`Friday`: Global Comprehension for Distributed Replay

Dennis Geels[†], Gautam Altekar[†], Petros Maniatis[φ], Timothy Roscoe[φ‡], Ion Stoica[†]
UC Berkeley[†], Intel Research Berkeley[φ], National ICT Australia, Sydney[‡]

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, as part of the debugging process.

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 microbenchmarks 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 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 system we present in this paper, is a first step towards realizing this vision. `Friday` takes on the challenge by (1) capturing the distributed execution of a system, (2) replaying the captured execution trace within a symbolic debugger in a single location, and (3) extending 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` [7], which can record the execution of a distributed application and then replay it deterministically and consistently. Replay takes place under control of a symbolic debugger, which in theory provides the developer with all the information needed to debug the application. But simple replay does not supply the global view of the system 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 debugging 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, 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 behavioral 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 [7]. In addition, the current implementation of Friday incurs a significant slowdown that limits its applicability to large data intensive applications.

Structure: We start with background on liblog in Section 2. Section 3 presents the design and implementation of Friday. We then present in Section 4 concrete usage examples in the context of two distributed applications: the Chord DHT [22], and a reliable communication toolkit for Byzantine network faults [23]. 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.

2 Background: liblog

Friday leverages liblog [7] to deterministically and consistently replay the execution of a distributed application. In this section, we give a brief overview of liblog and discuss its limitations.

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 process records its execution to a local log, with sufficient detail such that the same execution can be replayed later, from the beginning or from intermediate checkpoints, faithfully reproducing race conditions and non-deterministic failures. In this way, liblog ensures the programmer can 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. The library-based implementation has low overhead, as it requires neither extra context switches nor virtualization.

In addition, liblog ensures consistent group replaying, by maintaining Lamport clocks [14] during logging, which ensures that replay is causally consistent. That is, replay obeys the *happens-before* relation; for example, the reception of every message is replayed after the transmission of that message.

Finally, liblog can operate in an open environment, by allowing the liblog-instrumented application to communicate with applications that are not instrumented (e.g., DNS).

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 the next section, 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.

3 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 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

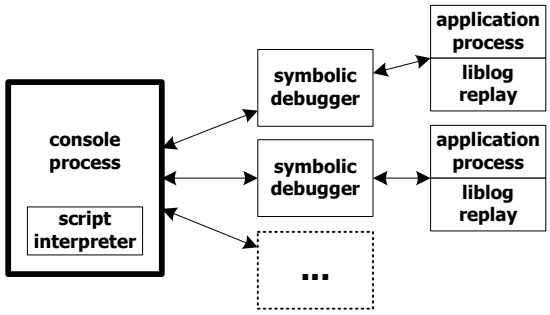

Figure 1: Overall architecture of `Friday`

extends the debugger's commands to handle distributed breakpoints and watchpoints, and to show status information about 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 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 3.2 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.

3.1.1 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 [25]. 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.

3.1.2 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 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. 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.

3.1.3 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 3.1.1. 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 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 `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.

3.2.1 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 IntroVirt [10]. 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.

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 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 4.2.1 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 for the duration of the function call. Unfortunately that precaution may be very costly (see Section 5). 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.

3.3 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 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 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, 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 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 [21], which has occupied networking and distributed research literature extensively. Then we turn to debugging `Tk`, a reliable communication toolkit [23], 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 `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 misbehavior of routing protocols that might result in route flaps, wormholes, or even blackholes.

4.1.1 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 2 provides an illustration.

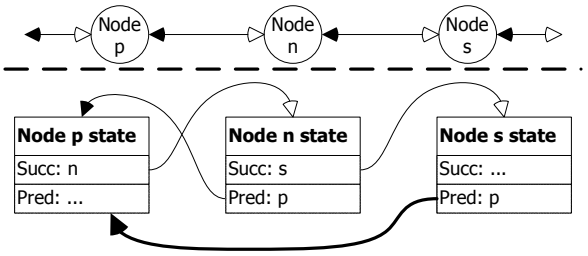

Figure 2: At the top, we show what node n hopes the ring topology looks like around it. At the bottom, we see the relevant state as stored by the involved nodes n, s and p. The thick routing entry from node s to its predecessor, which points to p instead of node n illustrates a possible inconsistency with node p and s's successor pointers.

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 3.2.

4.1.2 Ring Consistency Statistics

The techniques of Section 4.1.1 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 4.1.1, 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.

4.1.3 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.

4.1.4 Misdelivered Packets

Our last study moves beyond calculating the frequency and duration of routing inconsistencies to check an execution for actual occurrences of packets being misdelivered. 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 misdelivered 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.

4.2 A Reliable Communication Toolkit

In the second scenario, we investigate `Tk` [23], 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 pre-requisite. The predicate below constructs

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

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 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.

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

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 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 latency overhead required to watch data values and execute code on replayed process state. Table 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 afterwards.

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.

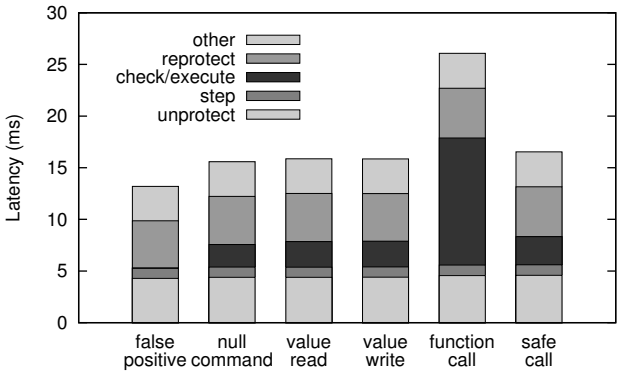

Figure 3: Latency breakdown for various watchpoint events.

Figure 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.

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 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

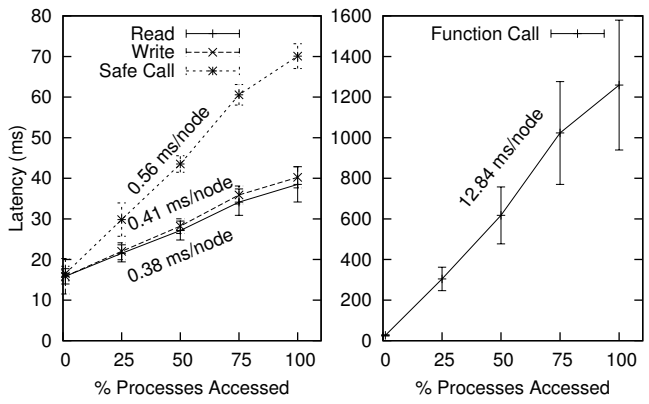

Figure 4: Microbenchmarks indicating latency and first standard deviation (y axis), as a function of the percentage of nodes involved in the operation (x axis). The population contains 62 nodes.

greater contention in the replay host's memory hierarchy due to the extra memory protection operations.

5.3 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 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.

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.

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.

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.

As expected, as the number of memory pages incurring false positives grows, replay slows down relative to the baseline. 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 perfectly linear, because some nodes were more active in the i3/Chord overlay 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 turn to the performance overheads incurred by `Friday` in the case studies from Section 4. Unlike the experiments up to this point, these case studies include realistic and useful commands. They exhibit a range of

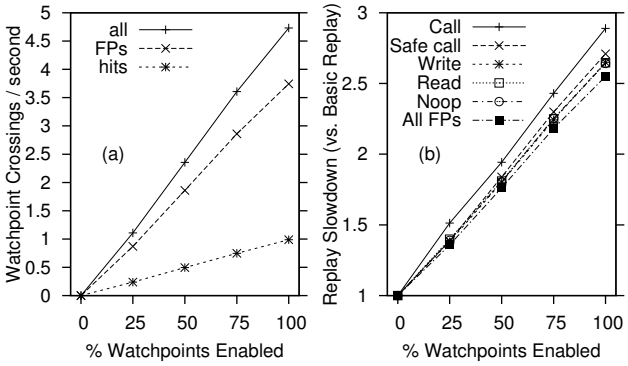

Figure 5: (a) Number of watchpoints crossed vs. percentage of nodes with watchpoints enabled (i3/Chord logs). Approximately linear. (b) Replay slowdown vs. percentage of nodes with watchpoints enabled, relative to baseline replay (i3/Chord logs).

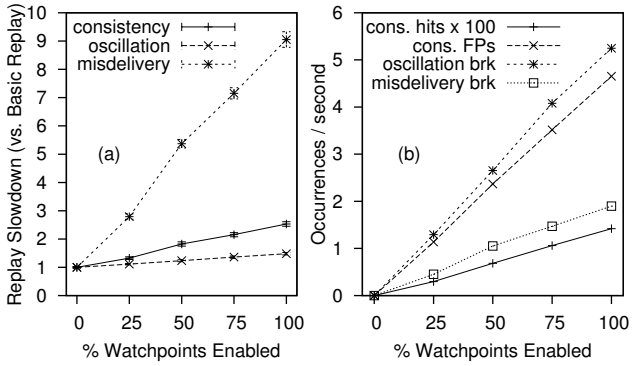

Figure 6: (a) Replay slowdown statistics for case study predicate performance vs. percentage of nodes with watchpoints enabled. (b) Watchpoint, breakpoint, and false positive rates vs. percentage of nodes with watchpoints/breakpoints enabled.

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

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.

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 4.1.2), State Oscillation (Section 4.1.3), and Misdelivered Packets (Section 4.1.4). Figure 6 plots the relative replay speed vs. baseline replay against the percentage of nodes on which the predicates are enabled. Table 3 summarizes the results. Results with the examples of 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 the predicate measured here and the micro-benchmarks in Figure 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 breakpoint 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 Misdelivered 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 6(a), is the slowest of the three predicates.

6 Related Work

In this section, we survey related literature focusing on the axes defining debugging with `Friday`: replay vs. log-based, and off-line vs. on-line.

6.1 Replay vs. Log-based Debugging

`Friday` utilizes library interposition to obtain a replayable deterministic trace of distributed executions. Much research has gone instead into replay debugging via virtualization, which can capture system effects below the system library level. Harris first made the case for pervasive, distributed debugging [8] through virtualization. Several projects have pursued that agenda

since [10, 11, 20], 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.

Closer to `Friday`, Jockey [17] and Flashback [20] use system call interposition, binary rewriting, and some operating system modifications to capture deterministic replayable traces, but only for a single node. DejaVu [13] targets distributed Java applications, but lacks the state manipulation facilities of `Friday`.

Moving away from replay debugging, many systems focus on extracting execution logs and then mining those logs for debugging purposes [1, 2, 5, 6, 9, 19, 26]. 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 causality (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) [12] and Pip [16]. 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 `Friday`, except for those relying on kernel-level interactions, which lie beyond our library tracing granularity. However, a bidirectional distributed backtracker implemented with `Friday` 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 [16] 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. They can be extracted automatically from the logs, or specified by the programmer and matched against the logs. Unlike Pip, `Friday` does not learn behaviors from a running system and has a much cruder, textual interface. However, `Friday` offers programmers greater flexibility in describing and capturing system behaviors 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 behavior 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 behaviors that go beyond pattern matching against logs. Such are the parametrized link symmetry checks of Section 4.1.1, where the identities of the pairs of processes that must satisfy the symmetry pattern are unknown until runtime and change as the system evolves.

6.2 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 [18] operates on the P2 [15] 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 [4, 24, 27]. 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 demanding 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 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?

In the long term, many promising directions exist in predicate expressiveness and functionality. For example, we have experimented with exposing messages as first-class objects in distributed predicates. This would allow the user to easily correlate transmission and reception, detect dropped messages, and trace message paths across a network. Growing in scope, our work with `Friday` motivates a renewed look at on-line distributed debugging as well. Our prior experience with P2 debugging [18] 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] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *USENIX Annual Technical Conference*, 2006.

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

[20] 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.

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

[22] 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.

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

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

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

[26] O. Wolfson, S. Sengupta, and Y. Yemini. Managing Communication Networks by Monitoring Databases. *IEEE Transactions on Software Engineering*, 17(9):944–953, 1991.

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