Debug Determinism: The Sweet Spot for Replay-Based Debugging

Cristian Zamfir[1], Gautam Altekar[2], George Candea[1], Ion Stoica[2]
EPFL, Switzerland[1] *and UC Berkeley*[2]

Abstract

Deterministic replay tools offer a compelling approach to debugging hard-to-reproduce bugs. Recent work on relaxed-deterministic replay techniques shows that replay debugging with low in-production overhead is possible. However, despite considerable progress, a replay-debugging system that offers not only low in-production runtime overhead but also high debugging utility, remains out of reach. To this end, we argue that the research community should strive for *debug determinism* —a new determinism model premised on the idea that effective debugging entails reproducing the *same failure* and the *same root cause* as the original execution. We present ideas on how to achieve and quantify debug determinism and give preliminary evidence that a debug-deterministic system has potential to provide both low in-production overhead and high debugging utility.

1 Introduction

Debugging is hard. A key hindrance is hard-to-reproduce non-deterministic failures that are immune to traditional cyclic-debugging techniques. These failures manifest in production runs and may take months to diagnose manually [9]. After all, debugging entails significant detective work. Thus, practical tools for debugging non-deterministic failures are sorely needed.

Replay-debugging techniques [2, 4, 5, 6, 11, 12] offer a compelling approach to dealing with non-deterministic failures. A replay debugger produces an execution that is similar to the original failed execution. The hope is that the developer can then employ traditional cyclic-debugging techniques or automated analyses on the generated execution to isolate the defect underlying the failure. Many kinds of replay techniques have emerged over the years, differing primarily in how they deal with non-deterministic events (e.g., inputs, scheduling order, etc.). Record/replay techniques [2, 5, 6, 11, 12], for example, record non-deterministic events at runtime. Deterministic execution techniques [4], eliminate non-determinism (e.g., by precomputing scheduling order) to ensure deterministic replay. Finally, inference-based replay techniques [2, 11, 12] provide replay by computing unrecorded non-deterministic events after the original execution has finished.

Despite a plethora of replay techniques, a truly practical replay debugger remains out of reach. The traditional obstacle has been high runtime overhead, that is unacceptable in production environments. Alas, this is exactly where most unexpected and hard-to-reproduce bugs often surface. It seems clear now, however, that in-production overhead is no longer an impenetrable barrier. In particular, recent work on relaxed-determinism models [2, 12] shows that, by making fewer guarantees about the execution properties that are reproduced, one can shift runtime overhead from production time to debugging time. The failure determinism model [12], for example, guarantees only that the replayed execution exhibits the same final failure state. In so doing, it altogether avoids the need to record non-determinism, but has to infer it after the failure.

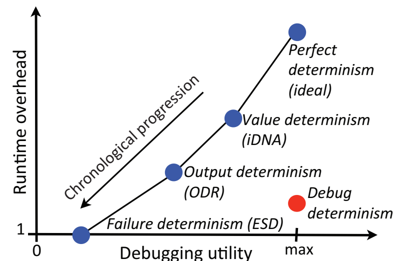

Figure 1: Trend in relaxation: recent ultra-relaxed systems reduce runtime overhead, but forego debugging utility.

In this paper, we argue that, while trying to satisfy the low runtime overhead requirement, designers of modern replay systems may have ignored another equally important one: effective debugging. The rush to relax determinism (plotted qualitatively[1] in Fig. 1) has left debugging utility by the wayside in favor of low runtime overhead. Overzealous relaxation (of which the present authors themselves are guilty [2, 12]) has resulted in a series of systems with low overhead, but unpredictable debugging utility.

To remedy the situation, we argue that a replay debugger should strive not only for low runtime overhead, but also for high debugging utility. This introduces two

[1] The figure is not based on new measurements. It shows the current trend in relaxation based on published results.

questions: what is high debugging utility determinism, and how do we get it?

Debug Determinism. Our answer to the first question is a new determinism model we call "debug determinism". The key observation behind debug determinism is that, to provide effective debugging, it suffices to reproduce *some* execution with the *same failure* and the *same root cause* as the original. A debug-deterministic replay system enables a developer to backtrack from the original failure to its root cause.

Root Cause-Driven Selectivity. One way to achieve debug determinism is to record or precompute the portions of the execution containing only the failure and its root cause(s), while relaxing the recording everywhere else. Unfortunately, this approach is infeasible, as the root cause of a failure is not known *a priori*. To this end, we give several heuristics that approximate this ideal approach by *predicting* the portions of the execution containing the root causes. A preliminary evaluation of such a heuristic on a data-corruption bug in the Hypertable distributed key-value store [1] suggests that this approach can indeed lead to low runtime overhead and debug determinism for that bug.

2 Perils of Over-Relaxation

In this section we describe several replay determinism models and the dangers of over-relaxing determinism.

Failure determinism, implemented by ESD [12], ensures that the replay exhibits the same failure as the original run. ESD does not do any recording. Instead, it extracts the failure information from a bug report or coredump and uses post-factum program analysis to infer an execution that exhibits the same failure.

Output determinism, implemented by ODR [2], ensures that the replay produces the same output as the original run. ODR uses several recording schemes. In the most lightweight scheme, ODR records just the outputs of the original run and infers all unrecorded nondeterminism. Scaling this inference process is hard, therefore ODR provides another scheme that also records the program inputs, the execution path, and the scheduling order. However, ODR does not record the causal order of the racing instructions running on different CPUs. Instead, it uses symbolic execution to infer the values that were read by the racing instructions.

Value determinism, implemented by iDNA [5], ensures that a replay run reads and writes the same values to and from memory at the same execution points as the original run. Value determinism does not guarantee causal ordering of instructions running on different CPUs, thus requiring more effort from the developer to track causality across CPUs.

Ultra-relaxed determinism models (e.g., ODR [2], ESD [12], PRES [11]) assume that debugging is possible regardless of the degree of relaxation performed. For some bugs, this is not true: ultra-relaxed models may not be able to reproduce the failure, hence making it hard to backtrack to and fix the underlying defect (i.e., root cause). For other bugs, these models will reproduce the failure, but may not reproduce the original root cause (indeed, multiple root causes are possible, see §4), hence *potentially deceiving the developer into thinking that there isn't a problem at all.* Finally, for some bugs, a significant amount of runtime information may need to be reconstructed, leading to prohibitively large post-factum analysis times.

To see why some failures may not be reproduced under ultra-relaxed determinism models, consider a program that outputs the sum of two numbers. Suppose, however, that the program has a bug such that for inputs 2 and 2, it outputs 5. To replay this execution, an output deterministic replay system (which guarantees only that the replay run exhibits the same outputs [2]) may produce an execution in which the output is 5 (like the original), but the inputs are 1 and 4. 1 plus 4, however, *is* 5 and thus is not a failure at all, much less the original failure. Unfortunately, without an execution exhibiting the original failure, developers cannot determine the true root cause of the faulty arithmetic (e.g., an array indexing bug).

To see how root causes may not be reproduced under ultra-relaxed determinism models, and why that can trick the developer into thinking there isn't a problem at all, consider the case of a server application that drops messages at higher than expected rates. Unbeknownst to the developer, the true root cause of this failure is a race condition on the buffer holding incoming messages. However, an output or failure deterministic replay debugger may not reproduce the true root cause. Instead, it may produce an execution in which the packets were dropped due to network congestion. Network congestion is beyond the developer's control and thus she naturally, yet mistakenly, assumes nothing more can be done to improve the program's performance. In the end, the true root cause (a race condition) remains undiscovered.

3 Debug Determinism

We argue that the ideal replay debugging system should provide *debug determinism*. Intuitively, a debug-deterministic replay system produces an execution that manifests the same *failure* and the same *root cause* (of the failure) as the original execution, hence making it possible to debug the application. The key challenge in understanding debug determinism is understanding exactly what is a failure and what is a root cause:

A **failure** occurs when a program produces incorrect output according to an I/O specification. The output includes all observable behavior, including performance characteristics. Along the execution that leads to failure, there are one or more points where the developer can fix

the program so that it produces correct output. Assuming such a fix, let P be the predicate on the program state that constrains the execution—according to the fix—to produce correct output. The **root cause** is the negation of predicate P.

A *perfect implementation* fully satisfies the I/O specification, that is, for any input and execution it generates the correct output. A deviation from the perfect implementation may lead to a failure. So, more intuitively, this deviation represents the root cause.

In identifying the root cause, a key aspect is the boundary of the system: e.g., if the root cause is in an external library (i.e., the developer has no access to the code), a fix requires replacing the library. Else, if the library is part of the system, the fix is a direct code change.

Debug determinism is the property of a replay-debugging system that it consistently reproduces an execution that exhibits the same root cause and the same failure as the original execution.

For example, to fix a buffer overflow that crashes the program, a developer may add a check on the input size and prevent the program from copying the input into the buffer if it exceeds the buffer's length. This check is the predicate associated with the fix. Not performing this check before doing the copy represents a deviation from the ideal perfect implementation, therefore this is the root cause of the crash. A debug-deterministic system replays an execution that contains the crash and in which the crash is caused by the same root cause, instead of some other possible root cause for the same crash. We give examples of failures with multiple root causes in §4.

The definition of the root cause is based on the program fix, which is knowledge that is unlikely to be available before the root cause is fixed—it is akin to having access to a perfect implementation. We now discuss how to achieve debug determinism without access to this perfect implementation.

3.1 Root Cause-Driven Selectivity

The definition of debug determinism suggests a simple strategy for achieving it in a real replay system: record or precompute just the root cause events and then use inference to fill in the missing pieces. However, the key difficulty with this approach is in identifying the root cause. One approach is to conservatively record or precompute all non-determinism (hence providing perfect determinism during replay), but this strategy results in high runtime overhead. Another approach is to leverage developer-provided hints as to where potential root causes may lie, but this is likely to be imprecise since it assumes *a priori* knowledge of all possible root causes.

To identify the root cause, we observe that, based on various program properties, one can often *guess* with high accuracy where the root cause is located. This motivates our approach of using heuristics to detect when a change in determinism is required without actually knowing where the root cause is. We call this heuristic-driven approach *root cause-driven selectivity (RCSE)*. The idea behind RCSE is that, if strong determinism guarantees are provided for the portion of the execution surrounding the root cause and the failure, then the resulting replay execution is likely to be debug-deterministic. Of course, RCSE is not perfect, but preliminary evidence (§4) suggests that it can provide a close approximation of debug determinism.

Next, we present several variants of RCSE.

3.1.1 Code-Based Selection

This heuristic is based on the assumption that, for some application types, the root cause is more likely to be contained in certain parts of the code. For example, in datacenter applications like Bigtable, a recent study [3] argues that the control-plane code—the application component responsible for managing data flow through the system—is responsible for most program failures.

This observation suggests an approach in which we identify control-plane code and reproduce its behavior precisely, while taking a more relaxed approach toward reproducing data-plane code. Since control-plane code executes less frequently and operates at substantially lower data rates than data-plane code, this heuristic can reduce the recording overhead of a replay-debugging system. The key challenge is in identifying control-plane code, as the answer is dependent on program semantics. One promising approach is suggested in [3]: deem low-data rate code as control-plane, since data-plane code often operates at high data rates. The same study empirically shows that such automated control-plane selection has high accuracy for several typical datacenter applications, such as Hypertable and CloudStore.

3.1.2 Data-Based Selection

Data-based selection can be used when a certain condition holds on program state. For instance, if the goal is to reproduce a bug that occurs when a server processes large requests, developers could make the selection based on when the request sizes are larger than a threshold. Thus, high determinism will be provided for debugging failures that occur when processing large requests.

A more general approach is to watch for a set of invariants on program state: the moment the execution violates these invariants, it is likely on an error path. This is a signal to the RCSE system to increase the determinism guarantees for that particular segment of the execution. Ideally, assuming perfect invariants (or specification), the root cause and the events up to the failure will be recorded with the highest level of determinism guarantees. If such invariants are not available, we could use dynamic invariant inference [7] before the software is released. While the software is running in production, the

replay-debugging system monitors the invariants. If the invariants do not hold, the system switches to high determinism recording, to ensure the root cause is recorded with high accuracy.

3.1.3 Combined Code/Data Selection

Another approach is to make the selection at runtime using dynamic triggers on both code and data. A trigger is a predicate on both code and data that is evaluated at runtime in order to specify when to increase recording granularity. An example trigger is a "potential-bug detector". Given a class of bugs, one can in many cases identify deviant execution behaviors that result in potential failures [13]. For instance, data corruption failures in multi-threaded code are often the result of data races. Low-overhead data race detection [10] could be used to dial up recording fidelity when a race is detected.

Therefore, triggers can be used to detect deviant behavior at runtime and to increase the determinism guarantees onward from the point of detection. The primary challenge with this approach is in characterizing and capturing deviant behavior for a wide class of root causes. For example, in addition to data races, data corruption may also arise due to forgetting to check system call arguments for errors, and increasing determinism for all such potential causes may increase overhead substantially. A compelling approach to create triggers is to use static analysis to identify potential root causes at compile time and synthesize triggers for them.

All heuristics described above determine when to dial up recording fidelity. However, if these heuristics misfire, dialing down recording fidelity is also important for achieving low-overhead recording. For code-based selection, we can dial down recording fidelity for data-plane code. For trigger-base selection, we can dial down recording fidelity if no failure is detected and no trigger fired for a certain period of time.

3.2 Assessing Debug Determinism

So far, work on replay-debugging has not employed metrics that evaluate debugging power. Instead, the comparison was mainly based on recording performance figures and ad-hoc evidence of usefulness in debugging. Instead, we propose a metric aimed at encouraging systematic progress toward improving debugging utility.

Debugging fidelity (*DF*) is the ability of a system to reproduce accurately the root cause and the failure. If a system does not reproduce the failure, debugging fidelity is 0, because developers cannot inspect how the system reaches failure. If the system reproduces the original root cause and the failure, debugging fidelity is 1. If the system reproduces the failure, but a different root cause from the original, debugging fidelity is *1/n*, where *n* is the number of possible root causes for the failure observed in the original execution. This definition takes into account the fact that a replayed execution is still useful for debugging even if it reproduces the failure through a different root cause, yet the replay is useless for debugging if it does not reproduce the failure.

It may be difficult to analytically determine a replay system's debugging fidelity. However, it is possible to determine it empirically. For instance, static analysis could be used to identify the location of all possible root causes for a certain failure, potentially including false positives. One can then manually weed out the false positives and check if the system can replay all of the true positives. Another approach is to empirically test if a replay-debugging system correctly replays in the cases when given root causes are guaranteed to be present in the original execution through some other means (e.g., deterministic execution).

Debugging efficiency (*DE*) is the duration of the original execution divided by the time the tool takes to reproduce the failure, including any analysis time. Normally this metric has values less than 1, but it is possible for techniques such as execution synthesis [12] to synthesize a substantially shorter execution. If this shorter execution compensates for post-factum analysis time, debugging efficiency can have values greater than 1.

Debugging utility (*DU*) is the product of debugging fidelity and debugging efficiency: $DU = DF \times DE$.

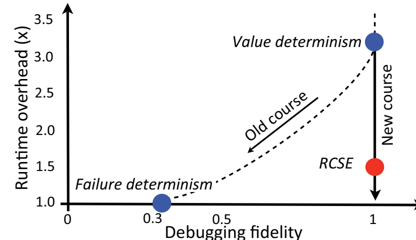

Figure 2: For the Hypertable bug, RCSE based on control-plane code selection enables escaping the relaxation trends shown in Fig. 1: it incurs slightly higher overhead than ultra-relaxed models, yet it achieves maximum debugging fidelity.

4 A Case Study

In this section, we present preliminary evidence that indicates replay-based tools can advantageously break out of the relaxation curve shown in Fig. 1. To acquire this evidence, we compared the recording overhead and debugging fidelity of RCSE against two other determinism models on a hard-to-reproduce data-corruption bug from the Hypertable distributed key-value store. We chose RCSE based on control-plane code selection (§3.1). The results in Fig. 2 show that RCSE has the potential to provide both low overhead recording and debug-deterministic replay.

We conducted our experiments on a previously-solved Hypertable defect. The *failure* is that updates to a

database table are lost when multiple Hypertable clients concurrently load rows into the same table. The load operation appears to be a success: neither clients nor slaves receiving the updates produce error messages. However, subsequent dumps of the table do not return all rows—several thousand are missing.

Root cause. The data loss results from rows being committed to slave nodes (i.e., Hypertable range servers) that are not responsible for hosting them. The slaves honor subsequent requests for table dumps, but do not include the mistakenly committed rows in the dumped data. The committed rows are merely ignored. The erroneous commits stem from a race condition in which row ranges migrate to other slave nodes at the same time that a recently received row within the migrated range is being committed to the current slave node.

Fig. 2 shows results for debugging fidelity, not full debugging utility. The latter requires that we empirically derive debugging efficiency (recall that debugging utility is a product of fidelity and efficiency). Empirical derivation at this early stage is hard, as it depends on the details of the particular inference engine. Debugging fidelity, in contrast, can be evaluated independently of an inference mechanism. Although high fidelity alone does not imply high utility, it suggests encouraging potential.

Debugging fidelity. Our measurement method for debugging fidelity depends on the determinism model.

Value determinism. Our approach was direct: we replayed the execution using Friday [8] and determined whether the replay indeed exhibited the original failure and root cause as described in the bug report. For our chosen bug, it always did, thus debugging fidelity is 1.

RCSE. Our approach was indirect: we determined whether the observed failure and its root cause were contained in the control-plane code. We classified application code into control and data-plane using the taint flow analysis described in [3]. If the root cause was recorded, we deemed the failure and root cause to be reproducible by an RCSE system based on control-plane code selection: such a system ensures that control-plane code behavior is reproduced consistently. For this bug, both the root cause and failure were in the control plane, hence the debugging fidelity of 1.

Failure determinism. By definition, failure-deterministic systems reproduce the failure and only one root cause. We computed fidelity as $1/3$, because the failure has at least 3 potential root causes, any of which may be reported by a failure-deterministic system. Specifically, another potential root cause is that a Hypertable slave responsible for a part of the table crashes after the data is uploaded, causing subsequent table dumps to return less data than expected (an expected behavior). Another potential root cause is that the client responsible for retrieving the previously stored table data runs out of memory before it has had a chance to finish the dump, resulting in apparent data corruption.

Recording overhead. We measured each model's recording overhead by modifying existing replay systems (Friday [8] for value determinism and RCSE, and ESD [12] for failure determinism). For RCSE, this meant recording just the data on control-plane channels and the thread schedule. For failure determinism, this meant recording only the failure state. For value-determinism, we recorded all inputs and thread interleavings, similarly to SMP-ReVirt [6].

5 Open Questions

Debug determinism assumes that the developer is interested solely in the original failure and root cause. It is possible, however, that a developer may want to find *all* potential root causes for a given failure. Thus, a system that records just the failure and finds all root cause-equivalent executions that exhibit the failure would be ideal. The challenge is scaling this approach to real programs.

Finally, while debug determinism may be the sweet spot in the problem domain of debugging, it is unclear what the sweet spot is for other replay-amenable problem domains. In particular, what are the ideal determinism models for replay-based forensic analysis and fault tolerance? Can the same principles behind debug determinism be applied to these problems?

References

[1] Hypertable issue 63. http://code.google.com/p/hypertable/issues/.

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore programs. In *Symp. on Operating Systems Principles*, 2009.

[3] G. Altekar and I. Stoica. Focus replay debugging effort on the control plane. In *Workshop on Hot Topics in Dependable Systems*, 2010.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Symp. on Operating Systems Design and Implementation*, 2010.

[5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Intl. Conf. on Virtual Execution Environments*, 2006.

[6] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments*, 2008.

[7] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Intl. Conf. on Software Engineering*, 2000.

[8] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *Symp. on Networked Systems Design and Implementation*, 2007.

[9] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[10] S. B. John Erickson, Madanlal Musuvathi and K. Olynyk. Effective data-race detection for the kernel. In *Symp. on Operating Systems Design and Implementation*, 2010.

[11] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles*, 2009.

[12] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[13] C. Zamfir and G. Candea. Low-overhead bug fingerprinting for fast debugging. In *Runtime Verification Conf.*, 2010.