Lightweight Cooperative Logging for Fault Replication in Concurrent Programs

Nuno Machado
nuno.machado@ist.utl.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Software is usually released with bugs. Therefore, fixing bugs in a post-deployment environment is crucial. Over the years, many techniques have been proposed to handle this problem and coping with failures resulting from non-deterministic events. This report presents a survey of some of these approaches and proposes some possible lines of research to improve the current state of art.

1 Introduction

Software bugs continue to hamper the reliability of software. It is estimated that bugs account for 40% of system failures [1], leading to huge costs both to software producers and end users [2]. Over the years, different efforts have been made to develop new techniques to prevent and avoid errors during software production. As examples, one can highlight techniques such as the use of box testing [3, 4] and the use of formal methods [5].

Despite their undeniable value, these techniques are still too time consuming and expensive to match the time-to-market requirements imposed to the software industry [6]. This problem is exarcebated when we take into consideration the increasing complexity of modern software, due to the advent of multi-core systems. As a result, the software released to the marked turns out to be error-prone. Therefore, it is imperative to design and implement debug tools that alleviate the developers' burden of finding and fixing the software bugs, in particular those arising from concurrency issues.

Unfortunately, if debugging single-threaded applications can be cumbersome, debugging multi-threaded software is typically way more challenging. Contrary to sequential bugs that usually depend only on the program input and on execution environments (and therefore can be easily reproduced), concurrency bugs show an inherently nondeterministic nature. This means that even when executing the same code on the same machine with the same input, the exact timing of an instruction or code segment execution may vary from one run to another. Thus, reproducing this kind of bugs can take hours, days, or even months [7]. Since the time to fix a bug is directly related to developer's ability to reproduce it for diagnosis [8], any debug mechanisms that is able to provide whole-system *deterministic replay* is a significant asset.

However, achieving the original execution replay may require the recording of all relevant details of the faulty execution [9] (including the order of access to shared memory regions, thread scheduling, program inputs, signals, etc), a task that induces a large space and performance overhead during production runs.

This problem could be mitigated by exploring the fact that there is usually a large number of users running the faulty software. In other words, one should be able to leverage the great number of executions performed. By gathering and analyzing information collected from different users regarding both correct and faulty runs, one can improve the bug tracking process and make bug reproduction more cost-effective.

In the past decade, a significant amount of research has been performed in order to provide better ways to efficiently debug nowadays complex systems. So far, the focus of related work has been on minimizing the recorded information during production runs to achieve deterministic replay, or on analyzing the logs provided by users' community to statistically isolate the bug. While in the past these techniques have been used in isolation, this project aims at combining the best of both worlds: develop a lightweight mechanism to log relevant information during original executions, but also leveraging the data provided by different users.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present all the background related with our work. Section 4 describes the proposed architecture to be implemented and Section 5 describes how we plan to evaluate our results. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of minimizing the recording overhead in production runs in order to reproduce bugs of a program. To achieve this, this work will study how one can infer patterns of similarity among different users' executions to improve the bug replay.

> *Goals:* This work aims at designing and implementing a lightweight mechanism to log relevant information from multiple executions performed by a community of users of the same program and, subsequently, statistically analyzing the gathered data to extract valuable clues to reproduce the failure and to pinpoint its likely source.

Our approach to address this problem is based on the observation that the work of recording the information required to find a bug can be distributed by the large number of users that execute that program. If each user collects only a fraction of the traces, the performance of its instance of the program will not be significantly affected. In addition, it is possible to relax the need to extensively log production runs to deterministic replay the bug on the first attempt, by performing a statistical analysis of the aggregate of information sent by the users. Using this technique, concurrency bugs can be addressed by tracing precise points of the code, such as accesses to shared variables, thread interleavings and lock acquisitions.

The project will produce the following expected results.

> *Expected results:* The work will produce *i)* a specification of the algorithms to trace, collect and analyze information in order to deterministic replay the bug; *ii)* an implementation of a prototype of this system, and *iii)* an extensive experimental evaluation using real life version of programs with bugs that have already been identified.

3 Related Work

There are various approaches to prevent bugs in a program and to optimize the debugging process. Albeit the overviewed work presented in this section focuses on those that try to reproduce the failure or to statistically isolate it, one can also highlight other techniques, such as code analysis [10, 11].

The remaining of this section is organized as follows. Section 3.1 presents the deterministic replay approach. Section 3.2 identifies the main challenges and performance metrics in order to have a good debugging tool. Section 3.3 overviews some systems that employ the record/replay technique. Section 3.4 presents the statistical debugging approach and the main solutions that follow this method. Finally, Section 3.5 concludes the section.

3.1 Deterministic replay

Developers often employ *cyclic debugging* [12] to understand the root cause of a failure. It is called cyclic debugging because the developers rerun the program several times, in an effort to incrementally refine their clues regarding the bug and narrow its location.

This approach works relatively well for deterministic failures, since they can be easily repeated and observed simply by re-executing the program. However, cyclic debugging is not feasible when dealing with non-deterministic bugs, because they don't always reveal themselves in every execution.

The problem of non-determinism can be addressed by employing an approach called *deterministic replay* (or *record/replay*) [13]. The purpose of this technique is to re-execute the program, obtaining the exact same behavior as the original execution. This is possible because almost all instructions and states can be reproduced as long as all possible non-deterministic factors that have an impact on the program's execution are replayed in the same way [13].

Deterministic replay operates in two phases:

1. ***Record phase*** - consists of capturing data regarding non-deterministic events, putting that information into a trace file.
2. ***Replay phase*** - the application is re-executed consulting the trace file to force the replay of non-deterministic events according to the original execution.

3.2 Challenges and performance metrics

Although simple in theory, it is not an easy task to build a deterministic replay system in order to be applicable in practice. One needs to take in account certain aspects and make some trade-offs. We now present the main challenges and performance metrics that need to be considered when developing these kind of debugging systems.

3.2.1 Overhead performance The main challenge lies in determining the level of abstraction at which it will operate, that is to say, what and how much information must be recorded in order to achieve deterministic replay. If one wishes to get a replay with high accuracy with respect to the production run, it will incur in a great recording overhead. On the other hand, the less information is collected, the harder it will be to get a replayed execution which resembles to the original. Therefore, a solution should be [12]:

– **Space efficient** - recording implies saving the information somewhere, typically in a trace file. Thus, the total amount of space needed to record the information should be minimized.
– **Time efficient** - in order to monitor the original execution, it is needed to instrument the application. Consequently, there are more instructions to run and the initial performance will be degraded. Hence, this overhead should also be minimized in order to maintain the use of the program acceptable.

3.2.2 Non-determinism External factors often interfere with the program execution, preventing the timing and the sequence of instructions executed to be always identical. These factors have internal (e.g. access order to shared memory locations by multiple threads) or external sources (e.g. inputs from keyboard or network). In particular, the following reasons can hamper the reproduction of a program [14]:

– ***Processor-specific instructions:*** some processors have dedicated input instructions, whose output depends on the processor version. One example is the `RDTSC` instruction on a Pentium processor, which reads the timestamp counter. In Pentium versions that support out-of-order execution (instructions are not necessarily performed in the order they appear in the source code), the `RDTSC` instruction can return a misleading cycle count, because it could potentially be executed before or after its location in the source code. To avoid this, a serializing instruction is used. Serializing instructions force every preceding instruction in the code to complete before allowing the program to continue.
– ***Signals:*** signals make processors aware of external events, but can happen asynchronously. A signal may change the memory state, register values, and the program control-flow.
– ***Reads of un-initialized memory locations:*** the values read from memory locations which are not explicitly initialized often change in different runs.

– ***System calls:*** certain system calls may present a differing behavior because their results depend on the environment in which they are running. The Linux system calls `gettimeofday()` and `uname()`are some examples.
– ***Different access order of shared memory locations:*** the interleaving of read/write accesses to shared memory locations by different threads may vary from run to run.
– ***Inputs:*** the inputs from keyboard and network may not be the same depending on the execution.

Non-determinism arising from thread scheduling and signal delivering can be tackled by recording them with "logical time", instead of ordinary physical time. In fact, logical time may be sufficient to support deterministic replay in uni-processor systems [15]. Nonetheless, when one moves to the field of multiprocessors (SMPs and multi-cores), the scenario becomes more complex. In addition to thread scheduling, asynchronous events, and signals, one has to take into account how concurrent threads interleave with each other, since they actually execute simultaneously on different processors. Therefore, one needs to capture the global order of shared memory access and synchronization points. Obviously, this is not a problem when threads are independent from each other.

3.2.3 Privacy and Security Generally, post-deployment debugging techniques need to collect some information at the user site regarding the failed execution. This information is then sent to the developers, in order to understand and fix the bug. This necessarily brings privacy and security concerns. The former is concerned with the sensitivity and the confidentiality of user information sent to the developer site (for instance, it can happen that the bug is only triggered by some determined input format placed in the password field of a form). On the other hand, the latter is linked with the vulnerabilities which may be explored by an attacker. For instance, an attacker may *eavesdrop* the communication channel between user and developer, aiming at gather significant information about the user. Alternatively, he can perform a *denial-of-service* attack through an overflow of the developer site with forged information, thereby exhausting its resources.

Addressing these issues is not trivial, because they are closely related to social aspects and are sometimes technically complex. However, for privacy two solutions may be provided: 1) before sending any error report, the user may optionally examine its contents and decide whether to send it or not. Unfortunately, most users do not have the expertise to properly understand these reports, albeit some techniques may be used to ease this task [16]; 2) the user strictly forbids the transmission of error reports, thus making monitoring useless.

In addition, some techniques may be employed to minimize the amount of user private data revealed [16, 17], although they are not yet capable of fully anonymize the bug report.

Security issues may be tackled using cryptographic mechanisms, such as asymmetric keys and digital signatures.

3.2.4 Network Bandwidth Given that information is sent from the user site to the developer site through the network, one has to take into account the available bandwidth. First, the transmission of data should not compromise other user tasks that also need to access the network. Second, the amount of information recorded during production runs must be transferred in an acceptable time, in order to allow a faster analysis in the developer site.

On the developer site, it is also necessary to have sufficient network bandwidth available so it can properly handle the large number of user sites.

3.3 Record/Replay systems

Two approaches have been initially proposed to achieve deterministic replay [12]: the *content-based* (or data-driven) replay approach and the *ordering-based* (or control-driven) replay approach.

The *content-based* approach advocates the storing of all data read by the instructions during record phase. Later, one replays the execution providing the correct input to each instruction, therefore getting the same output and, consequently, deterministic re-execution. However, this method generates a large amount of logged data, thus becoming impractical.

The *ordering-based* approach does not require recording every instruction to replay execution. Instead, one only needs to know the initial state of the application and to log the timings of interactions with external sources, such as I/O channels, program files, or other threads. If these asynchronous events are replayed at the same point as they were delivered in the original execution, an equivalent execution is obtained. This method has the advantage of creating smaller logs, because most of the data read by the instruction stream is produced during the run. Thereby, the log contains only data that is not produced by the program itself but comes from somewhere else. However, the *ordering-based* approach suffers from the drawback that all the interactions with the environment must take place, so that the internal state of the application and the state of the environment are updated correctly. Otherwise, the internal state of the program will eventually differ from the one seen in the original execution. As a result, contrary to content-based approach where all data to execute any instruction is always available, ordering-based technique is not able to execute isolated instructions.

In fact, the approaches described above in their pure state are not feasible in practice because they operate on a level of abstraction that is too low, thus demanding too much trace data. Therefore, a mix of content-based and ordering-based techniques is commonly used in deterministic replay. This scheme is based on the notion that part of the required information for executing the instruction stream can be reproduced by the interactions with the environment (ordering-based), while the rest can be consulted from the trace file (content-based).

In the past few years, much research has been done in order to develop better record and replay systems. Based on how they are implemented, prior work can be divided in two main categories: hardware-based and software-based. Within this classification, one can also distinguish the solutions according to another criterion: whether they support multiprocessor replay or only uni-processor replay.

3.3.1 Hardware-based

In general, hardware-based solutions offer support for multiprocessor systems. One of the first approaches in this direction was the one proposed by Bacon et. al [18], which introduces a mechanism of multiprocessor replay by attaching a hardware instruction counter to cache-coherence messages to identify memory sharing. Although fast, this mechanism produces a large log.

More recently, new hardware extensions were proposed to minimize the runtime recording overhead. Relevant examples are the Flight Data Recorder (FDR) [19] and, later, BugNet [20] and DeLorean [21].

We will briefly describe each of these systems in the next paragraphs.

Flight Data Recorder [19]: This system focuses on recording enough information to replay the last one second of whole-system execution before the crash. Like Bacon et. al's scheme, FDR snoops the cache-coherence protocol. It outstrips the former by using a modified version of Netzer's Transitive Reduction algorithm [22] to reduce the number of logged races.
FDR continuously traces activity information, such as interrupts, external inputs, and shared memory access ordering. It also implements checkpoints, relying on the SafetyNet mechanism [23] to obtain a state that can be used to start the re-execution.
The authors claim that FDR modestly affects program runtime, as the performance overhead is less then 2% [19]. The combined sizes of logs needed for replay in FDR (it records checkpoints, interrupts and external inputs, and memory races) is about 34 MB. For this reason, FDR can operate on a "always on" mode in anticipation of being triggered. Finally, the hardware complexity in FDR is about 1.3 MB of on-chip hardware and 34 MB of main memory space.
However, for providing the last second of the full system replay, FDR has to log additional information, such as interrupts, I/O, and direct memory access (DMA) events. For intensive I/O applications, the size of the logs may be too large to be used in practice. Furthermore, FDR requires a core dump snapshot to be sent to the developer, whose size can go up to 1GB, depending on the program's memory footprint and the size of the main memory chip used in the system.

BugNet [20]: This system also focuses on multiprocessor systems and makes use of dedicated hardware buffers to trace the runtime information required to re-execute instructions that preceded a system failure. BugNet is based on FDR, but contrary to the former it does not strive to replay the full system execution. Rather, it focus on detecting application level bugs and hence replays only executions in user code and shared libraries.
BugNet's implementation approach is based on checkpointing. Each new checkpoint is created after a certain number of instructions have been executed and captured (denoted by *checkpoint interval*), allowing the start of

re-execution at the beginning of each interval. Checkpoints can be terminated by program crashes, interrupts, system calls, and context switches. When the termination is caused by application failures, the logs generated during recording will be used to help the debugging.

At the beginning of each checkpoint, BugNet records the initial register state. Then, it traces the values read by the first load memory accesses in each replay interval, or when a data race is detected. This information is referred to as the *first-load* log and is stored in a hardware-based dictionary. This is enough to guarantee deterministic program's re-execution, without having to replay the interrupts and system calls routines.

Since BugNet focuses on just capturing application level bugs, the logs are smaller than in FDR. In BugNet, the log size is less than 1 MB, so users can effortlessly communicate the log back to the developer. Furthermore, BugNet has very little performance overhead (less than 0.01%, as the SPEC programs used in the tests do not have many interrupts or system calls), and the total on-chip hardware required is about 48 KB.

Limitations of BugNet include the fact that it only catches errors that are identified by the application itself or by the operating system. Therefore, errors resulting from incorrect programming logic are not addressed. Moreover, given that BugNet only tracks application code, it cannot track bugs that derive from complex interactions between the user process and the operating system. Finally, the data provided by BugNet is only sufficient to replay the last few checkpoints before the occurrence of the bug. This makes its record/replay scheme unsuitable for profiling purposes.

DeLorean [21]: This system is a hardware-assisted scheme for deterministic replay, where instructions are atomically executed by processors as blocks (or chunks), similarly to transactional memory or thread-level speculation. Then, rather than recording data dependencies, it logs the total order in which chunks commit.

This results in two main advantages over the previous schemes. First, since the memory accesses of a processor can overlap and reorder within and across the same-processor blocks, DeLorean can record and replay an execution at a comparable speed to that of Release Consistency (RC) execution. In contrast, FDR and BugNet only record at the speed of Sequential Consistency (SC) execution. Second, it provides a substantial reduction in log size. This reduction is accomplished by either omitting the chunk size or the ID of the committing processor from the entry. To be able to omit the chunk size, one needs to decide deterministically when to finish a chunk. On the other hand, to be able to omit the ID of the committing processor from the log entry, one has to "predefine" the chunk commit interleaving. This can be accomplished by enforcing a given commit policy — e.g., pick processors in a round-robin fashion, allowing them to commit one chunk at a time. The drawback is that, by delaying the commit of completed chunks until their turn, one may slow down execution and replay.

Given the need of making trade-offs between performance and log size, DeLorean provides two different executions modes: *OrderOnly* (for better performance) and *PicoLog* (for smaller logs).
In the *OrderOnly* mode, the commit interleaving is not predefined, but chunking is deterministic. Hence, the chunk size does not need to be logged. During execution, the arbiter (module which is responsible for observing the order of chunk commits) logs the committing processor IDs in the *Processor Interleaving* (PI) log. During replay, it uses the PI log to enforce the same commit interleaving. The log size is smaller because there is only the PI log. In reality, each processor also keeps a very small *ChunkSize* (CS) log where, for each of its few chunks that were truncated non-deterministically, it records both the position in the sequence of chunks committed by the processor and the size. This mode has a performance 2-3% lower than that of RC. With chunk sizes of 2000 instructions (the optimal size according to [21]), *OrderOnly* uses on average only 2.1 bits (or 1.3 bits if compressed) per processor per kilo-instruction to store both the PI and CS logs.
In the *PicoLog* mode, chunking is deterministic and the commit interleaving is predefined. During both execution and replay, the arbiter enforces a given commit order. Each processor keeps the very small CS log discussed for *OrderOnly*, but there is no PI log. Thus, the log size decreases comparing to the *OrderOnly* mode. For chunks with 1000 instructions, *PicoLog* needs a compressed log with an average of 0.05 bits per processor per kilo-instruction. However, *PicoLog* has a worse performance, being 14% lower than RC. This is still faster than SC, which is, on average, 21% slower than RC.

Unfortunately, despite various optimization endeavors to reduce hardware complexity [21], all the previous approaches still demand significant hardware modifications. These modifications are not yet available nowadays, except on simulations.

3.3.2 Software-based Given that changing hardware always brings high manufacturing costs, alongside with an increase in complexity, software-based approaches have been the focus of a significant amount of research lately. We now present an overview of some of these approaches. We start by reviewing those targeting uni-processor systems and then move to discuss solutions coping the additional sources of non-determinism characterizing multiprocessor systems.

IGOR [24]: This system was one of the first software-based solutions for deterministic replay. It relies on checkpointing techniques for replaying programs, reconstructing the application state from a given previous checkpoint. However, since IGOR does not record external I/O events, re-execution may not be identical to the original if the environment has changed.
This method operates collecting information at individual virtual memory page level. To achieve this, it makes use of a new `pagemod()` system call, which determines the set of pages that have been changed since the previous checkpoint. To control checkpoints it employs another system call - `ualarm()`.

In the replay phase, IGOR consults the log file to get the most recent checkpoint for each virtual memory page. After that, it uses an interpreter to proceed the execution from the last checkpoint up to a instruction defined by the user.
Unfortunately, IGOR involves changes to (i) the compiler - to log data allocations, (ii) both the library and loader - to initiate the trace and to enable dynamic function replacement. The recording overhead during production runs varies from 50% up to 400%. In addition, re-execution is about 140x slower, thus becoming an unattractive approach. Finally, IGOR does not support non-determinism caused by multithreaded programs.

Flashback [15]: This system is implemented as an operating system extension and it provides deterministic replay to assist software debugging. Like IGOR, Flashback uses an content-based approach and relies on checkpoint techniques. However, it employs shadow processes to capture non-deterministic interactions between the monitored process and the operating system in a lightweight fashion. These interactions include system call invocations, memory-mapping, shared memory usage for multithreaded applications, and signals. For instance, if a process makes a read system call, Flashback records the return value of the system call and the data that the kernel copies into the read buffer. During replay, when this specific system call is found, the previous recorded value is then injected to the process by Flashback.
This tool provides three primitives [15]:
- *checkpoint* - captures the current state and returns a handler state, allowing the program roll back to if required.
- *discard(x)* - discards the captured checkpoint provided, avoiding any future attempts to roll back to this specific state.
- *replay(x)* - rolls back the process to the previous execution state pointed by the state handler provided and then the execution is deterministically replayed up to where *replay()* primitive is called.

These primitives are implemented using shadow processes. A shadow process is a snapshot of the running process created by replicating the in-memory representation of the process in the operating system. Its creation is achieved by creating a new structure in the kernel and initializing it with the contents of the monitored process structure (e.g. registers contents, process memory, file descriptors etc). The pointer to the shadow process is stored in the current process structure. The copy-on-write mechanism is used in order to reduce overhead. Moreover, since Flashback's intent is not to recover from neither system crashes or hardware failures, one does not need to persistently store shadow processes, which still further reduces overhead.
The results presented in [15] show that the impact in the performance of the application is about 10%. However, the space overhead grows linearly with the number of invocations for both read and write system calls, and the combined log size ranges from around 440KB to 830KB with 4500 and 9000 invocations, respectively.
A main limitation of Flashback is that it requires modification to debugging tools to incorporate support for the framework. Besides that, Flashback is

also not suitable for profiling purposes because the replay mechanism does not allow the instrumentation of the target program for the replay phase. Finally, recording and replaying of signals and deterministic replay of multithreaded applications is outlined in the future work but it is not currently supported by Flashback.

The above discussed mechanisms ensure deterministic re-execution only on uni-processor systems, not coping with the non-determinism associated with possible data races among threads simultaneously running on different processors. The following approaches strive to address these issues.

InstantReplay [25]: This system was one of the first software systems for deterministic replay on multiprocessors. It consists of a technique to replay shared memory accesses using an ordering-based approach. This technique allows the access to shared memory objects only through well-defined protocol CREW (Concurrent-Reader-Exclusive-Writer) primitives. This protocol is instrumented for execution replay, and sets down one of two possible states for each shared memory object:

- *concurrent-read:* all the processors can read, none can write.
- *exclusive-write:* one processor (the owner) can read and write; all the others do not have access.

Then, each shared memory object is extended with a version number, that is incremented after each write access during both record and replay phases. All threads record versioning information to its own trace file.
During the record phase a reader traces the current version number of its shared object. In turn, a writer traces the current version number of its shared object and the number of readers since the previous write access on his shared object. During the replay phase a reader waits until the current version number of its shared object matches the previously traced version number. On another hand, a writer blocks until the version number on its shared object matches the previously traced version number and until the number of readers also matches the traced count.
This technique tends to generate great amounts of recorded data when the granularity of shared memory accesses within the program is very small. Moreover, it results in a severe performance degradation with more than 8 processors executing, imposing more than 10x production-run overhead.

DejaVu[26]: This system is a record/replay tool designed at IBM that provides deterministic execution replay of concurrent Java programs by capturing how threads have been scheduled (ordering-based approach). The technique used by DejaVu to capture scheduling decisions is independent of the underlying operating system. It is based on the notion of logical thread schedule, where the number of critical events occurring between thread swapping is counted. DejaVu distinguishes two types of events: (1) *critical events*, namely synchronization operations (e.g. `monitorenter` and `monitorexit`) and accesses to shared variables, and (2) *noncritical events*, that can only influence

the thread that executes them. Hence, the scheduling of noncritical events is not relevant for replaying the recorded execution. Total ordering is achieved by attaching a global scalar timestamp to each critical event. It also uses one local clock per thread to allow each thread to pinpoint their schedule intervals.
To limit the size of the trace files, only a pair of clock values *FirstCriticalEvent* and *LastCriticalEvent* of each thread schedule interval is recorded. To capture the schedule interval for each thread, DejaVu relies on the observation that the local clock of a thread is only incremented while the thread is running and, hence, global clock and local clock values will differ. When a thread starts executing a critical event, it compares its clock value to that of the global clock. If values are different, the thread detects the end of the previous schedule interval and the start of a new schedule interval. Once the thread finishes executing a critical event, it increments the global clock and then synchronizes its local clock with the global clock.
During the replay phase, DejaVu reads a thread schedule from the trace file previously generated. When a thread is created and begins its execution, DejaVu supplies it with an ordered list of its logical schedule intervals. Then, the thread sets its local clock to the value of the *FirstCriticalEvent* from the next schedule interval and waits until the global clock value becomes equal to that value. At the end of each critical event, the thread checks whether global clock value becomes larger than *LastCriticalEvent* value of the current interval, which is the point where the thread starts to execute the next schedule interval. When there are no more intervals left, the thread terminates.
Although supports multiprocessors, the technique used in DejaVu enforces a global order on variable accesses across multiple threads, which incurs a large runtime overhead on multiprocessor applications. Moreover, given that each critical event must be synchronized on the global timestamp, only non-critical events may actually run concurrently, leading to short thread intervals and huge trace files. However, on a uniprocessor, overheads are less than 88% during the record phase. Trace files are less than 1 KiB/s.

JaRec [27]: Is a portable record/replay system for Java. It addresses specifically the problem of synchronization races when executing multithreaded Java applications.
This tool operates entirely on the Java-bytecode level, but requires the JVM Profiler Interface (JVMPI) to be used without modifying the JVM. Each class that is loaded by the JVM is instrumented using JIT, thus requiring no static instrumentation. The instrumentation modifies the monitor entry and exit events, the starting and joining of threads, and invocation points of wait and notify primitives. Contrary to DejaVu, JaRec dropes the idea of global ordering and uses a *Lamport's clock* to preserve the partial order of threads and to reduce the size of logs.
However, JaRec requires a program to be data race free in order to guarantee a correct replay, otherwise it only ensures deterministic replay up until the

first data race. This constraint makes this approach unattractive for most real world concurrent applications, given that is common the existence of benign or harmful data races.

The overhead of JaRec ranges from 10% to 125% on micro-benchmarks, while on macro-benchmarks, the observed overhead lies around 80% during the record phase. During the replay phase the overhead varies from 40% to 300%.

LEAP [28]: Is a recent deterministic replay system for concurrent Java programs on multiprocessors. LEAP's approach is based on a new type of local order with reference to the shared memory locations and concurrent threads (ordering-based approach). It relies on the observation that one does not need to guarantee global order of thread accesses to shared memory locations. Instead, it is sufficient to record the thread access order that each shared variable sees to achieve deterministic replay. The authors use mathematical models to prove the soundness of this statement.

To track thread accesses, LEAP associates an *access vector* to each different shared variable. During execution, whenever a thread reads or writes in a shared variable, the thread ID is stored in the *access vector*. Therefore, one gets local-order vectors of thread accesses performed on individual shared variables, instead of a global-order vector. This simple technique allows lightweight recording.

The overall infrastructure of LEAP consists of three modules: the transformer, the recorder, and the replayer. The transformer receives the program Java bytecode and produces two files: the record version and the replay version. Then, the record version is executed and the recorder module gathers each SPE's access vector. When the recording stops, LEAP saves both the access vectors and the thread creation order information, and creates a replay driver.

Finally, the replayer uses the logged information and the generated replay driver to start the execution of the replay version of the program. To guarantee the correct execution order of threads, LEAP takes control of the thread scheduling.

The evaluation results in [28] show that LEAP incurs less than 10% runtime overhead for real world applications, but still imposes a significant overhead in some cases (626% for an application with several shared variables accessed in hot loops). However, when comparing to InstantReplay, DejaVu and JaRec, the tests performed in third-party benchmarks demonstrate that LEAP is 5x to 10x faster than these systems. In terms of space overhead, the log size in LEAP is still considerable, ranging from 51 to 37760 KB/sec.

LEAP has also some limitations. As it only captures the non-determinism caused by thread interleavings, LEAP may not reproduce executions containing non-deterministic inputs, such as random number generators. Another drawback comes from the fact that LEAP always replays the program from the beginning, making it unsuitable for long running applications. Finally, LEAP cannot reproduce bugs arising from data races in JDK library, because it does not record shared variables in these APIs.

SMP-ReVirt [9]: This system was the first to record and replay multiprocessor virtual machines without requiring new hardware components. To minimize the recording overhead, SMP-Revirt leverages hardware page protection mechanisms to detect races between virtual CPUs in a multiprocessor virtual machine, instead of instrumenting every shared memory access. This has the advantage of being able to record and replay an entire virtual machine without changing its software. To address other sources of non-determinism, SMP-Revirt logs virtual interrupts, input from virtual devices (e.g. the virtual keyboard), network, real-time clock, and the results of non-deterministic instructions (e.g. those that read processors' time stamp counter).
Due to the page-level granularity, this approach is well suited for applications with coarse-grained data sharing, resulting in less than 10% performance degradation for 4 processors. However, SMP-ReVirt imposes more than 10x overhead for applications with finer-grained data sharing and false sharing. For example, the relative overhead for FMM SPLASH-2 benchmark increases from 50% to 636% when the number of processors increase from 2 to 4, as reported in [9]. The space overhead is also significant and scales poorly. The log size requirements range from 0.562 GB/day with a single processor to 90 GB/day with 4 processors, in the worst case.

All the previous approaches try to reproduce the bug on the first replay run, thus inducing large overheads during production runs. This also has the drawback of penalizing bug-free executions, which are much more frequent than the faulty ones [13]. Motivated by this, recent work has tried to further minimize the cost of recording the production run. By relaxing deterministic replay sacrificing the idea of getting a completely faithful re-execution, one can decrease the number of data logged, thus reducing the cost for user site executions. However, this brings another challenge related to the time needed to infer the unrecorded information during production runs. We now briefly present some of these approaches.

PRES [13]: This system is a record/replay technique to help reproduce bugs on multiprocessors. PRES (*Probabilist Replay with Execution Sketching*) aims at reducing the number of attempts needed to reproduce the bug, but relaxing the constraint of replaying it at the first try. By doing this, PRES can minimize the recording overhead during production runs, albeit at the cost of a increase in the bug replaying time during diagnosis. Assuming that diagnosis is done offline and automatically, this trade-off can probably be well tolerated by programmers.
The authors make also another pertinent observation: as long as the bug can be reproduced, it is of less importance for the programmers to reproduce it with precisely the same execution path seen in the original execution. Thereby, during production runs PRES logs only partial execution information – a *sketch.* This sketch will be used later by an intelligent partial-information replayer to reproduce the bug via multiple attempts to reconstruct the missing information necessary for reproduction.
In particular, PRES operates in three stages:

- ***Production run:*** records only relevant events in an execution sketch, which will be then sent to the developer site if a bug occurs (the authors do not take into consideration any privacy issues). PRES instruments the code using Intel's tool Pin and employs 5 different techniques of sketch recording that trade reproduction for lowered recording overhead (SYNC, SYS, FUNC, BB-N, BB). For instance, the most lightweight technique (SYNC) records only the global order of synchronization operations, while the one that offers the fastest reproducibility of the bug (BB) records the global order of basic blocks, thus requiring more recording points.
- ***Reproduction phase:*** automatically repeats the multiple attempts of replaying the program until the bug is revealed. After each failed replay attempt, feedback is generated to improve future attempts. To re-execute the program, PRES uses a module named PI-Replayer that consults either execution sketches or feedbacks for previous replay attempts at every non-deterministic point. Alongside, a Monitor controls each replay, searching both for executions that do not match, at some point, the original sketch (here the execution is stopped and feedback is generated) and the moment at which the failure is reproduced.
- ***Diagnosis phase:*** PRES intelligent replayer leverages on the complete information from the previous stage to reproduce the bug with 100% certain during each replay.

The obtained results show that sketching methods can reduce significantly the logging overhead during record phase, and also allow the bug reproduction with high probability within an acceptable time. For example, SYNC and SYS result in 6-60% overhead for non-server applications and in 7-33% throughput degradation for servers. These schemes can also replay 12 of the 13 evaluated bugs within mostly fewer than 10 replay attempts. On the other hand, FUNC and BB-5 can reproduce all 13 tested bugs with mostly less than 5 replay attempts, but with an overhead of 8-48% for servers and 18-779% for non-server applications. The authors also claim that SYNC's and SYS's overhead remained small across executions with an increased number of processors (from 2 to 8-cores), thus achieving a good scalability. In terms of log sizes, for example, SYNC needs 2 KB/req up to 126 KB/req, while FUNC requires 5.42 KB/req up to 3485.63 KB/req, for different server applications. As one can see, the values vary dramatically depending on the number of recording points.

ODR [29]: Is a replay system that addresses the *output-failure problem.* In other words, ODR aims at reproducing all failures visible in the output of a program in its subsequent replays. It relaxes the need of generating a high-fidelity replay of the original execution by producing a possible execution that provides the same outputs as the first. This is called *output determinism.*

This approach has the drawback of making no guarantees about non-output properties of the original run. Nevertheless, the authors claim that output determinism is valuable for debugging purposes because: (i) the output-visible errors (e.g. crashes and core dumps) are reproduced, (ii) although sometimes different, the memory-access values provided are consistent with the failure, and (iii) it does not require the values of data races to be identical to original ones.

Although the obvious benefits in terms of decreasing the storage overhead, not recording the outcomes of data-races makes reproducing a failed run a very challenging task. This because the bugs often depend on the outcomes of races. To address this problem, rather than record data-race outcomes, ODR infers the data-race outcomes of an output-deterministic run. Once inferred, ODR substitutes these values in future program replays, thus achieving output-deterministic re-executions.

To infer data-race outcomes, ODR uses a technique named Deterministic-Run Inference (DRI). DRI's job is to search the space of possible runs to find one whose outputs are similar to those experienced in the original execution. Since an exhaustive search of the run space is intractable for all but the simplest programs, two techniques are employed to ease this task. The first is to *direct the search*, which by leveraging carefully on selected properties recorded during productions runs (e.g. schedule, input, and read trace), allows to prune extensive portions of the search space. The second technique consists in *relaxing the memory-consistency* of all runs in the run space to find output-deterministic runs with less effort. This is possible because, in general, a weaker consistency model allows more runs matching the original's output than a stronger model, i.e., under a weaker consistency model, DRI only needs to find a possible schedule that produces the same output as the original schedule, without having to be strictly identical to the latter.

The evaluation of ODR showed that, albeit the recording overhead results in a slowdown of the application of only 1.6x, inference times can be too high for many programs (more than 24 hours in some cases). However, a tradeoff between recording overhead and inference time can be made. For instance, if one records all branches of an execution, the inference time can be reduced by orders of magnitude, while the production run suffers only a slowdown of 4.5x on average. Regarding log sizes, no concrete values are given by the authors.

ESD [30]: This system uses a technique for automating the debugging process via a *synthesis* of an execution of the program that reveals the bug. ESD (*Execution Synthesis Debugger*) makes use of a program and the coredump associated with a bug report to produce an execution of that program that causes the given error to manifest deterministically. Like PRES and ODR, ESD relaxes the goal of achieving true deterministic replay. It is based on the idea that replaying a synthesized execution that exhibits the same bug, even if it is not precisely the execution experienced by the user, can be sufficient to make a noteworthy improvement in the debugging task.

Execution synthesis works in two steps:

– ***Sequential path synthesis:*** ESD defines a searching goal, which comprises the basic block where the bug appeared and the condition on

program state that held true at that the moment of failure. Then, it does static analysis (on both program's control flow graph and data flow graph) to shrink the search space of possible paths to the basic block presented in the goal. Finally, employs symbolic execution to derive a feasible path to the goal from the over-approximation computed during static analysis.
– ***Thread schedule synthesis:*** in the case of multithreaded programs, ESD finds a schedule for interleaving the execution paths of the individual threads. To do this, it extends symbolic execution to also treat thread preemption decisions as symbolic. It uses the stack trace from the bug report to place thread preemption points in strategic places, e.g. before calls to mutex lock operations, that can lead to the desired schedule with high probability.

While conceptually these two phases are separated, ESD overlaps them and synthesizes one "global" sequential path, by exploring the possible thread preemptions as part of the sequential path synthesis. Thereby, ESD can get a serialized execution of the multithreaded application. Also, for both sequential path and thread schedule synthesis phases, ESD applies heuristics to make the search for a suitable path and thread schedule efficient.

The main benefit of this approach is that it does not require any tracing during the original execution, thus causing no performance degradation of the program at user site. This makes ESD attractive for performance-sensitive applications, such as web servers and database systems. The time experienced by the authors to synthesize executions of real bugs is considerable low, being less than 3 minutes in all cases. This clearly outperforms inference time of ODR.

However, given that ESD is based on heuristics, it could suffer of lack of precision, which increases the time to find the bug. Also, ESD requires the coredump of the application, which is not always available due to privacy issues. Finally, symbolic execution is not suitable for reproducing bugs that rely on inputs resulting from complex operations, such as cryptographic functions (e.g. it is very hard to find a string that was the input for a given hash).

3.4 Statistical Debugging

The deterministic replay approach is not the only way to improve the task of debugging. Statistical debugging is a recently proposed approach that aims at isolating bugs by analyzing information gathered from a large number of users. This technique improves deterministic record and replay as it focus more on diagnosing the bug than repeating it.

The idea behind statistical debugging is based on the notion that software applications are usually executed by a large user communities. Hence, instead of trying to detect the bug by relying only in data from runs experienced by a single user, statistical debugging attempts to speed up the bug tracking process by distributing the monitoring across different clients. By doing this, it is possible to extract patterns of similarity among the universe of collected executions that could lead to the failure.

In general, the infrastructure for statistical debugging consists of a central database which receives user reports from both successful and unsuccessful runs, and a module to statistically analyze the collected data. After isolating the failure, the central site can send back to the users a patch to fix the application.

This technique also brings the problem of how much information record during production runs in order not to degrade runtime performance at the user site. Alongside, it must take into account scalability issues, given that the central site has to be able to manage all the received reports.

In this section we will give an overview of some solutions developed to provide statistical debugging.

GAMMA [31]: Is a system whose purpose is to provide a continuous improvement of software applications after their deployment. It achieves this by distributing monitoring tasks across different users, in order to collect partial information that will be then combined to obtain the overall monitoring information.

GAMMA is based on two main technologies:

- ***Software Tomography:*** is based on sparse sampling and information synthesizing. This technique divides the monitoring task into a set of smaller subtasks and assigns these subtasks to different user sites. Each subtask requires less instrumentation than the main task, which allows the distribution of the monitoring cost among different software instances. This is has a great advantage comparing to other traditional monitoring approaches which require all the instrumentation sites to be applied to the same user application. As result, the experienced performance degradation by the user is significantly smaller.
- ***Onsite code modification/update:*** allows modifying or updating the application code at the user site. This capability lets software developers dynamically adjust the instrumentation to collect different kinds of information and to efficiently deliver patches and new features to users.

Using these two technologies, the process of using the GAMMA system consists of two cycles:

- ***Incremental monitoring:*** lets developers interact with software instances to adjust the information to be collected. This allows the developers to investigate problems directly in the field, without endeavoring to recreate the user environment in-house.
- ***Feedback-based evolution:*** allows the software evolution to fit user's needs. Since the information monitored is directly related to how users use the software, it is more likely, for example, that features more commonly used be fixed before others rarely executed.

Although the authors state that GAMMA uses lightweight instrumentation, no concrete evaluation results are presented in the paper. Nevertheless, GAMMA has the drawback of not collecting information about the success or failure of the program's execution. This prevents the effective use of information collected in the field for coverage testing, because it is not possible to compare the expected output with the actual output. Another issue of the GAMMA system is that it requires developers to select subtasks, which sometimes is a very complex process.

CBI [32, 33]: Was one of the first systems to employ statistical debugging. CBI (*Cooperative Bug Isolation*) is a sampling infrastructure for gathering information software executions produced by its user community. After collecting information CBI performs an automatic analysis of that information to help in isolating bugs.
CBI is based on sampling, that is to say, it monitors information only from time to time. This brings the benefit of having a modest impact on the performance of the program. However, given that some bugs occur rarely, it becomes more difficult to track them. In other words, one needs to guarantee that the sampling is statistically fair, so that the analysis is consistent with the happening events. CBI addresses this by using a Bernoulli process to do the sampling.
The information regarding program runs is collected via *predicate profiles* from both successful and failing executions. Predicate profiles are particular points of the program which are instrumented to provide data about their values. Logged predicates can be classified in three categories:

- ***Branches:*** for every conditional, there are two predicates to track whether the true or false branch was taken, respectively.
- ***Returns:*** at each call point of functions which return scalar values, there are three predicates to track whether the return value is < 0, > 0 or $= 0$.
- ***Scalar-pairs:*** at each scalar assignment $x = ...$, identify each same-typed in-scope variable y_i and each constant expression c_j. There are three predicates to track whether the new value of x is smaller, greater or equal to y_i and c_j, respectively.

The data gathered across multiple executions of the program is integrated into feedback reports. Conceptually, the feedback report for a particular execution consists of a bit-vector, with two bits for each monitored predicate (observed and true). The observed bit indicates whether the predicate was ever observed, while the true bit states whether the predicate, if observed, was ever true. In addition, there is a final bit that captures the overall execution success or failure.
This approach has the advantage of producing always the same amount of data independently of the sampling density or running time. Unfortunately, this implies a significant loss of information, since the order of observations is not recorded.
The CBI's automatic bug isolation process proceeds with the statistical analysis of the information gathered in order to pinpoint the likely source of the failure. Given that many of the logged predicates are irrelevant, CBI assigns a score to every predicate to identify the best failure predictor among them.

The predictors are scored based on *sensivity* (accounts for many failed runs) and *specificity* (does not mis-predict failure in a successful execution). Using these metrics, CBI selects the top predictors.
The performance impact of CBI's sampling varies directly with its density. Unconditional instrumentation adds a performance penalty of 13%, while with a sampling density of 1/100 the impact decreases to 6%. In turn, a 1/1000 density imposes only 0.5% of performance degradation. The logs generated are less than 40KB.
One of the main problems with CBI system is that it relies on a code duplication-based instrumentation scheme that doubles the size of the program. Such a large increase in code size may not be suitable in practice for some applications. Moreover, CBI does not address non-deterministic bugs.

HOLMES [34]: This system is a statistical debugging tool that isolates bugs by finding paths that correlate with a failure. Inspired by previous work of CBI, HOLMES elaborates on statistical debugging by investigating the impact of using path profiles to improve the accuracy of bug isolation. It is based on the observation that paths are a natural candidate for debugging as they capture more information about program behavior than predicate profiles. For instance, paths can provide more context on how the buggy code was exercised, which helps in the task of debugging, while predicates can only locate the point in code where the error occurred.
HOLMES can operate in two modes:

- ***Non-Adaptive debugging:*** this mode implements CBI's statistical debugging algorithm using path profiles instead of predicate profiles. Like the previous work, HOLMES instruments the program and collects path profiles information during program runs, which is then aggregated in feedback reports. The feedback reports have the same structure as those of CBI.
  In the next step, gathered paths are assigned numeric scores to determine the top predictor of bug from the set of all available paths. These scores also follow the metrics specified in CBI's approach.
- ***Adaptive debugging:*** Is a mode that arises from the fact that in large programs, usually only a small fraction of the code is buggy and thus relevant to debugging. Contrary to sampling, HOLMES adaptive technique starts with no instrumentation. In the initial phase, HOLMES receives only bug reports, which consists of a stack trace and partial state of the program at the point of failure. After obtaining an enough number of bug reports, HOLMES employs static analysis to point out portions of code that more likely contain the causes of the failure. Then, these portions of code are instrumented to monitor useful information and collect detailed profiles, being afterwards redeployed in the field. Because only important parts of the code are instrumented, HOLMES avoid the need for sparse random sampling.
  The process is then repeated, but this time HOLMES collects partial profiles in place of bug reports. These profiles are later analyzed using the

same techniques as in the non-adaptive mode. The analysis compute a set of bug predictors and if some of then are strong enough to explain the failure, then the iterative process ends (a predictor is classified as strong if it's score exceeds a defined threshold and weak otherwise). If that does not happen, HOLMES expands its search by using static analysis and bug predictors to identify other parts of code which are closely related to the weak predictors. In practice, this consists in identifying a set of functions that interact with weak predictors to be profiled in the next iteration. This iterative process carries on until strong predictors are found and all bugs have been explained.

The slowdown observed by the authors when evaluating HOLMES was less than 10%. In turn, the code space overhead imposed by the instrumentation is generally smaller than 50%, with exception for the EDG compiler where it reaches 250%.

HOLMES is attractive for software maintenance, as it avoids the tedious manual task of selectively replace client binaries with instrumented versions in order to collect more information about the problem. Therefore, developers can focus exclusively on fixing bugs.

However, weak predictors can be sparse. Hence, given that HOLMES explores only near weak predictors, it is possible to get stuck with no new sites available to explore.

Unfortunately, all the previous approaches described are not suitable to track concurrency bugs. These kind of bugs arise from the non-determinism inherent to operations involving multiple threads. Thus, they cannot be captured by predicates or profiles used in prior work, which focus only on one thread at a time. Thereby, new research has been done to address concurrency bugs with statistical debugging. The Cooperative Concurrency Bug Isolation (CCI) is the first to tackle these issues.

CCI [35]: This system is a low-overhead instrumentation framework to diagnose production-run failures caused by concurrency bugs. CCI works by recording specific thread interleavings during the original run, using then statistical models to identify strong bug predictors among the information recorded. This approach is built based on CBI principles, so CCI also leverages on sampling to keep low overhead in production runs and relies on statistical models that assign scores to predicates to discover the root causes of the failure.

However, unlike CBI, CCI strives to find causes of concurrency bugs. Therefore, it implements new sampling techniques, that address the non-deterministic challenges of these kind of errors. For instance, CCI sampling may require cross-thread coordination, because concurrency bugs involve multiple threads. Moreover, it must also keep each sampling period active for some time, because concurrency bugs always involve multiple memory accesses.

Thereby, CCI consider three different instrumentation schemes, that offer different tradeoffs between performance and failure-predicting capability:

- ***CCI-Havoc:*** tracks whether the value of a memory location is changed between two consecutive accesses from one thread. This captures the change of program states in the view of one thread at two nearby points, and may help to diagnose atomicity violations, which happen when programmers make incorrect assumptions about atomicity and fail to enforce mutual exclusion for memory accesses that should occur atomically. If these accesses happen to be interleaved with conflicting accesses from different threads, the program might behave incorrectly.
- ***CCI-FunRe:*** tracks function re-entrance: simultaneous execution by multiple threads. This may help to diagnose errors arising from misuse of thread-unsafe functions.
- ***CCI-Prev:*** tracks whether two consecutive accesses (read or write) to one memory location come from the same thread or distinct threads. This captures interactions among multiple threads at a fine granularity, and may help to diagnose data races and atomicity violations.

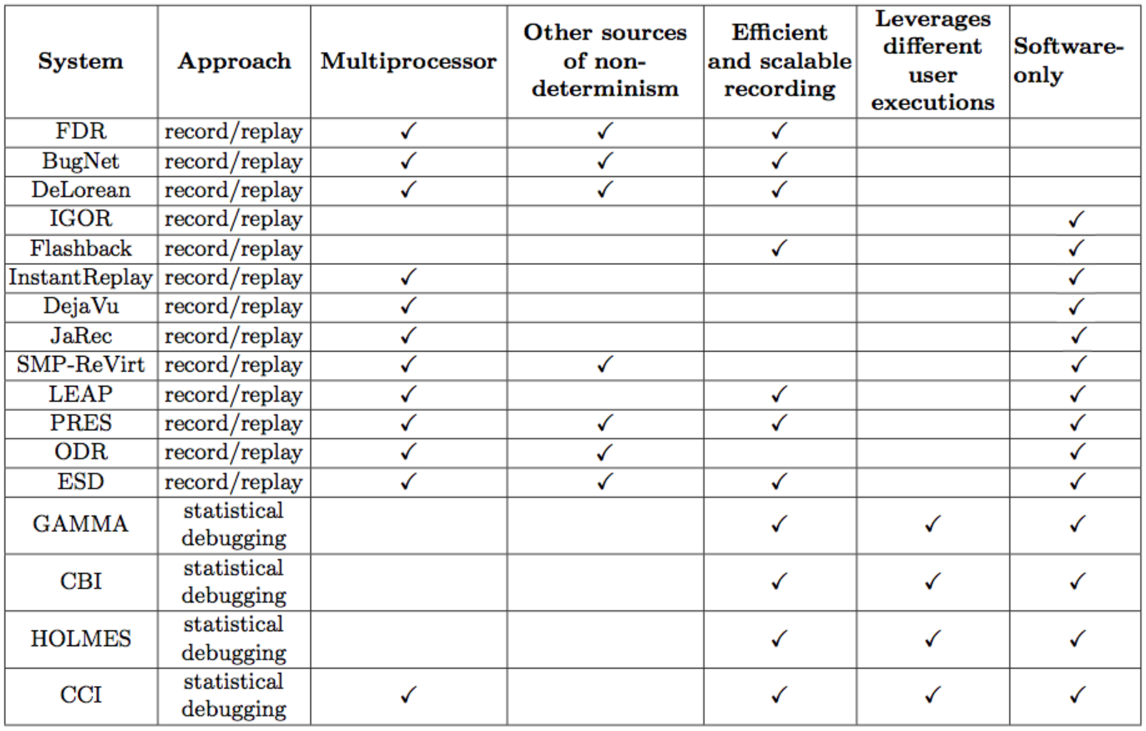
The evaluation results for CCI show that sampling significantly decreases monitoring overhead. The results obtained by the authors show that most of the runtime overhead experienced was lower than 10% for the applications tested. However, for memory-access intensive applications, the instrumentation schemes still incur very high monitoring overheads (more than 920%). Moreover, not all bugs can be diagnosed by CCI. From the 9 tested concurrency failures, CCI-Prev, CCI-Havoc, and CCI-FunRe could explain 7, 7, and 4, respectively. This is mostly due to limitations of each instrument scheme (CCI-FunRe shows the weakest diagnosis capability due to its coarse granularity) and loss of information in sampling.

3.5 Summary

Figure 1 summarizes the record/replay and statistical debugging systems previously presented. The systems are classified according to their approach and (i) whether they support multiprocessors and other sources of non-determinism (e.g. interrupts, I/O, etc), (ii) whether they employ an efficient and scalable recording mechanism (one considers the recording efficient if the overhead is generally less than 35%), and (iii) whether they leverage on data provided by different users.

Given that solutions were evaluated with different benchmarks and distinct units of measurement, one can not perform a precise comparative analysis. Despite that, some aspects can be highlighted. In general, hardware-based solutions present lower performance overheads than software-based solutions, but require hardware extensions which are not standard nowadays. On the other hand, software-only solutions typically have smaller logs.

Regarding software-based record/replay systems, one can highlight ESD as this solution do not incur any overhead during productions runs. Other solutions, namely Flashback, LEAP, PRES also have a modest impact on performance (less than 35%). However, Flashback does not cope with concurrency issues on multiprocessor. When comes to other sources of non-determinism, besides the



System	Approach	Multiprocessor	Other sources of non-determinism	Efficient and scalable recording	Leverages different user executions	Software-only
FDR	record/replay	✓	✓	✓		
BugNet	record/replay	✓	✓	✓		
DeLorean	record/replay	✓	✓	✓		
IGOR	record/replay					✓
Flashback	record/replay			✓		✓
InstantReplay	record/replay	✓				✓
DejaVu	record/replay	✓				✓
JaRec	record/replay	✓				✓
SMP-ReVirt	record/replay	✓	✓			✓
LEAP	record/replay	✓		✓		✓
PRES	record/replay	✓	✓	✓		✓
ODR	record/replay	✓	✓			✓
ESD	record/replay	✓	✓	✓		✓
GAMMA	statistical debugging			✓	✓	✓
CBI	statistical debugging			✓	✓	✓
HOLMES	statistical debugging			✓	✓	✓
CCI	statistical debugging	✓		✓	✓	✓

Fig. 1. Summary of the presented systems.

hardware-assisted approaches, only SMP-ReVirt, PRES, ODR and ESD provide support for I/O inputs and interrupts replay.

Comparing now record/replay systems with statistical debuging systems, Figure 1 shows that the former are mostly better prepared to deal with bugs arising from concurrent executions in multiprocessor programs. However, they rely only on one user execution, thus incurring more overhead in recording information during original executions or spending more time to infer unrecorded data in order to repeat the bug. In fact, CCI also supports multiprocessor applications, but, as CCI relies on sampling, it can miss some important clues to debug some concurrency errors.

4 Architecture

As discussed in the previous section, deterministic replay approaches do not leverage multiple executions performed by different users. This is done by statistical debugging approaches, but, unfortunately, these kind of systems are not yet too well suited to address concurrency bugs. As they rely on sampling, some important information may not be recorded.

The problem addressed in this work is to design and implement a prototype of a system that improves deterministic replay of failures by taking advantage of statistical information collected from different users' executions of the program. By doing this, we aim to further minimize the performance overhead and the log size that would be required if one had to record all the information at each client.

The system we propose will be built on top of LEAP [28]. In the following sections, we will present the relevant properties of LEAP that justify its choice as the basis of this work, as well as the proposed architecture to overcome some of the LEAP drawbacks.

4.1 LEAP Properties

Like LEAP, our solution will focus on replaying concurrency bugs, namely those resulting from data races and atomicity violations. Thereby, errors caused by other sources of non-determinism (e.g. I/O and network inputs) are outside the scope of this work.

LEAP was not built to leverage data from different clients, but, as stated before, has some key qualities:

- ***Software-based* -** LEAP is a software-only solution, thus it not requires extending commodity hardware with non-standard components.
- ***Local-order based deterministic replay* -** instead of enforcing a global order of thread accesses to shared memory locations in order to achieve deterministic replay, LEAP only records the thread access order that each shared variables. This information is stored in an access vector for each shared variable and used to enforce the same thread scheduling during replay.
- ***Recording overhead* -** the previous local-order recording scheme avoids the need for global synchronization operations, only requiring local synchronization operations which can be executed in parallel. This allows LEAP to be 5x to 10x faster than other related approaches, such as InstantReplay and DejaVu.

However, LEAP tends to produce considerable log sizes (51 up to 37760 KB/sec) and still incurs a significant overhead for some specific applications, namely those that have several shared variables accessed in hot loops. To address the log size issue and to further minimize recording overhead, our solution will expand LEAP in some aspects. To better explain the differences, we then present the proposed architecture.

4.2 Proposed Architecture

Figure 2 illustrates the architecture of the proposed system.

As one can see, there is a module in each client denoted *recorder*. The *recorder* will be responsible for collecting the access vectors for each shared variable during the execution of the (previously instrumented) program. Unlike LEAP, our

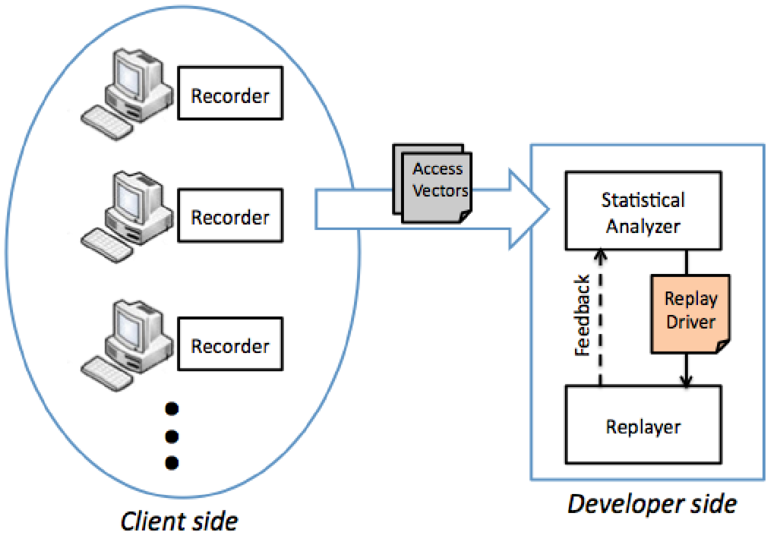


Fig. 2. Overview of the proposed system.

approach does not intend to record all shared variables' access vectors. Instead, each user will record accesses only to a part of the program's variables, which may differ from client to client. Assuming that the program will be executed by a large population of users, this mechanism will allow to gather access vectors for the whole set of shared variable with high probability. By doing this, we aim at minimizing the performance overhead that would be required if one had to record all the access vectors at each client.

Then, each client will send their partial access vectors (along with information about the success or failure of the execution) to the developer site to be analyzed. Here, the *statistical analyzer* will explore the received reports to complete the missing access vectors and generate a *replay driver* of the faulty execution to be replayed. By pinpointing the most common access vectors observed by the users, we plan to ease the task of inferring the original unrecorded access vectors.

Finally, the *replay driver* will serve as an entry point for the *replayer* component to control the replaying of the program execution. However, since the inferred access vectors may not be sufficient to reproduce the bug, one may have to generate a different replay driver. Because of this, the *replayer* will have to send feedback to the *statistical analyzer*, so the latter can investigate another feasible access vectors and produce a new replay driver.

5 Evaluation

The evaluation of the proposed architecture and the recording mechanisms employed will be performed experimentally, building a prototype. In detail, to evaluate our proposal we intend to proceed as follows:

- **Evaluation metrics:** to assess the quality of our system, we plan to quantify the *recording overhead* imposed, the *log size* needed and the correctness of the *bug reproducibility*, i.e., if the bug is in fact repeated or not.
- **Tests with real bugs:** there is a large variety of bug repositories of real applications available. We aim at using some of them (e.g. IBM ConTest and Tomcat bug repositories) to perform our tests. We will focus on concurrency bugs, trying to cover different types of these kind of failures, for instance data races, atomicity violations and deadlocks.
- **Comparison with other systems:** we can only claim that our solution is somehow better than other if we compare both. Therefore we plan to compare our system to the native execution and to LEAP system, but also, if time allows, to other solutions, such as DejaVu, InstantReplay and JaRec.

6 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23, 2011: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15, 2011: Deliver the MSc dissertation.

7 Conclusions

In this report we have surveyed some representative approaches to deal with the cumbersome task of finding and fixing bugs on released software. We have then discussed the main challenges and performance metrics that one has to take into account when building these kind of solutions. Several system have been presented to illustrate the current state of the art, with emphasis on those following the deterministic replay approach and the statistical debugging approach.

We have also proposed a solution that combines these two techniques and presented its architecture. We concluded the report with a description of the methodology to be applied in the evaluation of the proposed solution, as well with the schedule for future work.

Acknowledgments We are grateful to P. Romano, J. Garcia, P. Louro, P. Ruivo and L. Rosa for the fruitful discussions and comments during the preparation of this report. This work was partially supported by project "FastFix" (FP7-ICT-2009-5).

References

1. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now?: an empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability. ASID '06, New York, NY, USA, ACM (2006) 25–33
2. of Standards, N.I., Technology (NIST), D.o.C.: Software errors cost u.s. economy $59.5 billion annually. NIST News Release 2002-10 (2002)
3. E. Steegmans, P. Bekaert, F.D.G.D.N.S.M.v.D., Boydens, J.: Black & white testing: Bridging black box testing and white box testing
4. Tao Feng, K.B.: A survey of software testing methodology. (2010)
5. Hall, A.: Realising the benefits of formal methods. Journal of Universal Computer Science **13**(5) (2007) 669–678
6. Parnas, D.L.: Really rethinking 'formal methods'. Computer **43 (1)** (2010) 28–34
7. Godefroid, P., Nagappan, N.: Concurrency at microsoft – an exploratory survey. CAV Workshop on Exploiting Concurrency Efficiently and Correctly (2008)
8. S. Lu, S. Park, E.S., Zhou, Y.: Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. ASPLOS (2008)
9. Dunlap, G.W., Lucchetti, D.G., Fetterman, M.A., Chen, P.M.: Execution replay of multiprocessor virtual machines. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. VEE '08, New York, NY, USA, ACM (2008) 121–130
10. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. SOSP '07, New York, NY, USA, ACM (2007) 103–116
11. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. OSDI'08, Berkeley, CA, USA, USENIX Association (2008) 267–280
12. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., Bosschere, K.D.: A taxonomy of execution replay systems. In: In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet. (2003)
13. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: Probabilistic replay with execution sketching on multiprocessors (2009)
14. Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J.: Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. CGO '10, New York, NY, USA, ACM (2010) 2–11
15. Srinivasan, S.M., K, S., Andrews, C.R., Zhou, Y.: Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In: In USENIX Annual Technical Conference, General Track. (2004) 29–44

16. Castro, M., Costa, M., Martin, J.P.: Better bug reporting with better privacy. In: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. ASPLOS XIII, New York, NY, USA, ACM (2008) 319–328
17. Wang, R., Wang, X., Li, Z.: Panalyst: privacy-aware remote error analysis on commodity software. In: Proceedings of the 17th conference on Security symposium, Berkeley, CA, USA, USENIX Association (2008) 291–306
18. Bacon, D.F., Goldstein, S.C.: Hardware-assisted replay of multiprocessor programs. In: PROCEEDINGS OF THE ACM/ONR WORKSHOP ON PARALLEL AND DISTRIBUTED DEBUGGING, PUBLISHED IN ACM SIGPLAN NOTICES. (1991) 194–206
19. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In: Proceedings of the 30th annual international symposium on Computer architecture. ISCA '03, New York, NY, USA, ACM (2003) 122–135
20. Narayanasamy, S., Pokam, G., Calder, B.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: In ISCA. (2005) 284–295
21. Pablo Montesinos, L.C., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently
22. Netzer, R.H.B.: Optimal tracing and replay for debugging shared-memory parallel programs. In: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging. PADD '93, New York, NY, USA, ACM (1993) 1–11
23. Sorin, D.J., Martin, M.M.K., Hill, M.D., Wood, D.A.: Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: Proceedings of the 29th annual international symposium on Computer architecture. ISCA '02, Washington, DC, USA, IEEE Computer Society (2002) 123–134
24. Feldman, S.I., Brown, C.B.: Igor: a system for program debugging via reversible execution. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging. PADD '88, New York, NY, USA, ACM (1988) 112–123
25. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Comput. **36** (April 1987) 471–482
26. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools. SPDT '98, New York, NY, USA, ACM (1998) 48–59
27. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: Jarec: a portable record/replay environment for multi-threaded java applications. Softw. Pract. Exper. **34** (May 2004) 523–547
28. Huang, J., Liu, P., Zhang, C.: Leap: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. FSE '10, New York, NY, USA, ACM (2010) 385–386
29. Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. SOSP '09, New York, NY, USA, ACM (2009) 193–206
30. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: Proceedings of the 5th European conference on Computer systems. EuroSys '10, New York, NY, USA, ACM (2010) 321–334
31. Orso, R., Liang, D., Harrold, M.J., Computing, R.L.O.: Gamma system: Continuous evolution of software after deployment. In: In Proceedings of the international symposium on Software testing and analysis, ACM Press (2002) 65–69

32. Liblit, B.R., Liblit, B.R., Liblit, B.R.: Cooperative bug isolation. Technical report (2004)
33. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. SIGPLAN Not. **40** (June 2005) 15–26
34. Chilimbi, T.M., Nori, A.V., Liblit, B., Vaswani, K., Mehra, K.: Holmes: Effective statistical debugging via efficient path profiling. In: in 31st International Conference on Software Engineering (ICSE 2009. (2009)
35. Jin, G., Thakur, A., Liblit, B., Lu, S.: Instrumentation and sampling strategies for cooperative concurrency bug isolation. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '10, New York, NY, USA, ACM (2010) 241–255