WiDS Checker: Combating Bugs in Distributed Systems

Xuezheng Liu
Microsoft Research Asia

Wei Lin
Microsoft Research Asia

Aimin Pan
Microsoft Research Asia

Zheng Zhang
Microsoft Research Asia

Abstract

Despite many efforts, the predominant practice of debugging a distributed system is still printf-based log mining, which is both tedious and error-prone. In this paper, we present WiDS Checker, a unified framework that can check distributed systems through both simulation and reproduced runs from real deployment. All instances of a distributed system can be executed within one simulation process, multiplexed properly to observe the "happens-before" relationship, thus accurately reveal full system state. A versatile script language allows a developer to refine system properties into straightforward assertions, which the checker inspects for violations. Combining these two components, we are able to check distributed properties that are otherwise impossible to check. We applied WiDS Checker over a suite of complex and real systems and found non-trivial bugs, including one in a previously proven Paxos specification. Our experience demonstrates the usefulness of the checker and allows us to gain insights beneficial to future research in this area.

1 Introduction

From large clusters in machine rooms to large-scale P2P networks, distributed systems are at the heart of today's Internet services. At the same time, it is well recognized that these systems are difficult to design, implement, and test. Their protocols involve complex interactions among a collection of networked machines, and must handle failures ranging from network problems to crashing nodes. Intricate sequences of events can trigger complex errors as a result of mishandled corner cases. The most challenging bugs are not the ones that will crash the system immediately, but the ones that corrupt certain design properties and drive the system to unexpected behaviors after long runs.

Yet the predominant practice in debugging distributed systems has remained unchanged over the years: manually inspecting logs dumped at different machines. Typically, developers embed printf statements at various implementation points, perform tests, somehow stitch the logs together, and then look for inconsistencies. However, log mining is both labor-intensive and fragile. Log events are enormous in number, making the inspection tedious and error-prone. Latent bugs often affect application properties that are themselves distributed across multiple nodes, and verifying them from local events can be very difficult. More important, logs reflect only incomplete information of an execution, sometimes insufficient to reveal bugs. Pip [23], for instance, logs the application behavior in terms of communication structures, timing and resource usage, and compares them against developer expectations. However, our experience shows that applications with correct message sequences can perform wrong things and mutate inner states because of buggy logic. Therefore, it is impossible to catch the existence of these subtle bugs only from communication logs, unless much more state is also logged.

It is a common experience that omitting a key logging point can miss a bug thus defeating the entire debugging exercise, yet adding it could substantially change subsequent runs. The non-determinism of distributed applications plus the limitations of log-based debugging makes such "Heisenbugs" a nightmare for developers. Building a time machine so that bugs can be deterministically replayed gets rid of the artifacts of using logs [8]. However, one still lacks a comprehensive framework to express correctness properties, catch violation points, and identify root causes.

We believe that a desired debugging tool for distributed applications needs to: 1) efficiently verify application properties, especially distributed ones; 2) provide fairly complete information about an execution, so that developers can observe arbitrary application states, rather than pre-defined logs; 3) reproduce the buggy runs deterministically and faithfully, and hence enable the cyclic debugging process.

In this paper, we address the above debugging requirements with a unified framework called WiDS Checker. This platform logs the actual execution of a distributed system implemented using the WiDS toolkit [15]. We can then apply predicate checking in a centralized simulator over a run that is either driven by testing scripts or is deterministically replayed by the logs. The checker outputs violation reports along with message traces, allowing us to perform "time-travel" inside the Visual Studio IDE to identify the root causes.

1.1 Our Results

Evaluating the effectiveness of our tool is a challenge. The research community, though acutely aware of the difficulties of debugging distributed systems, has not succeeded in producing a comprehensive set of benchmarks so that different debugging approaches can be quantitatively compared. While we believe the set of applications we have experimented with are representative, there is also no clear methodology of how to quantify the usefulness of the tool. To alleviate these problems, we resort to detailed discussions of case studies, hoping that other researchers working on similar systems can compare their experiences. Where possible, we also isolate the benefits coming from predicate checking from using log and replay only. Our tool is targeted at the scenario in which the system is debugged by those who developed it, and thus assumes that the bugs are hunted by those who are intimately familiar with the system. How to propagate the benefits to others who are not as versed in the system itself is an interesting research question.

We applied WiDS Checker to a suite of distributed systems, including both individual protocols and a complete, deployed system. Within a few weeks, we have found non-trivial bugs in all of them. We discovered both deadlock and livelock in the distributed lock service of Boxwood [19]. We checked an implementation of the Chord protocol [1] on top of Macedon [24] and found five bugs, three of them quite subtle. We checked our BitVault storage system [32], a deployed research prototype being incrementally improved for more than two years. We identified mishandling of race conditions that can cause loss of replicas, and incorrect assumptions of transient failures. The most interesting experience was checking our Paxos [13] implementation, revealing a bug in a well-studied specification itself [21]. Most of these bugs were not discovered before; All our findings are confirmed by the authors or developers of the corresponding systems.

We also learned some lessons. Some bugs have deep paths and appear only at fairly large scale. They cannot be identified when the system is downscaled, thus calling for more efficient handling of the state explosion problem when a model checker is applied to check actual implementation. Most of the bug cases we found have correct communication structure and messages. Therefore, previous work that relies on verifying event ordering is unable to detect these bugs, and is arguably more effective for performance bugs.

1.2 Paper Roadmap

The rest of the paper is organized as follows. Section 2 gives an overview of the checker architecture. Section 3 provides implementation details. Section 4 presents our results, including our debugging experience and lessons learned. Section 5 contains related work and we conclude with future work in Section 6.

2 Methodology and Architecture Overview

2.1 Replay-Based Predicate Checking

The WiDS[1] checker is built on top of the WiDS toolkit, which defines a set of APIs that developers use to write generic distributed applications (details see Section 3.1). Without modification, a WiDS-based implementation can be simulated in a single simulation process, simulated on a cluster-based parallel simulation engine, or deployed and run in real environment. This is made possible by linking the application binary to three different runtime libraries (simulation, parallel simulation and deployment) that implement the same API interface. WiDS was originally developed for large-scale P2P applications; its parallel simulation engine [14] has simulated up to 2 million instances of a product-strength P2P protocol, and revealed bugs that only occur at scale [29]. With a set of basic fault injection utilities, WiDS allows a system to be well tested inside its simulation-based testing framework before its release to deployment. WiDS is available with full source code in [3].

However, it is impossible to root out all bugs inside the simulator. The deployed environment can embody different system assumptions, and the full state is unfolded unpredictably. Tracking bugs becomes extremely challenging, especially for the ones causing violation of system properties that are themselves distributed. When debugging non-distributed software and stand-alone components, developers generally check memory states against design-specified correctness properties at runtime using invariant predicates (e.g., *assert()* in C++). This dynamic predicate checking technique is proven of great help to debugging; many advanced program-checking tools are effective for finding domain-specific bugs (e.g., race conditions [25, 31] and memory leaks [22, 10]) based on the same principle. However, this benefit does not extend to distributed systems for two reasons. First, dis-

tributed properties reside on multiple machines and cannot be directly evaluated at one place without significant runtime perturbations. Second, even if we can catch a violation, the cyclic debugging process is broken because non-determinism across runs makes it next to impossible to repeat the same code path that leads to the bug.

To address this problem and to provide similar checking capabilities to distributed systems, we propose a **replay-based predicate checking** approach that allows the execution of the entire system to be replayed afterwards within a single machine, and at the same time checks node states during the replayed execution against user-defined predicates. Under modest scale, this solves both problems outlined above.

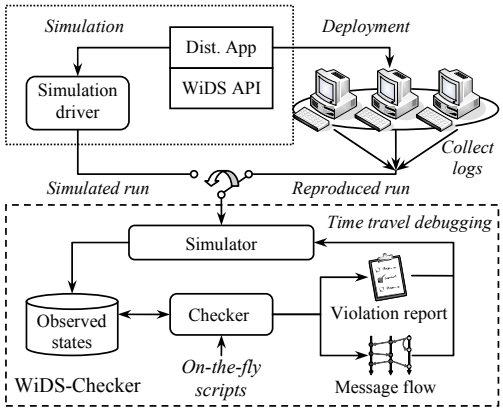

Figure 1: Components of WiDS Checker. The upper-left box was employed as debugging support in WiDS before the Checker was developed.

2.2 Checking Methodology

User-defined predicates are checked at *event* granularity. An event can be an expiration of a timer, receiving a message from another node, or scheduling and synchronization events (e.g., resuming/yielding a thread and acquiring/releasing a lock) specific for thread programming. WiDS interprets an execution of a single node or the entire distributed system as a sequence of events, which are dispatched to corresponding handling routines. During the replay, previous executed events from all nodes are re-dispatched, ordered according to the "happens-before" relationship [12]. This way the entire system is replayed in the simulator while preserving causality.

Each time an event is dispatched, the checker evaluates predicates and reports violations for the current step in replay. The choice of using event boundaries for predicate checking is due to a number of factors. First, the event model is the basis of many protocol specifications, especially the ones based on I/O-automata [18, 17]. A system built with the event model can be regarded as a set of state machines in which each event causes a state transition executed as an atomic step. Distributed properties thus change at event boundaries. Second, many widely adopted implementation models can be distilled into such a model. We have used the WiDS toolkit [15] to build large and deployed systems as well as many protocols; network middle-layers such as Macedon [24] and new overlay models such as P2 [16] can be regarded as event-based platforms as well. We believe event granularity is not only efficient, but also sufficient.

2.3 Architecture

Figure 1 shows the architecture of WiDS Checker.

Reproducing real runs. When the system is running across multiple machines, the runtime logs all non-deterministic events, including messages received from network, data read from files, thread scheduling decisions and many environmental system calls. The executable binary is then rerun inside the simulator, and all non-deterministic events are fed from the logs. We use Lamport's logical clock [12] to decide the replay order of events from different nodes, so that the "happens-before" relationship is preserved. Therefore, inside the simulator, we reconstruct the exact state of all instances as they are run in the real environment.

Checking user-defined predicates. We designed a versatile scripting language to specify system states being observed and the predicates for invariants and correctness. After each step of event-handling, the observed states are retrieved from the replayed instance in the simulator, and refreshed in a database. Then, the checker evaluates predicates based on the current states from all replayed instances and reports violations. Because predicates generally reflect design properties, they are easy to reason and write.

Screening out false alarms with auxiliary information. Unlike safety properties, liveness properties are only guaranteed to be true eventually. This poses a serious problem when checking liveness properties, since many violations can be false alarms. To screen out false alarms, we enable user-defined auxiliary information to be calculated and output along with each violation point. When a violation occurs, the auxiliary information is typically used to produce stability measures based on user-provided heuristics.

Tracing root causes using a visualization tool. In addition to the violation report, we generate a message flow graph based on event traces. All these facilities are integrated into the Visual Studio Integrated Development Environment (IDE). Thus, a developer can "time-travel" to violation points, and then trace backwards while inspecting the full state to identify root causes.

The first two components make it possible to check distributed properties, whereas the last two are critical features for a productive debugging experience. Note that developers can incrementally refine predicates and re-evaluate them on the same reproduced execution. In other words, by means of replay, cyclic debugging is re-enabled.

3 Design and Implementation

WiDS Checker depends critically on the replay functionality, which is done at the API level. In this section, we will first briefly describe these APIs. We then explain the replay facility and the checker implementation. This section concludes with a discussion of the known limitations. WiDS has close to 20K lines of code. The replay and checker components add 4K and 7K lines, respectively.

3.1 Programming with WiDS

Table 1 lists the class of WiDS APIs with some examples. The WiDS APIs are mostly member functions of the *WiDSObject* class, which typically implements one node instance of a distributed system. The WiDS runtime maintains an event queue to buffer pending events and dispatches them to corresponding handling routines (e.g., *OnMsgHandler()*). Beside this event-driven model, WiDS also supports multi-threaded programming with its thread and synchronization APIs. The context switching of WiDS threads is encapsulated as events in the event queue. We use non-preemptive scheduling in which the scheduling points are WiDS API and blocking system calls, similar to many user-level thread implementations. The fault-injection utilities include dropping or changing the latency of messages, and killing and restarting WiDS objects.

Macedon over WiDS. It's not easy to write a distributed application with message-passing and event-driven model. Defining better language support is an active research area [16, 24]. WiDS can be used as the low-level mechanism to implement such languages, and thus make its full functionality available to their applications. As an experiment, we ported Macedon [24], an infrastructure to specify and generate implementation of overlay protocols. Macedon has been used to reimplement many complex overlay protocols with a simple domain language and conduct performance comparisons. Porting to the WiDS API is simplified because both Macedon and WiDS provide a set of APIs working in an event-driven manner, and programming entities such as messages and timers exist in both platforms. An overlay protocol expressed with Macedon is composed of a .mac file, which our parser takes as input to generate a set of WiDS implementation files. Many overlay specific functionalities provided by the Macedon library are replicated in a WiDS-based library. Finally, all these files are linked to generate an executable, which, with a proper driver, can be both simulated and deployed. Supporting Macedon is accomplished with about 4K lines of code.

3.2 Enabling replay

The primary goal of the deterministic replay is to reproduce the exact application memory states inside the simulator. To achieve this, we need to log all non-deterministic inputs to the application and feed them to the replay simulator.

Logging. The WiDS runtime logs the following two classes of non-determinism: The first is internal to WiDS. We record all the WiDS events, the WiDS thread schedule decisions and the incoming message content. The second class are OS system calls, including reading from files, returned memory addresses for allocation and deallocation in heap, and miscellaneous others such as system time and random number generation. In Windows NT, each API call is redirected by the linker to the import address table (IAT), from which another jump is taken to reach the real API function. We changed the address in the IAT, so the second jump will lead to the appropriate logging wrapper, which will log the return results after the real API is executed. Furthermore, to enable consistent group replay [8], we embed a Lamport Clock [12] in each out-going message's header to perserve the "happens-before" relation during the replay. Table 1 describes logging and replay mechanisms for API calls.

In addition, we use a lightweight compressor to effectively reduce the log size. As we report in Section 4.5, the computation overhead for logging is small in the tests we performed.

Checkpoint. We use checkpoints to avoid over-committing storage overhead from logging and to support partial replay during replay. A checkpoint includes the snapshot of memory of the WiDS process and the running context for user-level threads and sockets as well as buffered events in the event queue.

Replay. Replaying can start from either the beginning or a checkpoint. Note that checking predicates requires all instances to be replayed with causality among them preserved. Therefore, during the replay, events from different instances are collected from logs, sequentialized into a total execution order based on the Lamport Clock, and re-executed one-by-one in the simulator.

To improve replay performance and scalability, we use only *one* simulation process to replay all instances, and use file-mapping to deal with the memory switch between different instances. The state of an instance is

WiDS API set		
Category	API example	Logging and replay mechanism
Event-driven program	SetTimer, KillTimer, OnTimerExpire	Log the event type and the sequence; redo the same events in replay
Message communication	PostMsg, PostReliableMsg, OnMsgHandler	Embed Lamport Clock to maintain causal order, log incoming message contents. Replay with correct partial order, feed message content.
Multi-threaded program	CreateThread, JoinThread, KillThread, YieldThread, Lock, Unlock	Log the schedule decision and the thread context. Ensure the same schedule decision and the same context during replay
Socket APIs for network virtualization	WiDSSocket, WiDSListen, WiDSAccept, WiDSConnect, WiDSSend, WiDSRecv	Log the operation along with all received data. Feed the received data from log during replay. Sending operations become no-ops in replay
Fault injection and message delay	ActivateNode, DeActivateNode, SetNetworkModel, OnCalculateDelay	Log the operation of activation/deactivation, and redo the operation in replay
Operation system APIs (for Windows)		
File system	CreateFile, OpenFile, ReadFile, WriteFile, CloseHandle, SetFilePointer	Log the operation along with all input data. Feed the input data from log during replay. Write operations become no-ops in replay
Memory management	VirtualAlloc/Free, HeapAlloc/Free	Ensure identical memory layout in replay.
Miscellaneous	GetSystemTimeAsFileTime, GetLastError	Log the return value, and feed the same value in replay

Table 1: WiDS API set and OS APIs with logging and replay mechanisms

stored in a memory mapped file, and is mapped into the process memory space on-demand. For example, to switch the replayed instance from A to B, we only update the entries in the page table of the simulation process to the base address of the mapped memory of B. Starting a process for each replayed instance and switching the processes would require local process communication (LPC) that is typically tens of times slower than function calls. Therefore, our approach has significant advantages. When the aggregated working set of all replayed instances fit into physical memory, the only overhead in switching instance is the update to the page table. It also avoids redundant memory usage caused by process context and executable binary for each instance.

The largest per-instance working set in our experiments is about 20MB, meaning that more than 40 instances can be replayed in 1GB physical memory without going to the disk. When the aggregated working set exceeds the physical memory, depending on the computation density of the replayed instance, we have observed 10 to 100 times slowdown due to disk swapping. The checker itself maintains a copy of the states being checked, and that varies across applications. Ultimately, the scalability is bound by the disk size and acceptable replay speed.

3.3 Checker

Deterministic replay that properly preserves causality has enabled the reconstruction of memory states of a distributed system. The next step is to write predicate statements to catch the violation points of correctness properties. We define a simple scripting language for specifying predicates, so that developers can easily specify the structure of the investigated states, retrieve them from the memory of the instances, and evaluate properties from these states.

As mentioned before, checking predicates is invoked at event boundaries. Each time an event is re-executed in a replayed instance, the checker examines the state changes in the instances, and re-evaluates the affected predicates. The states being checked are copies kept in a separate database. The checker refreshes these states in the database from the replayed instance and evaluates predicates based on the state copies of all instances. Therefore, checking predicates is decoupled from memory layout of the instances, and we do not require all instances to reside in memory simultaneously for evaluating global properties. This makes replay and the checker more scalable. Furthermore, maintaining state copies separately allows us to keep past versions of states if needed, which is useful for evaluating certain properties (see Section 4.1).

In this section, we explain the checker implementation, including the necessary reflection facilities that make memory states in C++ objects observable by the checker, state maintenance and predicate evaluation, and the auxiliary information associated with violations that deal with false alarms.

3.3.1 Observing memory states

For programming languages such as Java and C# that support runtime reflection, the type system and user-defined data structures are observable during the runtime. Unfortunately, this is not the case for C++, which is what WiDS uses. To check application states, we need to record the memory address of each allocated C++ object with type information during its lifetime. We use the Phoenix compiler infrastructure [2] to analyze the executable and inject our code to track class types and object

addresses. Phoenix provides compiler-independent intermediate representation of binary code, from which we are able to list basic blocks, function calls, and the symbol table that contains all type definitions. We then inject our logging function to function calls of constructors and deconstructors of the classes. The logging function will dump the timestamp, type of operation (i.e., construction or deconstruction) along with the address of object and type information. This information is used by the checker to inspect memory states. The following assembly codes show an example of a constructor after code injection. The lines beginning with "*" are injected code. They call our injected logging function *onConstruct* with the index number of this class found in symbol table. We perform similar code injection for object deconstruction. As a result, at each step of the replay, the checker is capable of enumerating pointers of all objects of a certain class, and further reading their memory fields according to the symbol table. The runtime overhead is negligible since the actions are only triggered at object allocation and deallocation time.

```
$L1: (refs=0)  START MyClass::MyClass
MyClass::MyClass: (refs=1)
this = ENTERFUNC
 * [ESP], {ESP} = push 0x17 //index number for MyClass
 * call _imp__onConstruct@4, $out[ESP] //call log func
[ESP], {ESP} = push EBP
EBP = mov ESP [ESP],
{ESP} = push ECX
... // other code in original constructor
ESP = mov EBP EBP,
{ESP} = pop [ESP]
{ESP} = ret {ESP}, MyClass::MyClass
MyClass::MyClass: (refs=1) Offset: 32(0x0020)
EXITFUNC
$L2: (refs=0) END
```

This code injection is carried out in a fully automated way. In addition, we provide some APIs that allow developers to explicitly calculate and expose states of an instance in the source code.

3.3.2 Defining states and evaluating predicates

A script for predicate evaluation is composed of three parts: declaration of tables, declaration of internal variables, and the predicates. Figure 2 shows the script we used for checking the Chord protocol [28] implemented on Macedon (see Section 4.4 for details).

The first section (starting with "**declare_table**") instructs the checker to observe objects of some classes and refresh the states of certain member fields into the database. Each row of the table corresponds to one object in the system, and table columns correspond to member fields of the object. Each table has two built-in columns *instance_id* and *memory_addr*, corresponding to the replayed instance and the object's memory address, respectively. The declaration gives the user shorthand notations to name the table and the states. A table stores global states from all instances, e.g., the table *Node* here maintains the investigated states of all the Chord nodes in the system. Sometimes it is useful to keep a short history of a state for checking. We provide an optional **keep_version(**N**)** after a column declaration to declare that the recent N versions of the state should be kept in the table.

```
                    # define data table
declare_table Node from CChord
  column id as m_nodeid
  column pred as m_predecessor
  column succ as m_successor
  column status as m_status
end_declare
                    # define checker variables
declare_derived last_churn_time
begin_python
  for x in Node :
      if (x.status == 0          # status "0" means joining
        or Runtime.msd_id== 108);      # MSG_FAIL_NOTIFY
        return Runtime.current_time;
  return last_churn_time;
end_python

declare_derived stabilized
begin_python
  retval = (Runtime.current_time - last_churn_time) / 10.0;
      if (retval < 1) : return retval;
  return 1;
end_python
                    # define predicates
predicate RingConsistency auxiliary stabilized{
  forall x in Node, exist y in Node,
    x.pred==y.id and y.succ == x.id
}
```

Figure 2: An example of check scripts for chord. The auxiliary information *Stabilized* is reset to 0 when joins or failures occur; otherwise it gradually grows to 1.

The second section allows users to define variables internal to the checker with the keyword **declare_derived**. These variables can also have histories, again using **keep_version(**N**)**. Between **begin_python** and **end_python** are python snippets to calculate the value of a named variable. The python snippet has read access to values of all prior declarations (i.e., data tables and internal variables), using the declared names. Data tables are regarded as enumerable python containers, indexed by a (*instance_id*, *memory_addr*) pair.

The last section uses the keyword "**predicate**" to specify correctness properties based on all declared states and varables, which are evaluated after refreshing tables and after the evaluation of internal variables. Each predicate is a Boolean expression. We support the set of common logical operators, e.g., **and**, **or**, **imply**. We also support two quantifiers, **forall** and **exist**, which specify the extent of validity of a predicate when dealing with tables. These built-in operators make it easy to specify many useful invariants. In Figure 2, the predicate states that the ring should be well formed: if node x believes node y to be its predecessor, then y must regard x as its successor. It is

a critical property for the stabilization of Chord topology.

After each step of the replay, the checker does the following. First, it enumerates all objects of classes defined in data tables in the memory of a replayed instance. It uses the type information and memory address provided by the log to refresh the table, inserting or deleting rows, and updating the columns accordingly. After updating tables, the checker also knows which declared states have changed, and it only re-evaluats all the affected derived values and predicates. When some predicates are evaluated as "false," the checker outputs the violation into a *violation report*, which contains the violated predicates, Lamport Clock value for each violation, and the auxiliary information defined in the script (discussed shortly).

Sometimes it is useful to replay and check a segment of execution, rather than to do it from the beginning. The problem here is how to reconstruct the states maintained by checker scripts when the checkpoint is loaded. To solve this problem, we support checkpoints in replay runs, which store both replay context and the tables and variables used by predicate scripts. These replay checkpoints can be used seamlessly for later checking. To start checking with an intermediate checkpoint from testing runs, the developers have to provide additional scripts to setup the states required by the script from the memory of instances in the checkpoint. Otherwise, there might be incorrect predicate evaluations caused by checkpoints.

3.3.3 Auxiliary information for violations

For safety properties that must hold all the time, every violation reveals a bug case. In contrast, liveness properties only guarantee to be true eventually, so a violation of liveness properties is not necessarily a bug case. For example, many overlay network systems employ self-stabilizing protocols to deal with churns, therefore most of their topology-related properties are liveness ones. As a result, checking liveness properties could generate a large number of false alarms that overwhelm the real violations. Adding a time bound to liveness properties is not always a desirable solution, because usually it's hard to derive an appropriate time bound.

To solve the problem, we enable users to attach auxiliary information to the predicates. The auxiliary information is a user-defined varable calculated along with the predicate, and it is only output when the predicate is violated. Developers could used the information to help screen out false alarms or prioritize violations. For liveness properties, an appropriate usage for auxiliary information is to output a measurement of a stabilization condition. For example, in Figure 2 we associate the eventual ring consistency property with an auxiliray variable *Stabilized* ranging from 0 to 1, as a measure of stabilization that shows the "confidence" of the violation.

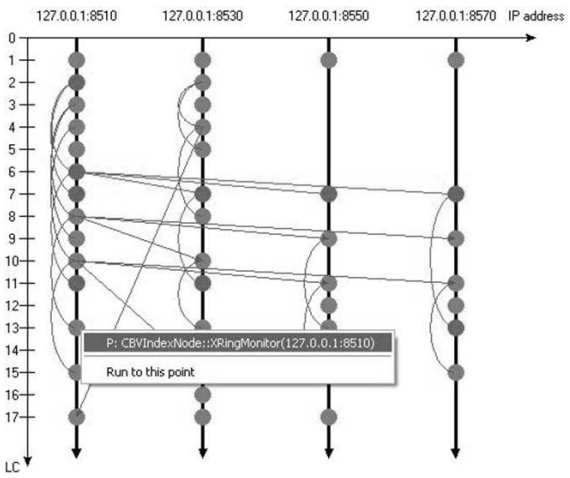

Figure 3: A screenshot of message flow graph. The vertical lines represent the histories of different instance, arcs denote messages across instances, and the nodes correspond to event handlings. Arcs with two ends on the same vertical lines are either timer events or messages sending to the instance itself.

We also maintain some built-in system parameters in the checker, e.g., the current time in the node, the current message type, and statistics of recent messages of each type. These parameters can be directly accessed in the scripts, and are useful in stabilization measurement. Our evaluation section contains more examples of using the auxiliary information.

3.4 Visualization tools

To pinpoint the root cause of a bug, a user often needs to trace back in time from a violation point. In addition to our replay facility, we provide the message flow graph generated based on message trace (Figure 3) to make this process easier. It is a common practice in our experience to perform time travel following the message flow, replay to a selected event point and inspect memory state of the replayed instance. We find that the visualization helps us understand system behaviors as well as the root cause after catching violations.

4 Evaluation

In this section, we report our experience of finding bugs using WiDS Checker over a comprehensive set of real systems. Table 2 gives a summary of the results. The checking scripts to reveal the bugs are short and easy to write from system properties. For each of these systems, we give sufficient descriptions of their protocols and explain the bugs we found and how they are discovered. We also summarize the benefits from WiDS Checker at the

end of each case study. We will discuss relevant performance numbers and conclude with lessons that we have learned.

Application	# of lines	# of bugs	Lines of script
Paxos	588	2	29
Lock server	2,439	2	33
BitVault	17,582	3	181
Macedon-chord	2,468	5	86

Table 2: benchmark and result summary

4.1 Paxos

Paxos [13] provides a fault-tolerant distributed consensus service. It assumes an asynchronous communication model where messages can be lost and delayed arbitrarily. Processes can operate at arbitrary speed, may fail-stop and restart. Our implementation strictly follows the I/O-automata specification in [21], in which there are two types of processes: *leaders* and *agents*.[2] Leaders propose values to the agents round-by-round, one value for each round. Agents can accept or reject each received proposal independently, and a decision is made if the majority of the agents agree on the same round/value pair. The important safety property in Paxos is *agreement*: once a decision is made, the decided value will stay unchanged.

The main idea of the algorithm is to have leaders work cooperatively: endorse the latest accepted round/value pair in the system as they currently understand it, or propose their own if there is none. The following is an informal description. The protocol works in two phases, learning (steps 1 and 2) and submitting (steps 3 and 4):

1. A leader starts a new round by sending a *Collect* request to all agents with its round number n. The round number must be unique and increasing.

2. An agent responds to a *Collect* request with its latest accepted round/value pair (if any), if and only if the round number in the request is greater than any *Collect* requests to which it has already responded.

3. Once the leader has gathered responses from a majority of agents, it sends a *Begin* request to all agents to submit the value for round n. The submitted value is the previously accepted value in the highest-numbered round among the responses in Step 2, or the leader's own value if there is none.

4. An agent that receives a *Begin* message of round number n accepts the value (and accordingly updates its latest accepted value and round number), unless it has already responded to a *Collect* request with a higher round number than n.

```
declare_table Agent from PaxosAgent
column LastRound as m_last_accepted_round keep_version(2)
column LastValue as m_last_accepted_value
end_declare

declare_derived decision_value keep_version(2)
begin_python
   rnd_cnt = {};        # map from round number to count
   rnd_val = {};        # map from round number to submitted value
   for x in Agents :    # endorse an existing decision
      if x.LastRound in rnd_cnt.keys() :
            rnd_cnt[x.LastRound] += 1;
      else :            # record a new decision
            rnd_cnt[x.LastRound] = 1;
            rnd_val[x.LastRound] = x.LastValue;
   for round, count in rnd_cnt.items() :      # report the decision
      if count >= 4 : return rnd_val[round]
   return "NIL"
end_python
                  # decision shall never change
predicate GlobalConsistency{
   decision_value.version(-1) == decision_value.version(0)
   or decision_value.version(-1) = "NIL"
}
            # accepted round number should always increase
predicate AcceptedRoundIncreasing {
   (forall x in Agents)
   x.LastRound.version(-1) <= x.LastRound.version(0)
}
```

Figure 4: Checker script for Paxos

In our implementation, leaders choose monotonically increasing round numbers from disjoint sets of integers. An agent broadcasts to all leaders when it accepts a value so that leaders can learn the decision. Each leader sleeps for a random period before starting a round, until it learns a decision is made. The test was carried out using simulation with seven processes acting as both leaders and agents. To simulate message loss, we randomly dropped messages with a probability of 0.3.

We wrote two predicates that are directly translated from safety properties in the specification (see Figure 4). The Python snippet of decision_value calculates the value accepted by the majority of agents. The first predicate, *GlobalConsistency*, specifies the agreement property: all decision values should be identical. It is a distributed property derived from all agents. The second predicate, *AcceptedRoundIncreasing*, states a local property in the specification that the newly accepted round number increases monotonically in an agent.

The checker caught an implementation bug in Step 3 with the *GlobalConsistency* predicate, finding that after an agent accepted a new *Begin* message, the decision value changed to a different one. The root cause is that, in Step 3, after the leader gathers responses from a majority of agents, it sends a *Begin* request with the submitted value from the *latest* received agent response, instead of the *highest-numbered* round among all responses. The predicate breaks immediately after the *Begin* message is handled that changes the decision. Tracing back one single step in the message flow and replaying the code for Step 3 allows us to immediately identify the root cause.

The second bug is far more subtle. We found that with very small probability, the accepted round number in an agent may decrease, which violates the second predicate. We ran our test several hundred times, each of them having thousands of messages, but only caught one violation. Using replay to trace back from the violation point, we identified that the bug was not in the implementation, but in the specification itself. In Step 3 the *Begin* requests are sent to all agents; under message loss it is possible for an agent to receive a *Begin* request without the pairing *Collect* request of the round. This means that the agent can have its accepted round number greater than its round number responding to the *Collect* request (the two are kept in separate counters). Thus, based on Step 2 the agent may in the future respond to (and in Step 4 accept a value from) some smaller-numbered rounds, decreasing the accepted round number. With a small probability, the bug can actually break the ultimate agreement property. (We have constructed such a sequence of events starting with our violating trace). However, this is a corner case and the protocol is robust enough that it never happened in our test. After researching the specification carefully, we also understood where the original proof went wrong. This bug and our fix for the specification is confirmed by one of the authors.

Study on effectiveness. The checker gives precise bug points from thousands of events in both bug cases. After that, identifying the root cause becomes simply tracing back one or a few messages in replay. Replay (or only logging, if we dump states of Paxos nodes to the log) is necessary for understanding root causes, however, as *GlobalConsistency* is a distributed property that cannot be directly verified from logs. Without the checker a developer has to go through a huge number of events in replay trace and check the correctness property manually, which is very inefficient. The second predicate, though a local one, proves the usefulness of rigorous predicate checking for distributed systems. Without this predicate, we would miss the specification bug altogether.

4.2 Lock Server in Boxwood

The Boxwood [19] storage system implements a fault-tolerant distributed lock service that allows clients to acquire multiple-reader single-writer locks. A lock can be in one of the states: *unlocked, shared, exclusive*, and multiple threads from one or more clients acquire or release locks by communicating to a server. The server uses a *pendingqueue* to record locks for which transactions are still in flight.

The system was written in C#. In order to check it, we ported it to C++ and the WiDS APIs in about two weeks; most of the effort was spent on the language difference (e.g., the pointers). In the resulting code, almost every line can be mapped to the original code. This enables us to map bugs found by WiDS Checker to the original C# code. At first we checked the safety property that if multiple clients simultaneously own the same lock, they must be all in shared mode. However, during the experiment we did not find any violations in both simulated and reproduced runs. Next, we focused on deadlock and livelock checking on a larger scale and found both of them.

```
declare_derived pendingsize keep_version(2)
begin_python
   return Server[0].pendingsize
end_python

declare_derived last_change_time
begin_python
    if (pendingsize.version(0) != pendingsize.version(-1)) :
    return RuntimeInfo.current_time;
    return last_change_time;       # remain unchanged
end_python

declare_derived enough_time {
begin_python
   return (Runtime.current_time – last_changed_time) / threshold
end_python

predicate PendingQueueIsEmpty auxiliary enough_time {
    pendingsize.version(0) == 0
}
```

Figure 5: Checker script for lock server

We ran the test inside a simulator and used 20 clients, one server and four locks. Each client has five threads that keep randomly requesting one of the locks and then releasing it after a random wait. Rather than writing sophisticated predicates to look for cycles of dependencies, we used a simple rule that if the protocol is deadlock-free, the *pendingqueue* should *eventually* be empty. The predicate is just that: return true if the *pendingqueue* is empty. Clearly there could be many false alarms. We attached an auxiliary output that computes how long the predicate remained broken since the last time the queue size changed (Figure 5).

Livelock. The livelock we discovered involves one lock and two clients. Based on replay and the message flow graph, we isolated the following bug scenario. Client A holds lock l in shared mode, and client B requests l in exclusive mode. The server inserts l to the $pendinglock$ queue and sends a revoke message to A. While this message is on its way, another thread in A requests to upgrade l to exclusive. According to the protocol, A needs to release l first before acquiring it again with the desired mode. A does so by first setting a $releasepending$ flag and then sending the server a release message.

According to the protocol, the server always denies a lock release request if a lock is in the $pendinglock$ queue. The revoke message arrives at A and spawns a revoking thread, which in turn was blocked because the lock is being released (i.e., the $releasepending$ flag is

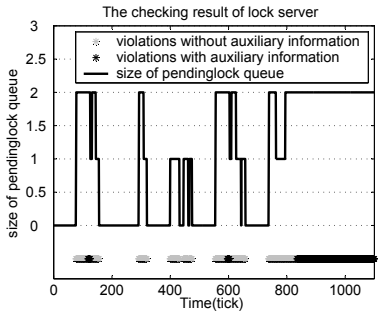

Figure 6: Violations reported in the livelock case. The stripe at the bottom contains all the violations; the dark ones are those with *enough_time* above the threshold.

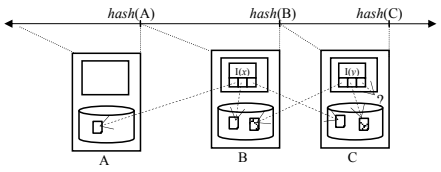

Figure 7: BitVault ID space and index structure. B and C are owners of object x and y, respectively. Object x has three replicas, whereas object y has one dangling pointer.

set). When A finds its release is unsuccessful it resets the $releasepending$ flag and retries. However, the retry code is in a tight loop, and thus the release request is sent again and the $releasepending$ flag is set. As a result the revoke thread wakes up only to find that it is to be blocked again. This cycle continues onwards and repeats as a livelock. It is livelock in the sense that there is a small possibility that the blocked revoking thread can cut in after the release response is handled but before the next release retry occurs.

Figure 6 visualizes the violation reports in a run that discovered the livelock. There were altogether five rounds of competition and the bug appears at the final one. We use *enough_time* to screen out false alarm in violations. After screening, many false alarms are eliminated. Auxiliary output helped us to prioritize inspection of violations; otherwise the violations will be too large in number to inspect.

Deadlock. The deadlock case is more sophisticated. The client implementation has an optimization of using a socket pool for outstanding requests, and a thread will be blocked if it cannot find a socket to use. Because the socket pool is shared among threads, it creates extra dependencies and induces a deadlock.

We configured the socket pool to use only one socket. The resulting deadlock case involves four clients and two locks. The initial state is that A has shared lock l_1, and C has exclusive lock l_2. Next, B and D request exclusive mode on l_2 and l_1, respectively. After a convoluted sequence of events, including threads on A and B attempting to upgrade and release their locks, the system enters a deadlock state. Based on replay and message flow graph, we find the deadlock cycle, which consists of a lock acquiring thread whose request is blocked on the server because the lock is in the $pendinglock$ queue, a revoking thread blocked by the lock's $releasepending$ flag, and a release thread blocked by the unavailability of socket.

Study on effectiveness. Unlike the Paxos case in which the checker directly locates the bug point, here we do not have effective predicates to reveal deadlock/livelock; the predicate based on $pendingsize$ only provides hints for the starting point of the replay. So, the checker is more like an advanced "conditional breakpoint" on the replay trace, and mining the root cause heavily depends on the replay facility and the message flow visualization tool.

4.3 BitVault Storage System

BitVault [32] is a scalable and content-addressable storage system built by ourselves. The system is composed of more than 10 interdependent protocols, some of which incrementally developed over a stretch of two years. BitVault achieves high reliability with object replication and fast parallel repair for lost replicas. Self-managing routines in BitVault ensure that eventually every existing object in the system has its specified replication degree.

To understand the bugs, it is necessary to describe BitVault's internal indexing structure and repair protocols. BitVault uses a DHT to index the replicas of each object. Each object has a unique 160bit hash ID, and the entire ID space is uniformly partitioned by the hashes of the participating nodes, each node owning a particular ID range. An object's replicas can reside on any node, while its index (which records locations of the object's replicas) is placed on the *owner* node that owns the ID of the object. The design of the index enables flexible control of replica placement. Figure 7 shows the index schema. Mapping between ID space and nodes is achieved by a decentralized weakly-consistent membership service, in which every node uses heart-beat messages to detect node failures within its ID neighborhood, and maintains a local membership view of all nodes in the system. According to the membership view, the owner of an index can detect replica loss and then trigger the creation of an another replica. A replica also republishes itself to the new owner node of the index when it detects the change of the owner node.

BitVault has the following correctness properties derived from its indexing scheme: 1) correct index ownership, i.e., for each object, eventually the owner node

holds the index; 2) complete reference, i.e., when the system stabilizes, there should be neither dangling replica references nor orphan replicas; 3) correct replica degree, i.e., in a stabilized system every object has exactly $degree$ (say 3) replicas. Because a node's membership view is guaranteed to converge *eventually*, all these must be considered liveness properties. We use these properties to check BitVault, and associate them with a stabilization measure based on membership change events. All experiments are conducted over an 8-node configuration in a production run, and we found the three bugs with the checker. Due to space limitations, we only explain two of them.

Replica loss due to protocol races. BitVault passes intensive testing before we added a routine that balances load between nodes by moving replicas. After adding the load balancing routine, we observed an occasional replica loss from our monitor GUI. Before we developed WiDS Checker, we did not actually know the root cause because the bug case was buried in irrelevant log events.

The checker catches a violation of the replica degree predicate saying that an object's replica number should be no less than 3. From the violation point we trace back a few messages and quickly found how the replica number for this object goes from 3 to 2. BitVault has a "remove-extra-replicas" routine that periodically removes over-repaired replicas which are caused by transient failures and retries during replica repairing. The bug was caused by a race between the load balancing routine and the remove-extra-replica routine, where the load balancing routine copies the replica from node A to B and deletes the replica in source node A, and at the same time the remove-extra-replicas routine detects that there are 4 replicas in total and chooses to delete the one in B.

Mishandling of transient failures. Sophisticated predicates enable more rigorous and thorough checks to catch bugs. As an example, the predicate of reference correctness ("no dangling references nor orphan replicas") checks the matching between index entries and replicas, and it helps us to identify a subtle bug caused by transient failures, which may degrade reliability. We killed a process in a node and restarted it after about 5 seconds. The predicate remained violated after quite a long time, even when the auxiliary measure for stabilization was high. Then we refined the predicate to output the owner of orphan replicas, which turned out to be the node suffering the transient failure. The bug is caused by mishandling of transient failures. The churn of the failed node cannot be observed by other nodes because the churn duration is shorter than the failure detection timeout (15 seconds). As a result, other nodes will not repair replicas or re-publish the index to this failed node, whose memory-resident state (e.g., the index) has already been wiped out by the failure.

Study of effectiveness. Both bugs are non-deterministic, complicated and sensitive to the environment, while catching the bugs and understanding the root causes require detailed memory states. Deterministic replay is necessary because we cannot dump all the memory states in logs in production runs. Like the Paxos case, the predicates directly specify the complex guarantee for correctness. Although they are liveness properties, predicate checking is very useful to pinpoint the starting point for inspection in replay. Suppose that we only have replay but not checking facility. For the first bug where replica loss has been observed, screening traces and finding out the root cause is tedious, while possible. In contrast, the second bug is difficult to even detect. This is because, eventually the replicas and owner nodes will notice the data loss through a self-managing routine in BitVault and repair the loss. Thus, the delay in repair is undetected and will degrade reliability.

4.4 Macedon-Chord

From the latest version of the Macedon release (1.2.1-20050531) [1], we generated the WiDS-based implementation for three protocols, RandTree, Chord, and Pastry. Due to space limitations, we only report our findings of Macedon-Chord. The Macedon description files already have logics of initialization, joining and routing, and we wrote our drivers to add our testing scenarios. The test is carried out in the simulator: 10 nodes join the ring around the same time and then remove one node after stabilization. We use the predicate in Figure 2 to check that the ring is well-formed, and add another to check fingers to be pointing to correct destinations.

Interestingly, this simple test altogether revealed five bugs. Two bugs are not caught by the predicates - they are programming errors that crash the simulation (divide-by-zero and dereferences of uninitialized pointers). The remaining three bugs are protocol implementation bugs caught by the predicates.

The first one caused a broken ring after the node leaves, caught by the $RingConsistency$ predicate[3]. Using replay to trace the origin of the incorrect successor value from the violation point, we found that the field of the successor's successor is wrongly assigned with the hash of the corresponding node's IP address, instead of the IP address itself.

The remaining two bugs caught by finger predicates cause problems in finger structures. One does not properly update the finger table, making the fingers converge to their correct targets much later than it needs to take. The last one is a mishandled boundary case. Let f be the ID of the i^{th} finger and $f.curr$ denote its current value. $f.start$ records the ID that is 2^i away from this

node's ID. If $f.start$ equals to $f.curr$, then the current finger is the perfect one. When a new candidate node with ID y is evaluated, a checking function $isinrange$ is called, and $f.curr$ is replaced with y if y falls between $[f.start, f.curr)$. $isinrange$ regards the special case $[f.start, f.start)$ as the entire ring and returns true for arbitray y. As a result, the perfect finger is replaced by any arbitrary y. In later rounds the perfect finger will make it back, and the process continues to oscillate. Macedon-Chord is later re-written with Mace. The author confirmed that the first four bugs were removed as a result of this exercise, yet the last one still remains in the new version.

Study of effectiveness. Structured overlay protocols are perfect examples of the complexity of distributed logic. Based on system properties on topology structures, checking overlay protocols could be very effective and productive. Sometimes the violation report is sufficient to infer the root cause and find the buggy code, e.g., the oscillation bug reveals the buggy logic for choosing finger node ID, without the need for the replay.

4.5 Performance and overhead

The logging overhead heavily depends on the applications. We performed a test run on BitVault with 4 nodes, each of which is a commodity workstation with 3GHz Pentium IV CPU and 512MByte memory, and the results are shown in Figure 8. The experiments consists of inserting 100 100KB objects at the 3^{rd} minute, crashing a node at the 6^{th} minute, rejoining it at the 9^{th}, and finally retrieving all objects at the 12^{th}. The replication degree is set to 3. The peak of the performance hit occurs on the 3^{rd} minute, with roughly a 2% runtime overhead. To collect logs more efficiently, we generate the objects that mostly contain a single value so as to achieve high compression rates. As a result, the log size reaches close to 600KB after compression. For uncompressed logs, the size is around 60MB.

Table 3 summarizes the performance of the checker. The second column shows how long it takes to perform the original run, with logging turned on. Depending on the message rate and complexity of predicate calculation, the checker slows down the replay between 4 and 20 times. The 15-minute BitVault run consumes about 37 minutes. We believe that given the benefits of using the checker, these overheads are acceptable for debugging tasks.

4.6 Discussions

Our experience validates a number of design points. In almost all cases, on-the-fly scripting has allowed us to adaptively refine predicates in response to predicate errors and to chase newly discovered bugs. This iterative process is especially useful when we check reproduced runs from deployed experiments, since the trace can be reused to find all the bugs it contains. The experiences also gave us a number of interesting lessons.

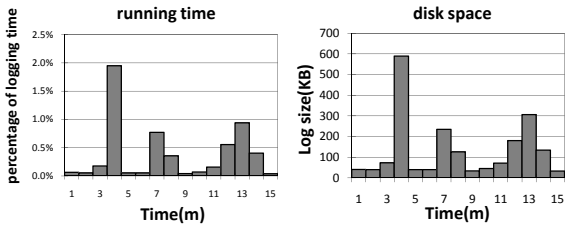

Figure 8: Logging overhead: (a) running time; (b) disk space.

Application	Original run	w/o checker	w/ checker
Paxos(simu.)	0.62	0.34	6.56
BitVault(deployed)	900.00	236.25	2219.77
Lock server(simu.)	5.34	3.14	11.99
Chord(simu.)	1.64	0.719	3.00

Table 3: Running time (in seconds) for evaluating predicates. Orignal run, w/o checker, and w/ checker columns show the running time for testing with log turned on, replay, and replay with predicate checking, respectively.

The advantage of predicate-based checking is its simplicity: it depends on only a snapshot of states (and sometimes augmented with a short history) to define invariant properties and catch violations. This methodology does not require the developer to build a state machine inside the checker that mirrors the steps of the implementation. At the same time, however, this means that we need to pay extra effort to trace the root cause backwards from violation point, which might be far behind. Time-travel with message flow definitely helps, yet it can be tedious if the violation involves long transactions and convoluted event sequences, as is the case of the Boxwood lock server. In these scenarios, the predicate checking is more like a programmable conditional breakpoint that people use to "query" the replay trace. Effectively pruning the events to only replay the relevant subset is one of the directions for our future work.

We also obtained considerable insight in terms of debugging distributed systems in general. The Paxos bug involves at least 5 processes, the deadlock case in the lock server involves 4 clients and 1 server, 2 locks, and the client has 6 flags in total. In both cases the error is deep and the system scale is larger than what a model checker is typically applicable to. It is therefore unclear whether a model checker can explore such corner states successfully. Also, 9 out of the 12 bugs have correct message communication pattern. For example, the bug found in the Paxos specification will not cause any violation of

the contract between proposers and acceptors on sending and receiving messages. Thus, without dumping out detailed states, it is questionable whether log analysis is able to uncover those bugs.

5 Related Work

Our work is one of many proposals to deal with the challenging problem of debugging distributed systems. We contrast it with the three broad approaches below.

Deterministic replay. In response to the lack of control of testing in real deployment, building a time machine to enable deterministic replay is required [7, 8, 27]. Most of this work is for a single node. Our implementation is capable of reproducing the execution of a distributed system within one simulation process. Friday [9] enhances GDB with group replay and distributed watchpoints. We share their methodology that incremental refinement of predicates is an integral part of cyclic debugging process, however, replay in WiDS checker is much more efficient because we use one process to replay all instances. In addition to debugger extensions, WiDS checker provides a unified framework with advanced features tailored for distributed systems. It allows simulation with controlled failures, which is valuable for early development stages. It can trace user-defined object states with historical versions and evaluate predicates only at event-handler boundaries, and thus provides better functionalities and performance for predicate checking. These unique contributions of WiDS Checker are proven to be important for debugging. Our current drawback is that the tool is limited to applications written using the WiDS API or Macedon.

Model-check actual implementation. Our work complements these recent proposals [20, 30] that check actual implementations. Model checking explores the state space systematically, but the issue of state explosion typically restricts the scale. MaceMC [11] and WiDS checker share many design concepts, but differ fundamentally on how we test a system and hit bugs. MaceMC uses model checker with heuristics to deal with liveness properties, but has to tradeoff the scale of the system. As we discussed in Section 4.6, some bugs rely on a fairly large scale and low-level implementation details, and cannot manifest in a downscaled or abstracted system (e.g., the deadlock case in Boxwood derived from running out of the socket pool). For such cases, tools like WiDS checker which simulates low-level OS APIs and uses deployed testing with replay-based checking will be necessary.

Log-based debugging. Communication structures encode rich information. Logs can be used to identify performance bottlenecks, as advocated by Magpie [5], Pinpoint [6] and many others [4, 23]. The logs that WiDS Checker captures contain enough information to enable performance debugging, but the focus of this study is on correctness debugging. Pip [23] also proposes that log-based analysis can root out structural bugs. In general, we are more confident that log-based analysis can reveal performance bugs than structural ones (a close read on Pip's results seems to confirm this). As demonstrated by the bug cases in this study, a correct communication pattern is only the necessary but not the sufficient condition to enforce correct properties. Logging detailed states is prohibitively expensive; it is easier to log non-deterministic events and to reconstruct the states. From our experience, full-state inspection with time-travel is critical to identify the root cause of a correctness bug. A nice by-product is that it also enables us to exhaustively test all bugs in a given reproduced run, an endeavor that is usually quite labor-intensive. We also differ on how correctness is expressed. Pip can construct path expectations from history. Since we are taking logs, this alternative is also available. However, we believe that our assertions are more powerful. They represent the minimum understanding of a system's correct properties, and are much easier to write and refine than a model.

Singh et al. [26] propose to build an online monitoring and debugging infrastracture based on a declarative development language [16]. They require that the system is programmed with a highly abstracted deductive model so as to enable checking, while WiDS Checker mimics low-level OS API semantics (thread, sockets and file I/O) and enhance them with replay, in order to check predicates and find bugs in existing systems. In addition, their online checking methodology is restricted by the communication and computation overhead in distributed systems, and thus the checking facility is less powerful than the offline checking used in WiDS Checker. As online and offline checking complement each other, our future work is to look at interesting combinations of the two methods.

6 Conclusion and On-going Work

In a unified framework, WiDS Checker is capable of checking an implementation using both simulated as well as deterministically reproduced runs reconstructed from traces logged in deployment. Its versatile script language allows a developer to write and incrementally refine assertions on-the-fly to check properties that are otherwise impossible to check. Single console debugging and message-flow based time-travel allows us to quickly identify many non-trivial bugs in a suite of complex and real systems.

Our on-going work addresses the limitations in supporting legacy binaries with function interception techniques. We intercept OS system calls and APIs to change them into events, and use an event queue to schedule the

execution, including thread switches, OS notifications, and socket operations. We also intercept a layer beneath the socket interface to add Lamport Clock annotation before network data chunks in a transparent way. By this means application can be logged and faithfully replayed in a transparent way, and we can futher bring the simulation, virtualization, and checking functions to legacy binaries in the fashion we performed with WiDS Checker.

7 Acknowledgments

We would like to thank our shepherd Petros Maniatis and the anonymous reviewers for their comments and suggestions. We are also indebted to Linchun Sun, Fangcheng Sun, Shuo Tang, Rui Guo for their help with the WiDS Checker experiments, as well as Roberto De Prisco, Lidong Zhou, Charles Killian, and Amin Vahdat for verifying the bugs we found.

References

[1] Macedon: `http://macedon.ucsd.edu/release/`.

[2] Phoenix compiler framework. `http://research.microsoft.com/phoenix/phoenixrdk.aspx`.

[3] WiDS release. `http://research.microsoft.com/research/downloads/details/1c205d20-6589-40cb-892b-8656fc3da090/details.aspx`.

[4] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[5] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004).

[6] CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic, internet services. In *Int. Conf. on Dependable Systems and Networks* (2002).

[7] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev. 36*, SI (2002).

[8] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *USENIX* (2006).

[9] GEELS, D., ALTEKARZ, G., MANIATIS, P., ROSCOEY, T., AND STOICAZ, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[10] JUMP, M., AND MCKINLEY, K. S. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL* (2007).

[11] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

[12] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978).

[13] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998).

[14] LIN, S. D., PAN, A. M., GUO, R., AND ZHANG, Z. Simulating large-scale p2p systems with the wids toolkit. In *MASCOTS* (2005).

[15] LIN, S. D., PAN, A. M., ZHANG, Z., GUO, R., AND GUO, Z. Y. Wids: an intergrated toolkit for distributed system development. In *HotOS* (2003).

[16] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005).

[17] LYNCH, N. *Distributed Algorithms*. 1996, ch. 8.

[18] LYNCH, N., AND TUTTLE, M. An introduction to input/output automata. In *Technical Memo MIT/LCS/TM-373* (1989).

[19] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004).

[20] MUSUVATHI, M., AND ENGLER, D. Model checking large network protocol implementations. In *NSDI* (2004).

[21] PRISCO, R. D., LAMPSON, B. W., AND LYNCH, N. A. Fundamental study revisiting the paxos algorithm. *Theoretical Computer. Science. 243*, 1-2 (2000).

[22] QIN, F., LU, S., AND ZHOU, Y. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA* (2005).

[23] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[24] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIC, D., AND VAHDAT, A. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI* (2004).

[25] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997).

[26] SINGH, A., ROSCOE, T., MANIATIS, P., AND DRUSCHEL, P. Using queries for distributed monitoring and forensics. In *EuroSys* (2006).

[27] SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX* (2004).

[28] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11*, 1 (2003).

[29] YANG, H., PIUMATTI, M., AND SINGHAL, S. K. Internet scale testing of pnrp using wids network simulator. In *P2P Conference* (2006).

[30] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst. 24*, 4 (2006).

[31] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP* (2005).

[32] ZHANG, Z., LIAN, Q., LIN, S. D., CHEN, W., CHEN, Y., AND JIN, C. Bitvault: A highly reliable distributed data retention platform. In *MS Research Tech Report (MSR-TR-2005-179)* (2005).

Notes

[1]WiDS, a recursive acronym that stands for "WiDS implements Distributed System", is the name of the toolkit.

[2]For conciseness, we omit the learners of paxos in our description.

[3]This is only a simplified ring consistency predicate for illustration purpose. The more complicated one which also detects disjoint and loopy rings would possibly catch more problems.