state. Specifically, by augmenting the monitoring statements from Section 4.1.3, she 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 (@nodes[successorID](srv.predecessor->id)
      == @nodes[predecessorID](srv.successor->id)
      == myID): #inconsistent?
    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.

4.1.5 State Oscillation

If Frieda finds that most of the time such inconsistencies exist, she may decide this is indeed a bug and move to fix it by ensuring a node blocks requests while it agrees on link symmetry with its immediate neighbors.

In the unhappy case in which Frieda's system is indeed intended to have no such inconsistencies (i.e., she has already written the code that causes nodes to agree on link symmetry), she would like to determine what went wrong. She can do by testing a series of hypotheses.

One such hypothesis—which is frequently the case of inconsistencies in a broad range of distributed applications—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. Frieda can 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 formal parameter
  events = routeEvents[@(srv.node.addr)]
  if finger not in events:
    events[finger] = []
  events[finger].append(("DOWN",__LOGICALCLOCK__))
  cont
end

break insert_finger
command
  finger = @(addr) # addr is formal parameter
  events = routeEvents[@(srv.node.addr)]
  if finger in events:
    lastEvent,time = events[finger][-1]
```

```
    if lastEvent == "DOWN":
      events[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 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.

4.2 A Reliable Communication Toolkit

In the second scenario, we investigate Tk [26], 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 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.

4.2.1 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 a watchpoint dynamically
  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 neighbor'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.

4.2.2 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 between source and destination nodes. A more efficient approach calls for using the max-flow algorithm, but that's precisely the kind of implementation complexity we wish to avoid. Since predicates are run offline, `Friday` affords us the luxury of using an easy-to-implement, albeit slow, algorithm.

4.3 Discussion

As the preceding examples illustrate, the concept of an invariant may be hard to define in a distributed system. So-called invariants are violated even under correct operation for short periods of time and/or within subsets of the system. `Friday`'s embedded interpreter allows pro-grammers to encode what it means for a particular system to be "too inconsistent".

By gaining experience with the patterns frequently used by programmers to track global system properties that are transiently violated, we intend to explore improved high-level constructs for expressing such patterns as part of our future work.

The Python code that programmers write in the process of debugging programs with `Friday` can resemble the extra, temporary code added inline to systems when debugging with conventional tools: in cases where simple assertions or logging statements will not suffice, it is common for programmers to insert complex system checks which then trigger debugging code to investigate further the source of the unexpected condition.

In this respect, `Friday` might be seen as a system for "aspect-oriented debugging", since it maintains a strict separation between production code and diagnostic functionality. The use of a scripting language rather than C or C++ makes writing debugging code easier, and this can be done after compilation of the program binaries. However, `Friday` also offers facilities not feasible with an on-line aspect-oriented approach, such as access to global system state.

It has often been argued that debugging code should never really be disabled is a production distributed system. While we agree with this general sentiment, in `Friday` we draw a more nuanced distinction between code which is best executed all the time (such as configurable logging and assertion checks), and that which is only feasible or useful in the context of offline debugging. The latter includes the global state checks provided by `Friday`, something which, if implemented inline, would require additional inter-node communication and library support.

5 Performance

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 [3]. After the overlay had reached steady state, we manually restarted several nodes each minute for ten minutes, in order to force interesting events for 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

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 1: Micro-benchmarks - single watchpoint

RAM, running the Fedora Core 4 OS with version 2.6.16 of the Linux kernel.

5.1 Micro-benchmarks

Here we evaluate `Friday` on six micro-benchmarks that illustrate the overhead required to watch variables and execute code on replayed process state. Table 1 contains latency measurements for the following operations:

- *False Positive*: A watchpoint is triggered by the modification of an unwatched variable that occupies the same memory page as the watched variable.
- *Null Command*: 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*: A single fetch of a variable from one of the replayed processes. The overhead involves contacting the appropriate GDB process and reading the variable's contents.
- *Value Write*: Updates 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. This means our implementation of watchpoints is 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.
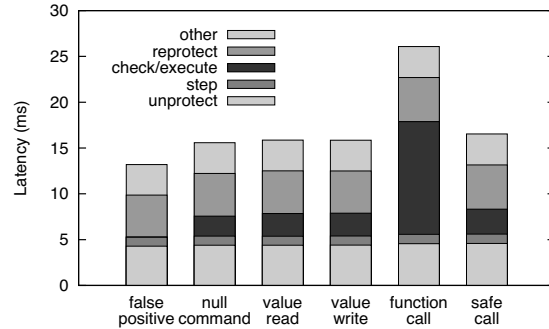

Figure 3: Latency breakdown for various watchpoint events.

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.

We break down the processing latency into 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. This requires a temporary breakpoint, used to return to the instruction from the fault handler.
- *Reprotect*: Re-enable 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.
- *Other*: Miscellaneous tasks, including reading the faulting address from the signal's user context.

Figure 3 shows that a false positive costs the same as a watchpoint hit. The dark segments in the middle of each bar show the portion required to execute the user command. It is small except the unsafe function call, where it dominates.

5.2 Micro-benchmarks: Scaling of Commands

Next we explored the scaling behavior of the four command micro-benchmarks: *value read*, *value write*, *function call*, and *safe call*. Figure 4 shows the cost of processing a watchpoint as the command accesses an increasing number of nodes. Each data point is 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 correspond-
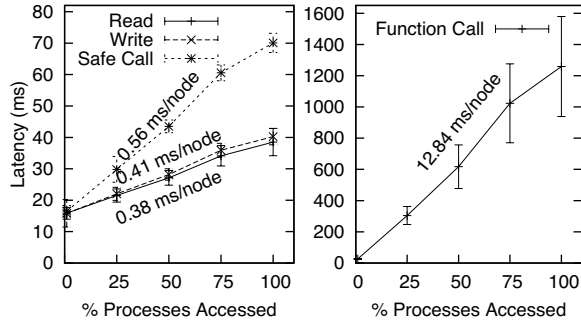
Figure 4: Micro-benchmarks indicating latency and first standard deviation (y axis), as a function of the percentage of nodes involved in the operation (x axis). The population contains 62 nodes.

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

ing 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.

5.3 Micro-benchmarks on 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. Other applications would experience more or less overhead, depending on the relative frequency of `libc` calls and watchpoint triggers.

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. For 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. Without `liblog`, we let GDB replay each log fully before moving on.

To measure false positives, we add a dummy watchpoint on a variable at a memory page written about 4.7 times per second 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 micro-benchmarks predict: 4.7 triggered watchpoints per second should expand every replayed second from the baseline 3.12 seconds by an additional $4.7 \times 62 \times 0.0132 = 3.87$ seconds for a slow-

down of $4.87x$. We conjecture that the extra slowdown is due to 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 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 5.4.

Next, 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.

As expected, as the number of memory pages incurring false positives grows, replay slows down. Figure 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 perfect, because some nodes were more active and executed the watched inner loop more often than others.

Figure 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.

5.4 Case Studies

Finally, we return to the case studies from Section 4. Unlike the micro-benchmarks, these case studies include realistic and useful commands. They exhibit a range of performance, and two of them employ distributed break-
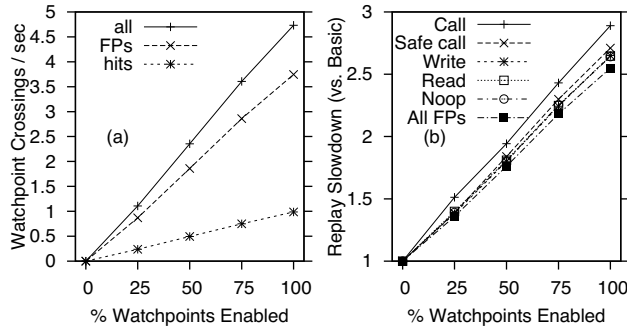
Figure 5: (a) Number of watchpoints crossed vs. percentage of nodes with watchpoints enabled (i3/Chord logs). Approximately linear. (b) Replay slowdown vs. percentage of nodes with watchpoints enabled, relative to baseline replay (i3/Chord logs).
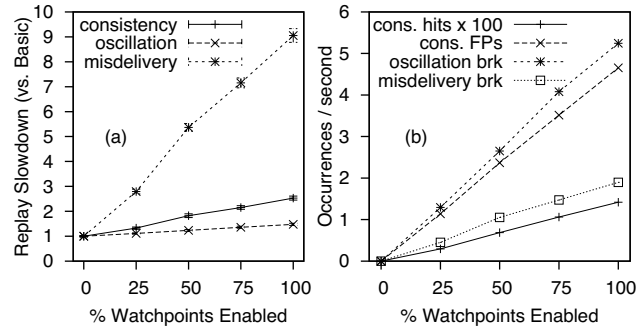

Figure 6: (a) Replay slowdown statistics for case study predicate performance vs. percentage of nodes with watchpoints enabled. (b) Watchpoint, breakpoint, and false positive rates vs. percentage of nodes with watchpoints/breakpoints enabled.

Predicate	Slowdown
None	1.00
Ring Consistency Stat.	2.53
w/Software Watchpoints	8470.0
State Oscillation	1.48
Misdelivered Packets	9.05

Table 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.

points instead of watchpoints.

We replayed the same logs used in earlier experiments with the predicates for Misdelivered Packets (Section 4.1.2), Ring Consistency Statistics (Section 4.1.4), and State Oscillation (Section 4.1.5). Figure 6 plots the relative replay speed against the percentage of nodes on which the predicates are enabled. Table 3 summarizes the results. Results with the case studies from Section 4.2 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 6, the performance of the Ring Consistency Statistics predicate closely matches that of the micro-benchmarks in the previous section (cf., Figure 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, both sets of predicates 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 breakpoint is almost free, and the commands are similar in complexity, so `Friday` runs much faster for State Oscillation predicates.

The Misdelivered Packets case study 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 6(a), is the slowest of the three predicates.

6 Related Work

`Friday` utilizes library interposition to obtain a replayable deterministic trace of distributed executions. The WiDS Checker [17] has many similar characteristics with some notable differences: whereas `Friday` operates on unmodified applications and checks predicates at single-machine-instruction granularity, the WiDS Checker is applicable only to applications developed with the WiDS toolkit and checks predicates at event-handler granularity. Similarly to `Friday`, Jockey [20] and Flashback [23] use system call interposition, binary rewriting, and operating system modifications to capture deterministic replayable traces, but only for a single node. DejaVu [15] targets distributed Java applications, but lacks the state manipulation facilities of `Friday`.

Further afield, much research has gone into replay debugging via virtualization, which can capture system effects below the system library level, first articulated by Harris [9]. Several projects have pursued that agenda

since [11, 13, 23], 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.

Moving away from replay debugging, many systems focus on extracting execution logs and then mining those logs for debugging purposes [1, 2, 5, 6, 10, 22]. Such systems face the challenge of reconstructing meaningful data- and control-flow from low-level logged monitoring information. `Friday` 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 at connections across layers (as with black-box approaches) or recompile the system (as with annotation-based systems).

Notable logging-based work in closer alignment with `Friday` comes from the Bi-directional, Distributed BackTracker (BDB) [14], XTrace [7], and Pip [19]. BDB and XTrace track and report causality among events within a distributed system, e.g., to trace identified backdoor programs backwards to their onset or, in the case of XTrace, to identify problems along cross-layer paths. Pip [19] works by comparing actual behavior and expected behavior to expose bugs. Such behaviors are defined as orderings of logged operations at participating threads and limits on the values of annotated and logged performance metrics. In both cases, the kinds of checks performed can be readily encoded in `Friday`, except for those dependent on kernel-level sensors, which lie beyond our library tracing granularity. However, the replay-based nature of `Friday` allows programmers to refine checks after repeated replays without the need for recompilation and fresh log extraction, as would be the case for disambiguating noisy tasks (e.g., directory listing filesystem operations in BDB) or for creating new sensors (e.g., heap size monitors when none were initially thought necessary in Pip).

At a more abstract level, model checking has been recently proposed as a tool for debugging distributed systems. Most notably, MaceMC [12] is a heuristic model checker for finding liveness violations in distributed applications built using the Mace language. As with model checking in general, MaceMC can exercise a distributed application over many more possible executions than any replay debugging system, include `Friday`, can. However replay systems, such as `Friday`, tend to capture more realistic problems than model checkers such as complex network failures and hardware malfunctions, and can typically operate on much longer actual executions than the combinatorial nature of model checking can permit.

A growing body of work is starting to look at on-line debugging [27], in contrast to the off-line nature of debuggers described above. The P2 debugger [21] operates on the P2 [18] 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.

More broadly, many distributed monitoring systems can perform debugging functions, typically with a statistical bend [4, 28, 30]. 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`.

7 Conclusion and Future Work

`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 distributed debugging tasks that were, thus far, underserved by commercial or research debugging tools.

The road ahead is ripe for further innovation in distributed debugging. One direction of future work revolves around reducing 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 tool. Building a hybrid system that leverages the limited hardware watchpoints, yet gracefully degrades to slower methods, would also be rewarding.

Another high-priority feature 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.

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.

We plan to expand our proof-of-concept cluster-replay mechanism to make more efficient use of the cluster's resources. Our replay method was designed to ensure that each replay process is effectively independent and requires little external communication. Beyond cluster-parallelism, we are developing a version of `liblog` that allows replay in-situ on PlanetLab. This technique increases the cost of centralized scheduling but avoids the transfer of potentially large checkpoints and logs.

Further down the road, we want to improve 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?

Growing in scope, `Friday` motivates a renewed look at on-line distributed debugging as well. Our prior experience with P2 debugging [21] indicates that a higher-level specification of invariants, e.g., at "pseudo-code level," might be beneficially combined with system library-level implementation of those invariants, as exemplified by `Friday`, for high expressibility yet deep understanding of the low-level execution state of a system.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP*, 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[3] 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 *NSDI*, 2004.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, 2004.

[5] A. Chanda, K. Elmeleegy, A. Cox, and W. Zwaenepoel. Causeway: System Support for Controlling and Analyzing the Execution of Distributed Programs. In *HotOS*, 2005.

[6] M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *NSDI*, 2004.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. XTrace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[8] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *USENIX Annual Technical Conference*, 2006.

[9] T. L. Harris. Dependable Software Needs Pervasive Debugging (Extended Abstract). In *SIGOPS EW*, 2002.

[10] J. Hollingsworth and B. Miller. Dynamic Control of Performance Monitoring of Large Scale Parallel Systems. In *Super Computing*, 1993.

[11] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through VulnerabilitySpecific Predicates. In *SOSP*, 2005.

[12] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

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

[14] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[15] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *IPDPS*, 2000.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[17] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[18] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[19] P. Reynolds, J. L. Wiener, J. C. Mogul, M. A. Shah, C. Killian, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[20] Y. Saito. Jockey: A user-space library for record-replay debugging. In *International Symposium on Automated Analysis-Driven Debugging*, 2005.

[21] A. Singh, P. Maniatis, T. Roscoe, and P. Drushel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys*, 2006.

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

[23] S. M. Srinivashan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, 2004.

[24] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, 2002.

[25] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions of Networking*, 11(1):17–32, 2003.

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

[27] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Automatic On-line Failure Diagnosis at the End-User Site. In *HotDep*, 2006.

[28] R. van Renesse, K. P. Birman, D. Dumitriu, and W. Vogel. Scalable management and data mining using Astrolabe. In *IPTPS*, 2002.

[29] R. Wahbe. Efficient data breakpoints. In *ASPLOS*, 1992.

[30] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *SIGCOMM*, 2004.