dBug: Systematic Evaluation of Distributed Systems

Jiri Simsa, Randy Bryant, Garth Gibson
Computer Science Department
Carnegie Mellon University

Abstract

This paper presents the design, implementation and evaluation of "dBug" – a tool that leverages manual instrumentation for systematic evaluation of distributed and concurrent systems. Specifically, for a given distributed concurrent system, its initial state and a workload, the dBug tool systematically explores possible orders in which concurrent events triggered by the workload can happen. Further, dBug optionally uses the partial order reduction mechanism to avoid exploration of equivalent orders. Provided with a correctness check, the dBug tool is able to *verify* that all possible serializations of a given concurrent workload execute correctly. Upon encountering an error, the tool produces a trace that can be replayed to investigate the error.

We applied the dBug tool to a couple of distributed systems – the Parallel Virtual File System (PVFS) implemented in C and the FAWN-based key-value storage (FAWN-KV) implemented in C++. In particular, we integrated both systems with dBug to expose the non-determinism due to concurrency. This mechanism was used to verify that the result of concurrent execution of a number of basic operations from a fixed initial state meets the high-level specification of PVFS and FAWN-KV. The experimental evidence shows that the dBug tool is capable of systematically exploring behaviors of a distributed system in a modular, practical, and effective manner.

1 Introduction

This paper addresses the problem of verification of distributed and concurrent systems. In the context of this paper such systems are assumed to be pieces of complex software, such as a distributed file system (e.g. GFS [11], GPFS [19]), a distributed key-value storage (e.g. BigTable [6], Dynamo [8]) or a distributed hash table implementation (e.g. Chord [16], Pastry [18]).

The question answered by this paper is how to systematically explore *all* possible executions of a concurrent workload of such a system from some initial state.

In a concurrent system, the worst-case number of possible orders in which concurrent events can execute grows exponentially with the number of the events. To provide guarantees about the behavior of the system, the system designers must verify all distinct orders. The first criterion for successful verification in a concurrent setting is thus the ability to evaluate the outcome of every execution order. The next criterion is then the effectiveness of automating the evaluation process, which should avoid evaluation of redundant or infeasible orders.

Traditional approaches to evaluating correctness of concurrent systems fail to meet these criteria. For example, *non-exhaustive approaches* such as testing [10] evaluate a system using a range of tests. Each test is typically repeated many times with the intention to drive the system into as many different situations as possible. However, there is no guarantee that all possible execution orders of a given test are encountered. To overcome these limitations, researchers developed *exhaustive approaches* such as theorem proving [5] or model checking [7, 13]. Although these approaches have the potential to provide guarantees about all executions of a given system, their practicality is often impeded by the abstractions introduced during modeling [2] or the need for expert knowledge to deploy the technique [15].

In recent years, we witnessed the creation of several *execution-based* checkers such as Verisoft [12], MaceMC [14], Chess [17], or MoDist [20]. Unlike testing, these tools are able to inspect all possible executions of a given test and unlike model-based approaches, they inspect the actual system using dynamic analysis. In principle, the execution-based checkers are built around the same idea: for a given non-deterministic system and its initial state, systematically and automatically explore all possible executions searching for errors. What differentiates these tools from one another is:

1. the range of systems they support
2. the sources of non-determinism they consider
3. the techniques used for exploring and reducing the state space of possible executions
4. the type of errors they are able to detect

This paper presents the design and implementation of the tool "dBug", which falls into the category of execution-based checkers. Unlike its predecessors, dBug targets systematic evaluation of distributed concurrent systems without imposing restrictions on the design platform of the system or sources of non-determinism considered. This flexibility is achieved by deferring the responsibility to select the level of abstraction at which non-determinism is exposed to the system developers.

Range of systems supported

The dBug tool supports a wide range of systems by decoupling the identification of sources of non-determinism from the exploration of different executions. Specifically, dBug offers a simple interface for distributed and concurrent systems to communicate information to dBug. In this way, the dBug tool can evaluate any system that can be linked with an implementation of the required interface.

Sources of non-determinism considered

The interface offered by dBug can be used by system developers to identify concurrent events of the system. These events may include, but are not limited to, communication over the network, issuing local I/O requests, acquiring and releasing locks, or accessing shared in-memory data structures.

State space exploration technique

The dBug tool uses the interface implemented by the system in question not just to identify the non-deterministic choice points, but also to keep track of the set of processes that currently run in the system and to control the execution of the system. In other words, dBug maintains a global view of the system and it uses this view to select an event in the system to execute next. In order to combat the state space explosion, dBug optionally uses a known technique [9] to exploit the commutativity between transitions of the system.

Another unique feature of dBug is its use of virtual machine infrastructure to run the system and to explore different executions of a workload in the system. This feature comes with several key benefits. First, when an error in the system is encountered, dBug produces a recording of the execution. Second, virtual machine snapshots are used to revert to previously visited states of the system. This feature enables the setup of initial states with complex in-memory and on-disk state. Third, the use of virtual machine infrastructure opens the possibility to leverage computational power of heterogeneous computing infrastructures.

Type of errors detected

The dBug tool checks correctness of tests that exercise some functionality of the system. These tests are expected to be provided by system developers in the form of binaries. For instance, in our experiments dBug was typically used to test if a concurrent workload can drive the system into an inconsistent state.

Contribution

The contribution of the paper is twofold. First, we present the design and implementation of dBug – a publicly available tool that leverages manual instrumentations for systematic exploration of distributed and concurrent systems. Second, we demonstrate the practicality of dBug through two case studies – the Parallel Virtual File System (PVFS) [4] and the key-value storage based on the FAWN architecture (FAWN-KV) [1].

The interface of dBug was used to expose the non-determinism resulting from concurrent operations in these systems. The number of lines of code needed to integrate both system with dBug was less than 100. The dBug tool was then used to systematically evaluate all possible execution orders of several workloads in order to verify correct operation of these systems. The dBug tool found new concurrency bugs in both of the systems.

Organization

The rest of the paper is organized as follows. We describe the design (§2) and the implementation (§3) of the dBug tool, our experience with applying dBug to PVFS and FAWN (§4), and conclude with a summary of the paper and plans for future work (§5).

2 dBug Design

In the remainder of this paper, a distributed concurrent system is viewed as a set of *agents* that communicate and coordinate through *shared resources*. An important characteristic of an agent is that its execution is sequential, and the concurrency in the system arises only from the concurrent execution of multiple agents. For instance, in a real system, agents represent either single-threaded processes or threads of a multi-threaded process.

While agents are an abstraction of the components of the system, the shared resources are an abstraction of the environment in which the system runs. Examples of shared resources include local CPUs, volatile memories, persitent storage, and the network. In practice, the access to the shared resources is managed by a software stack and an agent accesses the shared resources either implicitly – for example, a process or a thread is scheduled for execution – or explicitly through a well-defined interface – for example, making a system call to allocate dynamic memory.

For deterministic computing platforms, it is the interaction with the environment that motivates us to view concurrent systems as non-deterministic. Although these systems are in fact often deterministic, it is bad practice to make assumptions about the environment in which the systems will run. If these assumptions are inaccurate, the systems will act unexpectedly. Therefore, a good system design assumes that the outcome of an interaction with the environment is arbitrary. This is for example why good programmers always test return values of system calls.

Viewing the boundary between a distributed concurrent system and its environment as the divide between the deterministic and the non-deterministic lays the foundation for a method that systematically explores possible executions of a distributed concurrent system. The premise of the method is that if we identify and control all interactions of the system with the environment, we can run the system deterministically by picking a particular order in which these interactions happen. Assuming that we can repeatedly execute the system from the same initial state, we can explore all possible orders in which the interactions with the environment happen.

To combat the combinatorial explosion of the number of possible serializations of concurrent interactions, one may decide to ignore certain interactions. In other words, the non-determinism of the system is resolved only partially. Formally, the universum of all possible orders of interactions is collapsed into equivalence classes implied by some equivalence relation. The repeated execution of the system then selectively explores only orders that belong to different equivalence classes.

Arbiter

The dBug design introduces a centralized entity called the *arbiter* that maintains a global view of the agents in the system. It is assumed that every time a new agent enters or exits the system, the arbiter is notified. Similarly, an agent is assumed to contact the arbiter to request a permission to perform selected interactions with the environment. Notably, the requests to the arbiter are not considered to be interactions with the environment.

Assuming that the arbiter has a mechanism for identifying when all agents are done issuing requests to the arbiter, the arbiter can explore arbitrary order of interactions of the system with the environment. Specifically, the arbiter runs two loops: The first loop receives information from the agents and updates the global view accordingly. The second loop repeatedly waits for all agents to be done asking for permissions to perform an interaction. When such a situation occurs, the arbiter grants a permission to one of the agents to perform their interaction.

Let us illustrate the dBug design on a scenario in which two clients are trying to create the same object in a distributed file system. For simplicity, let us assume that both the clients and the servers are single-threaded processes. Further, let the interactions controlled by the arbiter be the messages sent between the nodes of the system.

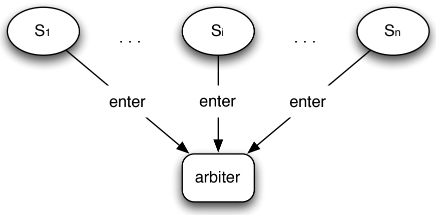

Figure 1: Server processes enter the system and notify the arbiter

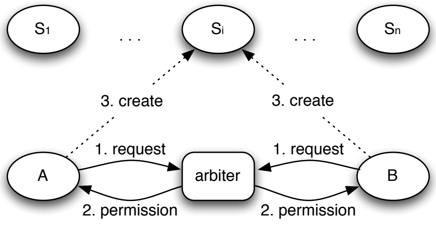

Figure 2: Steps taken to send a message: 1) An agent request a permission from the arbiter, 2) The arbiter grants the permission, 3) The agent sends the message

Initially, the server nodes enter the system and send a notification to the arbiter (Figure 1). Next, the two clients enter the system and send a notification to the arbiter. Under normal operation each client would then initiate the create operation in the system. Depending on the implementation of the distributed file system, the operation would require some number of communication messages. For simplicity, let us assume that there is only a single message going from each client to the server managing the object to be created followed by a single message going from the server to each client. Before each of the two clients proceed to sending the initial message,

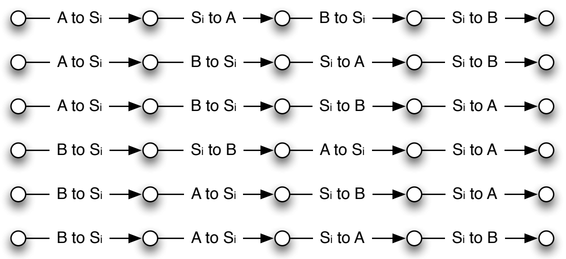

Figure 3: Six possible orders in which the four messages can be serialized by the arbiter

they ask the arbiter for a permission to send the message (Figure 2). The arbiter then decides, which of the two requests is serviced first. Similarly, the arbiter decides in which order should the subsequent messages be sent. When both clients receive a reply from the server, they exit the system and the execution terminates. Note that by following different strategies, the arbiter can trigger any of the six possible behaviors of the system (Figure 3).

Unfortunately, even when we limit ourselves to a specific type of interactions with the environment, the combinatorial explosion can still cause the number of possible orders of interactions to be astronomical. For example, if each of the above client operations would always require a sequence of 16 messages, there would be more than 600 millions different orders in which the two client operations could execute.

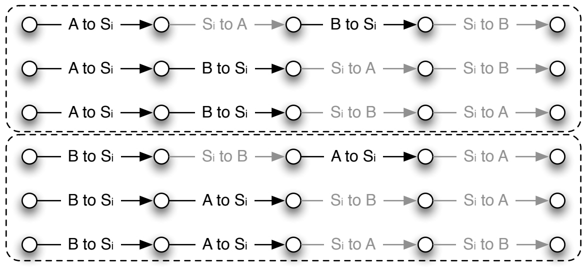

Figure 4: Two equivalence classes of message serializations. The gray events are independent of all events of other agents. Note that in this case the equivalence class is uniquely determined by the first message of the serialization.

The dBug design addresses this problem by providing an additional method that further collapses different executions into equivalence classes. This method is an adaptation of the *dynamic partial order reduction* of Godefroid and Flanagan [9]. The key idea behind the method is that interactions of different agents with the environment are inferred to be *independent*. If two orders of interactions with the environment differ only in the order of independent interactions, then they are considered equivalent. For instance, applying this method to our earlier example might infer that the replies send from the server to the two clients are independent of all interactions of other agents. In that case, the universum of 6 executions would collapse into 2 equivalence classes (Figure 4).

3 dBug Implementation

The dBug implementation consist of several components. First, it is the arbiter component, which is referred to as the dBug server. Second, it is the implementation of the interface that a distributed concurrent system uses to communicate information to dBug. The implementation of the interface is referred to as the dBug client. Third, it is the infrastructure for restoring an initial state of the distributed concurrent system and exploring different orders of interactions with the environment.

dBug Server

The dBug server is as a multi-threaded program. The *handler* thread of the server is responsible for listening for connections on a socket. This socket is used by the distributed concurrent system to communicate information about the state of the system to dBug and to request permissions to interact with the environment. The handler thread processes the messages received from the distributed concurrent system and maps this information into shared data structures of the server.

The *explorer* thread of the server periodically inspects the shared data structures of the server and checks if a request to interact with the environment can be serviced. The dBug server can run in two modes – the *blind* mode and the *informed* mode. The modes adopt different strategies to decide when to service a request. Ideally, we would like the dBug server to wait until it receives requests for all interactions that could get reordered – a situation referred to as the *steady state*. Unfortunately, as soon as there is a single agent in the system that can run indefinitely without requesting an interaction with the environment that is controlled by the arbiter, the check for the steady state becomes non-trivial.

The *blind* mode addresses the issue of detecting a steady state using a timeout – the mode waits for some fixed amount of time after receiving a request. If no other request is received in that time window, the server assumes that it has reached the steady state. The advantage of this mode is that it does not require any additional communication with the system. The problem with this approach is that there are two contradicting requirements on the length of the time window: The longer the time window, the higher the chance the server reaches the actual steady state. The shorter the time window, the lower the overhead of the exploration. The dBug tool by default uses a timeout of one second, while partially mitigating the performance overhead implied by the timeout. Specifically, the dBug tool remembers the number of requests at all steady states it has reached and uses this information to accelerate exploration of subsequent execution orders that encounter some of these steady states.

The *informed* mode addresses the issue of detecting a steady state by collecting additional information from the agents. In particular, an agent is expected to inform the arbiter when it reaches an *idle state* – a state in which the agent is not going to interact with the environment unless some event controlled by the arbiter happens. At the same time, an agent is expected to inform the arbiter when it transitions from an idle state to a *progress state* – a state in which the agent might potentially interact with the environment. The handler thread of the dBug server collects this additional information about the agents in the system and the explorer thread uses this information to detect a steady state. In particular, a steady state is detected when all agents reached an idle state and at least one agent visited its progress state since the previous steady state.

dBug Client

The dBug client is currently implemented as a C and C++ library that can be linked to a distributed concurrent system. The library implements the following interface:

- `register()`: Notifies the arbiter that an agent is entering the system. The arbiter replies with a unique agent identifier that is used for all subsequent communication between the agent and the arbiter.
- `unregister()`: Notifies the arbiter that an agent is exiting the system.
- `interaction()`: A blocking call that contacts the arbiter requesting a permission to interact with the environment of the system.
- `idle()`: A non-blocking call that notifies the arbiter that the agent entered an idle state.
- `progress()`: A non-blocking call that notifies the arbiter that the agent entered an progress state.

The interface provides for a range of instrumentations of a distributed concurrent system. The simplest instrumentation of a system requires that every process and thread creation issues the `register()` call and every process and thread destruction issues the `unregister()` call. Further, one needs to instrument the system with `interaction()` calls to identify interactions with the environment (sources of nondeterminism) for the arbiter to control. For example, in our experiments these interactions included sending messages between agents and acquiring and releasing locks by an agent.

By default, the dBug tool runs in the blind mode. Optionally, one may leverage the understanding of the system in question and instrument the system with `idle()` and `progress()` calls to notify the arbiter about the transitions between the idle and the progress state respectively. With these notifications in place, the dBug tool can run in the informed mode.

Exploration Mechanism

The last part of the dBug tool is the infrastructure that is used for restoring some initial state of a distributed concurrent system so that different orders of interactions with the environment for the same workload can be explored. Given the broad goals that dBug targets, we decided to use a virtual machine environment to implement this mechanism.

There are several advantages of using a virtual machine (VM) environment. First, a VM environment already supports a snapshot mechanism that can be used to store and to load a particular system state. Second, a VM infrastrucure allows us to record the execution of the system for further investigation of an unexpected behavior. Third, a VM environment allows us to leverage the computational power of heterogenous computing clouds, which account for a large part of current computational resources.

However, the use of a VM environment comes with certain disadvantages as well. First, the distributed concurrent system is assumed to function correctly in a virtual machine environment. This is not necessarily true for all distributed concurrent systems as some distributed concurrent system might rely on a custom hardware support that is not available in a VM environment. Second, the VM environment imposes a non-trivial overhead on the execution time of the system. In particular, reverting to a state of the system takes time on the order of seconds. This overhead limits the number of orders that can be explored within a given time budget.

Our original motivation for using a VM environment was its ability to capture the complete state of the system. Unlike previous tools [3, 21], we did not want to commit to some subset of the in-memory and on-disk state of the system. This design decision results in states of size proportional to the size of dynamic memories used in the system and space occupied on the persistent storage of the system. This can easily be gigabytes. Given the size of the states, the dBug tool implements a *stateless* exploration. Specifically, for each order of interactions explored, the dBug tool always starts anew from the initial state of the system. In other words, the stateless exploration only ever stores the initial state at the cost of exploring some parts of the state space multiple times.

The program that systematically explores different orders of interactions with the environment runs the following loop as long as there are orders to be explored:

1. Restore the initial state of the system
2. Generate a strategy for the arbiter based on past iterations
3. Start the arbiter with the strategy
4. Run the workload of the system
5. Store information collected by the arbiter for future use

The steps 2 and 5 of this loop ensure that no order of interactions is explored more than once.

A separate problem that dBug needs to address is that of the combinatorial explosion. For instance, if at each steady state except for the last one the arbiter has two request to select from, each execution takes approximately 5 seconds and each execution reaches 11 steady states before finishing, then it takes 85 minutes to explore all 2^{10} possible orders of interactions with the environment. If instead, the number of steady states of each execution would be 21, then the time needed to explore all of the 2^{20} possible orders of interaction with the environment would be approximately two months. To combat the combinatorial explosion, the dBug tool implements a variant of the dynamic partial order reduction of Godefroid and Flanagan [9]. In particular, the dBug tool uses information collected by the arbiter to compute independence relation over the universum of interactions with the environment. The key benefit of the independence relation is that it identifies pairs of interactions that can be reordered without changing some abstract behavior of the execution. The second step of the loop above then uses this information to avoid exploration of orders with identical abstract behavior.

4 The dBug Experience

We have used dBug to systematically explore execution orders of several workloads of two different systems: the distributed file system Parallel Virtual File System (PVFS) [4] implemented in C and the key-value storage based on the FAWN architecture (FAWN-KV) [1] implemented in C++. The following subsections describe the high-level functionality of the two systems and give a detailed account of both the process of integrating these systems with dBug and the verification case studies. All experiments were carried out on a Dual 2.26 Quad-Core Intel Xeon machine with 6 GB of memory. The virtual machine environment used in our experiments was the VMware Fusion running Ubuntu 10.4.

4.1 PVFS

The PVFS is a distributed file system that partitions both data and metadata across multiple servers. At each server, data is managed by a local ext2 file system and metadata is managed by a Berkeley DB. Further, file system operations are implemented using *state machines*. A state machine is a directed graph, with nodes representing either a C function or a nested state machine and labelled edges representing possible return value of nodes. An edge from one node to another is taken upon a match of the return value of the node with the label of the edge. A PVFS server is a single-threaded process that does cooperative multitasking over a set of state machines.

Interaction with PVFS is possible through a number of interfaces including virtual file system interface or custom MPI I/O libraries. In our experiments, we used the user-level implementation of PVFS file system operations provided as part of the PVFS distribution. In this context, each file system operation maps to a client process that executes a sequence of state machines that carry out the desired operation. Naturally, some of the client state machines can message a PVFS server to trigger execution of a server state machine.

Several steps were necessary to integrate PVFS with dBug. First, we extended the PVFS build process with an access to the dBug client object file. Second, we added instrumentations that notify the arbiter about creation and destruction of the client and server processes. Specifically, we added the `register()` and `atexit(unregister)` function calls to the beginning of the `main()` function of each client and server process. Third, we extended the messaging mechanism of PVFS so that each message sent is delayed until a permission from the arbiter arrives. This modification was probably the most involved as the send operations in PVFS are non-blocking. To correctly emulate this behavior we wrote a wrapper for the send operation, that spawns a slave thread to handle the send operation, while the master thread returns immediately. The slave thread registers at the arbiter, waits for the permission to issue the send, issues the send, unregisters at the arbiter, and returns. The three steps taken to integrate PVFS with dBug required less than 80 lines of code and enabled exhaustive exploration of different serializations of concurrent send operations.

However, the basic instrumentation had two deficiencies. First, with these instrumentation in place, the dBug server could run only in the blind mode – limiting both the speed of the exploration and the guarantees one can infer from the exploration. Second, the dBug server had only a limited chance to infer that two send operations are independent, resulting in the state space explosion.

To overcome the first deficiency, we instrumented both the client and the server processes to inform the arbiter about the idle/progress state of the respective node. Although this instrumentation in the end required only 10 lines of code, it was arguably more complicated than all the previous instrumentations put together. In order to correctly instrument the processes, we needed to inspect the design of PVFS to identify the proper way to check for the idle/progress state. In other words, while the previous instrumentations could be in principle automated with almost no understanding of the system, automating the instrumentation of idle/progress notification is infeasible as it – in the general case – requires solving the halting problem.

To battle the state space explosion, we used an abstraction of the system state. In particular, we instrumented the PVFS code so that each operation that updated the metadata state of the system was treated as an non-commutative operation and all other operations were treated as commutative operations. Any two commutative operations were then treated as independent. By adopting this abstraction we were able to guide the state space exploration towards the serializations that differ in the order of update operations. This instrumentation again required only 10 lines of code. Note that it is not necessarily true that two operations that are independent in the abstract domain are independent in the concrete domain. Consequently, this approach should be thought of as a search heurisitic rather than a sound method for collapsing the state space.

Experiment	**Mode**	**POR**	**Orders**	**Time**	**Bug**
ls	Blind	No	2520	43157	No
ls	Informed	No	2520	20469	No
ls	Informed	Yes	1	8	No
cr + cr	Blind	No	4e15*	1.3e17*	N/A
cr + cr	Informed	No	4e15*	6.7e16*	N/A
cr + cr	Informed	Yes	407	4619	No
ls + cr	Blind	Yes	932	28677	Yes†
ls + cr + rm	Blind	Yes	2271	86400+	Yes†

* Back of the envelope estimate. + Timed out after 12 hours.
† Deadlock encountered, not confirmed by the PVFS team as a bug.

Table 1: PVFS Experiments

The Table 1 details the experiments carried out with PVFS version 2.8.1. The system was configured with four PVFS servers with both data and metadata partitioned across all of the servers. The EXPERIMENT column identifies the name of the experiment, the MODE column identifies the mode in which dBug was ran, the POR column identifies if partial order reduction was used, the ORDERS column identifies the number of serializations explored, the TIME column identifies the time in seconds taken by the exploration, and the BUG column identifies if an error was encountered. First, a single `ls` operation was evaluated using the blind mode and the informed mode both with and without the partial order reduction. The `ls` operation results in 2520 possible message serializations because the implementation of `ls` concurrently communicates with all four servers. The experiment demonstrated that the informed mode halves the runtime necessary to explore all of these serializations and, unsurprisingly, partial order reduction collapses all serializations into one equivalence class. Second, two concurrent `create` operations were evaluated. Without employing partial order reduction, there would be on the order of 10^{15} different serializations to explore. With partial order reduction in place, 407 different serializations were sampled from the state space, representing executions with distinct abstract behavior. Finally, to test the scalability of the approach we evaluated workloads that concurrently list a directory and modify its contents using one or two operations. In the process of exploring these workloads using the informed mode the PVFS system reached a deadlock. However, this behavior could not be reproduced solely by fixing the message order and we are currently working together with the PVFS developers on further analysis of the behavior. The blind mode would ignore this behavior and simply restart the current iteration of the exploration loop every time a deadlock scenario was encountered.

4.2 FAWN-KV

The key-value storage based on the FAWN architecture partitions the dataset across a ring of *back-end* nodes using the technique of consistent hashing. The complexity of the back-end node is hidden behind *front-end* nodes that expose a simple put/get interface. The front-end nodes both route traffic between the clients and the back-end nodes and manage the ring of back-end nodes. The implementation of both front-end and back-end nodes consists of a single multi-threaded process. The system specification states the following consistency guarantee: a get operation on a key will return the result of the latest acknowledged put operation on the same key.

The implementation has several important sources of concurrency that are tighly linked to the consistency guarantee. Firstly, a back-end node stores its portion of the dataset in a local log-structured store to avoid the need for random writes. Periodically, a sequential scan of the store is used to compact the on-disk representation by removing obsolete entries. Secondly, when a new back-end node joins the system, it is assigned a portion of the key range to manage, possibly receiving existing data from the previous manager. Thirdly, to achieve fault tolerance the dataset can be optionally replicated on a number of back-end nodes, migrating data as existing nodes fail or new nodes join the system. For performance reasons, all of these operations happen on the background while incoming put and get requests are being serviced. As a result of these requirements the implementation uses a non-trivial amount of both local and distributed coordination (locks and RPC respectively) to strike a balance between correctness and performance.

Similarly to PVFS, we first extended the FAWN-KV build process with an access to the dBug client object file. Next, we added instrumentations that notify the arbiter about creation and destruction of the client and server processes. Specifically, we added the `register()` and `atexit(unregister)` function calls to the beginning of the `main()` function of both front-end and back-end nodes and wrote a wrapper for each thread creation. Third, we extended the locking mechanism of FAWN-KV – the pthread library – and the RPC mechanism of FAWN-KV – the Apache Thrift library – so that each coordination event is delayed until a permission from the arbiter arrives. The modifications to the pthread library were implemented via interposition, while the modifications to the Apache Thrift library were implemented at the source code level. The total number of lines required for all these instrumentations was less than 100.

Unlike for PVFS, we did not instrument FAWN-KV with the idle/progress notifications. The reason for this was both the complexity of the code base and marginal the performance gain of the informed mode over the blind mode experienced with PVFS. To combat the state space explosion, we decided to use an abstraction that assumes that two events are independent as long as they do not acquire conflicting locks. This abstraction represents a trade-off between precision and performance overhead.

Experiment	POR	Orders	Time	Bug
rewrite	No	5562	86400^{+}	No
rewrite	Yes	11	221	No
rewrite-bug	Yes	11	221	Yes
join	No	173	3107	Yes

$^{+}$ Timed out after 12 hours.

Table 2: Fawn Experiments

The dBug tool was used to verify correctness of 1) the "rewrite" operation that cleans up the log-structured store of a back-end node, and 2) the "join" operation that introduces a new back-end node into the ring. The Table 2 details the experiments carried out with a version of FAWN-KV that was made available to us by the FAWN team. The EXPERIMENT column identifies the name of the experiment, the POR column identifies if partial order reduction was used, the ORDERS column identifies the number of serializations explored, the TIME column identifies the number of seconds taken by the exploration, and the BUG column identifies if an error was encountered. First, the `rewrite` operation was ran concurrently with a put followed by a get on the same key. The purpose of this experiment was to evaluate if any order of concurrent events at the back-end node can lead to an inconsistent state, that is a situation in which the get operation returns an incorrect value. Without partial order reduction dBug explored 5562 orders and timed out after running for 24 hours. With partial order reduction, 11 orders were sampled from the state space. In both cases, FAWN-KV was found to be working correctly. The same experiment was also ran on a code base into which the FAWN-KV developers introduced a data race error. This error was detected by dBug running both with and without partial order reduction. Finally, we set up a test that introduced a new node into the ring of back-end nodes, while doing a put followed by a get on the same key. Again, the goal was to see if the system can be driven to an inconsistent state. This time, in less than one hour dBug discovered an error that the FAWN-KV developers suspected might have existed, but have not encountered it even after months of experimentation. The reason for this might be the corner case nature of the error. The error occurs only when FAWN-KV is ran with no replication and only two out of the 173 inspected orders actually lead to the error.

Download The experimental results and binaries of dBug are available for download from the website `http://www.cs.cmu.edu/~jsimsa/dbug`.

5 Conclusions

In this paper we have presented dBug – a tool that leverages manual instrumentation for systematic evaluation of concurrent and distributed systems. The tool belongs to the same category of verification tools as its predecessors Verisoft [12], Chess [17], MaceMC [14], and MoDist [20]. Together with MoDist it represents the only tool that targets systematic evaluation of distributed system written in general purpose programming languages. Unlike MoDist, the tool is publicly available and uses manual instrumentation that provides for different levels of abstraction.

Our initial experience with dBug showed that dBug is capable of enumerating thousands of different orders in which concurrent events triggered by a distributed system workload can execute. We used dBug to evaluate correctness of a number of tests for the PVFS and the FAWN-KV systems and found concurrency bugs in both of them.

The experimental evaluation demonstrated a considerable runtime overhead of using dBug. This overhead stems from two sources. First, it is the use of virtual machine environment. This overhead can be addressed by using more lightweight mechanism for restoring an initial state, such as a custom initialiazation function. Second, it is the performance penalty of detecting a steady state. Optimizing the detection of a steady state is one of our future goals.

Another aspect of dBug that yields itself to improvements is the degree to which integration with dBug perturbs the original system. Potential avenues for future work here are the use of the virtual machine environment to freeze agents that are waiting for a permission from dBug, or to control the system time of the agents.

The transparent verification of distributed systems is an emerging research area and there are many interesting research questions still to be asked and answered. The next question we would like to answer is when to use manual instrumentation over automated one. Our initial experience with both methods suggests that each approach comes with certain advantages and disadvantages. We plan to integrate both approaches inside of dBug and carry out a detailed comparison of the two.

Acknowledgement The authors would like to thank Sam Lang and Rob Ross for their advice on PVFS and Vijay Vasudevan for his help with FAWN-KV.

References

[1] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 1–14.

[2] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (New York, NY, USA, 2006), ACM, pp. 73–85.

[3] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08: Proceedings of the 8th Conference on USENIX Symposium on Operating Systems Design and Implementation* (2008), R. Draves and R. van Renesse, Eds., USENIX Association, pp. 209–224.

[4] CARNS, P. H., LIGON, W. B., III, ROSS, R. B., AND THAKUR, R. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference* (2000), USENIX Association, pp. 317–327.

[5] CHANG, C.-L., AND LEE, R. C.-T. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1973.

[6] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 205–218.

[7] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (1986), 244–263.

[8] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proceedings of 21st ACM Symposium on Operating Systems Principles* (2007), pp. 205–220.

[9] FLANAGAN, C., AND GODEFROID, P. Dynamic partial-order reduction for model checking software. *SIGPLAN Not. 40*, 1 (2005), 110–121.

[10] GELPERIN, D., AND HETZEL, B. The growth of software testing. *Communications of ACM 31*, 6 (1988), 687–695.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[12] GODEFROID, P. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), ACM, pp. 174–186.

[13] JR., E. M. C., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, 1999.

[14] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI '07: Proceedings of the 5th Conference on USENIX Symposium on Networked Systems Design and Implementation* (2007).

[15] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *SOSP '09: Proceedings of 22nd ACM Symposium on Operating Systems Principles* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 207–220.

[16] MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001* (San Diego, CA, September 2001).

[17] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *OSDI '08: Proceedings of the 8th Conference on USENIX Symposium on Operating Systems Design and Implementation* (2008), pp. 267–280.

[18] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Middleware* (2001).

[19] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 2002 Conference on File and Storage Technologies* (2002), pp. 231–244.

[20] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)* (April 2009), pp. 213–228.

[21] YANG, J., SAR, C., AND ENGLER, D. R. eXplode: A lightweight, general system for finding serious storage system errors. In *OSDI '06: Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation* (2006), USENIX Association, pp. 131–146.