INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Lightweight Cooperative Logging for Fault Replication in Concurrent Programs

Nuno de Ferraz Almeida e Peixoto Machado

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Doutor João Emílio Segurado Pavão Martins
Orientador:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Vogais:	Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Outubro de 2011

Acknowledgements

I wish to thank a handful of people whom I believe had contributed, even if not in a direct way, to the development of this thesis.

First of all, my advisor Professor Luís Rodrigues. His wise ideas and comments and his outstanding ability to motivate were essential for me to outstrip the many appearing obstacles and lead this work to fruition. Also, a special word for the remaining team members working on FastFix project: João Garcia, Paolo Romano, Pedro Louro, and João Matos, for the precious and enlightening discussions. Thank you all for the priceless help along this sinuous path.

Jeff Huang, for his total availability to answer my (countless) questions about LEAP.

My colleagues from the GSD group at INESC-ID, in special Xavier Vilaça, for making room 601 much more than just a workplace. Your help and encouragement were fundamental.

The Gen (youth branch of the Focolare Movement), for being my second family. I am truly grateful for having you all in my life. Thank you for the relentless faith, unconditional support, and for the deep experience that we make together, that goes beyond the simple human friendship.

Violeta, who backed me up during most of this journey. She made me counteract my innate passivity and instilled in me the desire to embrace the unknown and to seize every opportunity that life gives us. Thank you for being the catalyst for the great personal growth that I have experienced this year.

My cousin Luís Maia, for always providing me a very nice conversation (and forcing me to rest my mind) at the end of every exhausting day of work.

Finally, and most important, my parents Margarida and José, and my sister Maria. Thank you for your unwavering love and support, for always making me believe that everything would be fine, even when I doubted it. You are my alpha and omega, and make me feel the most fortunate person on earth.

This work was partially supported by FCT (INESC-ID multi-annual funding) through the PIDDAC program funds, and by the european project “FastFix” (FP7-ICT-2009-5).

Lisboa, Outubro de 2011

Nuno de Ferraz Almeida e Peixoto Machado

To those who no one ever
remembers.

Resumo

O aparecimento dos multi-processadores tornou atraente o desenvolvimento programas de software paralelos, visto que estes permitem tirar maior partido dos recursos de computação disponíveis e obter melhor desempenho. Contudo, escrever e depurar programas concorrentes têm-se revelado tarefas bastante complicadas devido à natureza não-determinística deste tipo de aplicações, isto é, correr várias vezes o mesmo programa pode levar a diferentes resultados para cada execução. A técnica de reprodução determinística resolve este problema ao providenciar a reprodução fiel da execução original. Infelizmente, o uso desta técnica traz elevados custos adicionais, pois requer a gravação de todas as fontes de não-determinismo do programa, de modo a obter o seu comportamento original.

Para abordar este problema, esta tese apresenta o CoopLEAP, um sistema que providencia a replicação de faltas de programas concorrentes, baseado em gravação cooperativa e na combinação de históricos parciais. O CoopLEAP aplica um esquema de gravação parcial para reduzir a quantidade de informação que uma dada instância do programa necessita de guardar de modo a suportar reprodução determinista. O uso de históricos parciais permite reduzir substancialmente os custos adicionais impostos pela execução do código instrumentado, mas levanta o problema de encontrar uma combinação de históricos capaz de reproduzir a falta. Esta tese também propõe uma heurística, denominada *Similarity-Guided Merge*, para efectuar esta procura. As bancadas experimentais, usadas para avaliar a implementação de um protótipo do CoopLEAP, mostram que este consegue não só reproduzir com sucesso erros de concorrência, como também impor menores custos adicionais, em comparação com outras soluções existentes.

Abstract

With the advent of multi-processors, it becomes appealing to develop parallel software programs that take full advantage of the available computing resources and achieve better performance. However, writing and debugging concurrent programs are very challenging tasks because of the non-deterministic nature of this kind of applications, i.e. running the same program several times may lead to different outcomes for each run. The deterministic replay technique addresses this problem, as it provides a faithful reproduction of the original run. Unfortunately, deterministic replay comes with very expensive overheads, since it requires recording all sources of non-determinism to achieve the original program behavior.

To address this problem, this thesis presents CoopLEAP, a system that provides fault replication of concurrent programs, based in cooperative recording and partial log combination. CoopLEAP employs a partial recording scheme to reduce the amount of information that a given program instance is required to store in order to support deterministic replay. The use of partial logs allows to substantially reduce the overhead imposed by the instrumented code execution, but raises the problem of finding the combination of logs capable of replaying the fault. This thesis also proposes an heuristic, denoted Similarity-Guided Merge, to perform this search. In-house and third-party benchmarks, used to evaluate the implemented prototype of CoopLEAP, show that it can not only successfully replay concurrency bugs, but also impose smaller overheads in comparison with other existing solutions.

Palavras Chave
Keywords

Palavras Chave

Erros de Concorrência

Reprodução Determinística

Depuração

Desempenho

Keywords

Concurrency Errors

Deterministic Replay

Debugging

Performance

Table of Contents

List of Figures

List of Tables

Acronyms

CREW	Concurrent-Reader-Exclusive-Writer
DMA	Direct Memory Access
I/O	Input/Output
SMP	Shared Memory Processor
SPE	Shared Program Element
TLO	Thread-Local Objects Analysis

1 Introduction

This thesis addresses the problem of replaying faulty executions of concurrent programs in order to help developers to remove bugs from software. For this purpose, the thesis proposes a scheme to reduce the amount of information that a given program instance is required to store to support deterministic replay.

1.1 Motivation

Software bugs continue to hamper the reliability of software. It is estimated that bugs account for 40% of system failures (Li, Tan, Wang, Lu, Zhou, & Zhai 2006), leading to huge costs both to software producers and end users (of Standards & Technology 2002). 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 (Steegmans, Bekaert, Devos, Delanote, Smeets, van Dooren, & Boydens 2004; Omar & Mohammed 1991) and the use of formal methods (Hall 2007).

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 (Parnas 2010). This problem is exacerbated 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 (Godefroid & Nagappan 2008). Since the time to fix a bug is directly related to developer's ability to reproduce it for diagnosis (Lu, Park, Seo, & Zhou 2008), any debug mechanism that is able to provide whole-system *deterministic replay* is a significant asset.

However, obtaining the faithful original execution replay may require the recording of all its relevant details (Dunlap, Lucchetti, Fetterman, & Chen 2008) (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. On the other hand, if one records too little data, it may not be sufficient to reproduce the bug. In summary, a trade-off must be made between the degree of fidelity in replay and the runtime overhead imposed by the amount of information traced.

1.2 Approach

In the past decade, a significant amount of research has been performed in order to develop efficient solutions (either based on hardware or software) that provide deterministic replay. Several of these solutions (Georges, Christiaens, Ronsse, & De Bosschere 2004; Choi & Srinivasan 1998) aim at replaying the bug on the first attempt, but this comes with an excessively high cost (10x-100x slowdown) on the original run, which is still too expensive to be practical.

As user-side executions are much more performance critical when compared to developers' in-house debugging, it is important to reduce the production run overhead, even if it results in a slightly longer bug-reproduction time during diagnosis. Motivated by this, some recent works (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009; Altekar & Stoica 2009) try to minimize the recording overhead at the cost of a greater number of replay attempts. This is achieved by first recording partial information during the original execution and then inferring the unrecorded information, on the developer side.

Despite some significant results, these solutions always try to reproduce the error relying on the data collected from one single run of the application. This constrains the reduction that can be made on the amount of partial information to be collected.

Our observation is that one can further mitigate the runtime penalties 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, analyzing and combining information recorded from different users regarding program's faulty runs, one can make bug reproduction more cost-effective. If each user collects only a fraction of the traces, the performance of its instance of the program will not be so significantly affected. 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.

Therefore, the approach followed in this thesis is the following: *i)* a lightweight mechanism to partially record relevant information from multiple executions performed by a community of users of the same program, and *ii)* strategies to merge the logs of the faulty executions and generate a replay driver that reproduces the bug.

1.3 Contributions

The contributions of the thesis are the following:

- A set of novel statistical metrics to detect correlations between partial logs, namely two metrics to measure similarity of partial logs (named Plain Similarity and Dispersion-based Similarity) and a metric to classify each partial log according to the likelihood of its reconstruction reproduce the error (denoted Relevance);
- A novel heuristic, named Similarity-Guided Merge, that leverages on the previous metrics to systematically perform a guided search, among the possible combinations of partial logs, to find those which generate complete replay drivers capable of reproducing the bug with high probability.

1.4 Results

This thesis produced the following results:

- A prototype of a lightweight deterministic replay system, named Cooperative LEAP, or simple CoopLEAP (since it is based on LEAP (Huang, Liu, & Zhang 2010), a deterministic replay software-based solution). CoopLEAP implements three low overhead recording schemes and leverages on the partial logs generated by multiple clients to deterministically reproduce concurrency bugs.

- An experimental evaluation of the implemented prototype based on a developed micro-benchmark, third-party benchmarks, and on a real-world complex server application.

1.5 Research Context

This work was performed in the context of the FastFix project[1]. One of the main goals of this project is to build a plataform for remote software maintenance, capable of monitoring execution environments and replicate application failures. During my work, I benefited from the fruitful collaboration with the remaining members of the GSD team working on FastFix, namely Paolo Romano, João Garcia, Pedro Louro and João Matos. A preliminary description of this work can be found in a paper published in INForum 2011 by Machado, Romano, & Rodrigues (2011).

1.6 Structure of the Document

The rest of this thesis is structured as follows: Chapter 2 presents some background concepts related to this work, as well as an overview of some deterministic replay and statistical debugging systems. Chapter 3 introduces CoopLEAP, describing in detail not only its architecture, but also the Similarity-Guided Merge heuristic and the metrics used to measure similarity of partial logs. Chapter 4 shows the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

[1]https://services.txt.it/fastfix-project

2 Related Work

There are various approaches to prevent bugs in a program and to optimize the debugging process. This chapter focuses on approaches that try to reproduce the failure or to statistically isolate it, as these are the most relevant to the work reported in the thesis.

The remaining of this chapter is organized as follows. Section 2.1 presents the deterministic replay approach. Section 2.2 identifies the main challenges and performance metrics in order to have a good debugging tool. Section 2.3 overviews some systems that employ the deterministic replay technique. Section 2.4 presents the statistical debugging approach and the main solutions that follow this method. Finally, Section 2.5 concludes the chapter.

2.1 Deterministic replay

Developers often employ *cyclic debugging* (Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, & Bosschere 2003) 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 do not always reveal themselves in every execution.

The problem of non-determinism can be addressed by employing an approach called *deterministic replay* (or *record/replay*) (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009). 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 (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009).

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.

2.2 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. We now present the main challenges and performance metrics that need to be considered when developing these kind of debugging systems.

2.2.1 Overhead performance

The main challenge lies in determining the level of abstraction at which the debugger 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 require 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, according to Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, & Bosschere (2003) a solution should be:

- **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.

2.2.2 Non-determinism

External factors often interfere with the program execution, preventing the timing and the sequence of instructions executed to be always identical. The sources of these factors can be divided into two types: *input non-determinism* and *memory non-determinism* (Pokam, Pereira, Danne, Yang, & Torrellas 2009). We now describe each type in more detail.

2.2.2.1 Input Non-determinism

In general, the input non-determinism encompasses all the inputs that are received by the system layer being recorded and are not produced by that layer. Therefore, one has different levels of abstraction according to the system layer being considered: user-level or system-level (Pokam, Pereira, Danne, Yang, & Torrellas 2009). In the case of user-level replay, all inputs proceeding from the operating system are not granted to be repeatable across two runs and, therefore, are considered as non-deterministic. In turn, for system-level replay, the non-deterministic inputs are those coming from external devices (I/O, DMAs, interrupts, etc). We now focus on the sources of non-determinism at each level.

For user-level replay, the following sources can hamper the deterministic replay (Patil, Pereira, Stallcup, Lueck, & Cownie 2010):

- ***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.

- ***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.

Regarding system-level replay, the sources of non-determinism are listed as follows:

- ***Inputs:*** the inputs from keyboard and network may not be the same depending on the execution.
- ***Hardware interrupts:*** the control flow of the execution changes whenever an interrupt service routine is called to handle a hardware interrupt. These interrupts are useful to notify the processor that some data (e.g. disk read) are ready to be consumed. Given that an interrupt can happen asynchronously, one needs to record the instant of time at which the interrupt arrived. Moreover, it is also necessary to log the source of the interrupt (e.g. disk I/O, network I/O, timer interrupt, etc).
- ***Direct Memory Access (DMA):*** DMA allows other hardware subsystems to access directly to the memory, independently of the processor. To achieve deterministic replay, one needs to log the values written by DMA as well as the timestamp at which they were written.

It should be noted that input non-determinism is present in both single-processor and multi-processor machines.

2.2.2.2 Memory Non-determinism

When comes to memory non-determinism in single-processor, the following reasons are sources of non-determinism:

- ***Reads of un-initialized memory locations:*** the values read from memory locations which are not explicitly initialized often change in different runs.
- ***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. This is due to interrupts being delivered at different times, as a result of differences in the

architectural state (e.g. cache line misses, memory latencies, etc.) and the load in the system.

Non-determinism arising from thread scheduling and signal delivering can be tackled by recording them with "logical time" (Lamport 1978), instead of ordinary physical time. In fact, logical time may be sufficient to support deterministic replay in single-processor systems (Srinivasan, Kandula, Andrews, & Zhou 2004). However, when one moves to the field of multi-processors (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.

2.2.3 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: *i)* 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 (Castro, Costa, & Martin 2008); *ii)* 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 (Castro, Costa, & Martin 2008; Wang, Wang, & Li 2008), 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.

2.2.4 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.

2.3 Deterministic Replay systems

Two approaches have been initially proposed to achieve deterministic replay (Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, & Bosschere 2003): 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 multi-processor replay or only uni-processor replay.

2.3.1 Hardware-based

In general, hardware-based solutions offer support for multi-processor systems. One of the first approaches in this direction was the one proposed by Bacon & Goldstein (1991), which introduces a mechanism of multi-processor 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) (Xu, Bodik, & Hill 2003) and, later, BugNet (Narayanasamy, Pokam, & Calder 2005) and DeLorean (Pablo Montesinos & Torrellas 2008).

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

Flight Data Recorder (Xu, Bodik, & Hill 2003): 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 (Netzer 1993) 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 (Sorin, Martin, Hill, & Wood 2002) 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 about 10% (Xu, Bodik, & Hill 2003). The combined sizes of logs needed for replay in FDR (it records checkpoints, interrupts and external inputs, and memory races) are about 34 MB, for the performed experiments. 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 1 GB, depending on the program's memory footprint and the size of the main memory chip used in the system.

BugNet (Narayanasamy, Pokam, & Calder 2005): 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 the deterministic re-execution of the program, 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 (Pablo Montesinos & Torrellas 2008): 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[1] (RC) execution. In contrast, FDR and BugNet only record at the speed of

[1]A system is said to provide release consistency, if all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter acquires it.

Sequential Consistency[2] (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 an average performance 2-3% lower than that of RC. With chunk sizes of 2000 instructions (the optimal size according to Pablo Montesinos & Torrellas (2008)), *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

[2]A system is said to provide sequential consistency, if every node of the system sees the (write) operations on the same memory part in the same order.

RC. This is still faster than SC, which is, on average, 21% slower than RC.

Unfortunately, despite various optimization endeavors to reduce hardware complexity (Pablo Montesinos & Torrellas 2008), all the previous approaches still demand significant hardware modifications. These modifications are not yet available nowadays, except on simulations.

2.3.2 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 multi-processor systems.

IGOR (Feldman & Brown 1988): 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 (Srinivasan, Kandula, Andrews, & Zhou 2004): 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 (Srinivasan, Kandula, Andrews, & Zhou 2004):

- *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 by Srinivasan, Kandula, Andrews, & Zhou(2004) 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 440 KB to 830 KB 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 (LeBlanc & Mellor-Crummey 1987): This system was one of the first software systems for deterministic replay on multi-processors. 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.

SMP-ReVirt (Dunlap, Lucchetti, Fetterman, & Chen 2008): This system was the first to record and replay multi-processor 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 multi-processor 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 by Dunlap, Lucchetti, Fetterman, & Chen (2008). 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.

DejaVu (Choi & Srinivasan 1998): 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: *i) critical events*, namely synchronization operations (e.g. `monitorenter` and `monitorexit)` and accesses to shared variables, and *ii) 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 it supports multi-processors, the technique used in DejaVu enforces a global order on variable accesses across multiple threads, which incurs a large runtime overhead on multi-processor 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 KB/s.

JaRec (Georges, Christiaens, Ronsse, & De Bosschere 2004): 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 drops the idea of global ordering and uses a *Lamport's clock* (Lamport 1978) 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 (Huang, Liu, & Zhang 2010): Is a recent deterministic replay system for concurrent Java programs on multi-processors. LEAP's ordering approach is based on a new type of local order with reference to the shared memory locations and concurrent threads. 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 evaluation results presented by Huang, Liu, & Zhang (Huang, Liu, & Zhang 2010) 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.

A detailed description of the LEAP system is presented in Section 3.1.

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 (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009). Motivated by this observation, 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 (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009): This system is a record/ replay technique to help reproduce bugs on multi-processors. 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, denoted 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 (Altekar & Stoica 2009): 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 (Zamfir & Candea 2010): 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 core dump 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 core dump 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).

2.4 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 (Orso, Liang, Harrold, & Lipton 2002): 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 (Liblit, Aiken, Zheng, & Jordan 2003; Liblit, Naik, Zheng, Aiken, & Jordan 2005): 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 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 *sensitivity* (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 (Chilimbi, Liblit, Mehra, Nori, & Vaswani 2009): 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 a 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 (Jin, Thakur, Liblit, & Lu 2010): 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.

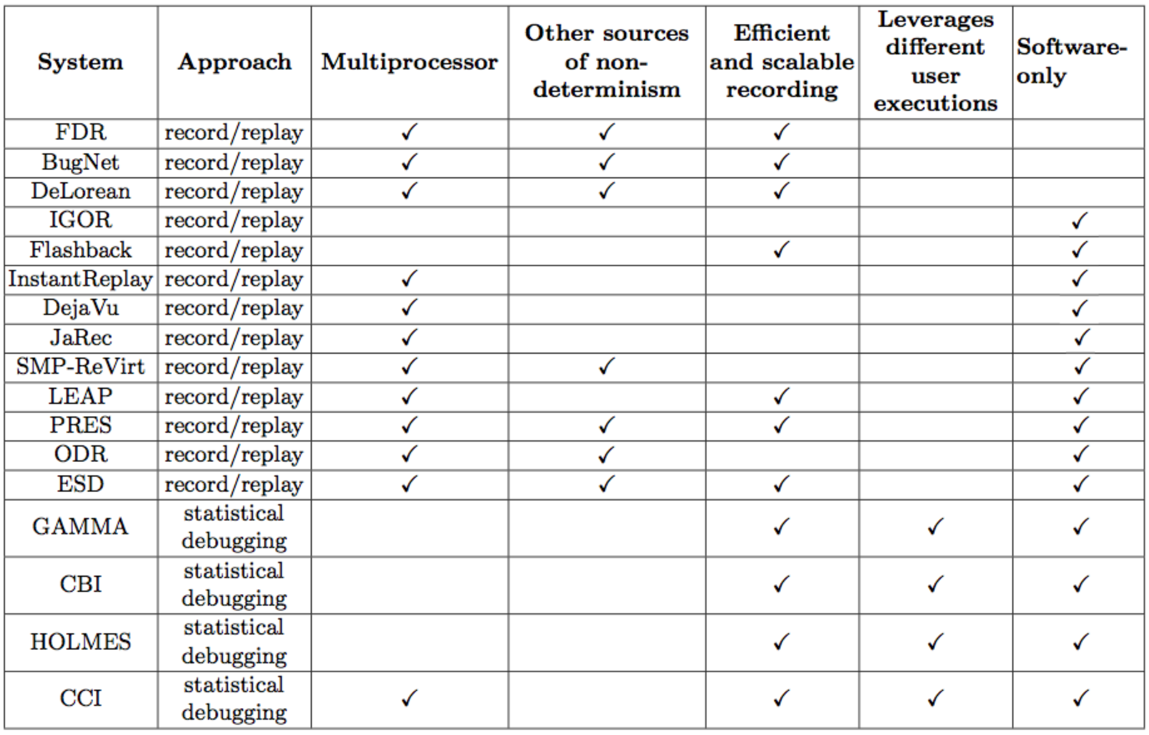
Thereby, CCI consider three different instrumentation schemes, that offer different trade-offs between performance and failure-predicting capability:

- ***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.

2.5 Summary

Figure 2.1 summarizes the deterministic replay (record/replay) and statistical debugging systems previously presented. The systems are classified according to their approach and *i)* whether they support multi-processors and other sources of non-determinism besides thread



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	✓		✓	✓	✓

Figure 2.1: Summary of the presented systems.

interleaving when accessing shared variables (e.g. interrupts, user inputs, 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 the discussed solutions log different events, use different log compression schemes, and 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%, in general). However, Flashback does not cope with concurrency issues on multi-processor. When comes to other sources of non-determinism, besides the 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 debugging systems, Figure 2.1 shows that the former are mostly better prepared to deal with bugs arising from concurrent executions in multi-processor 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 multi-processor applications, but, as CCI relies on sampling, it can miss some important clues to debug some concurrency errors.

It should be noted that, in this chapter, we have only addressed approaches that try to reproduce the failure or to statistically isolate it. Other techniques, to prevent bugs in a program, such as code analysis (Lu, Park, Hu, Ma, Jiang, Li, Popa, & Zhou 2007; Musuvathi, Qadeer, Ball, Basler, Nainar, & Neamtiu 2008) are outside the scope of this thesis. Moreover, since recording and replaying input non-determinism can be achieved with an overhead less than 10% (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009; Dunlap, Lucchetti, Fetterman, & Chen 2008; Srinivasan, Kandula, Andrews, & Zhou 2004), the work presented in this thesis only focuses on addressing memory non-determinism. In fact, a recent study on the evolution of the types of errors in MySQL database (Fonseca, Li, Singhal, & Rodrigues 2010) shows a growth trend in the number and proportion of concurrency bugs over the years. Thereby, this thesis addresses the deterministic replay of this kind of bugs (e.g. atomicity violations, data races), disregarding other sources of non-determinism.

The following chapter introduces CoopLEAP, a novel deterministic replay system based on partial logging and on statistical debugging principles. The statistical metrics and the heuristic used to combine partial logs are also presented.

3 CoopLEAP System

This chapter introduces CoopLEAP, a system that provides fault replication of concurrent programs, based in cooperative recording and partial log combination. Given that CoopLEAP is based on LEAP, it shares most of its features and its main components. Hence, this chapter starts with Section 3.1 presenting LEAP in detail. The chapter follows with a description of the CoopLEAP architecture and its main components, in Section 3.2. Then, Sections 3.3 and 3.4 describe the strategies to store partial logs during production runs and how to merge them in order to generate a complete replay log that reproduces the bug.

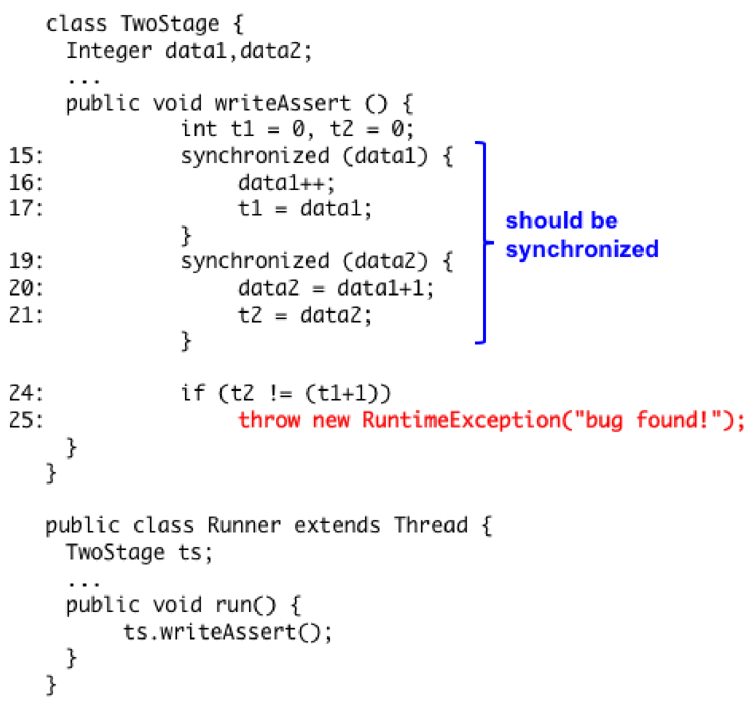
3.1 Standard LEAP

LEAP (Huang, Liu, & Zhang 2010) proposes a general technique for the deterministic replay of concurrent programs in multi-processors. It is based on the insight that, to reproduce the execution, it is sufficient for each shared variable to track the order of the thread accesses it sees during production run. In this section, a description of the LEAP system is provided.

3.1.1 Local-order Deterministic Replay

As noted before, instead of enforcing a global order of thread accesses to shared memory locations, LEAP relies on a new type of local order with reference to the program shared variables and concurrent threads. 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, its 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.

In order to better understand this technique and state its differences with the conventional global-order based approach, let us present an example. Figure 3.1 shows a code snippet with



```
     class TwoStage {
       Integer data1,data2;
       ...
       public void writeAssert () {
                  int t1 = 0, t2 = 0;
15:               synchronized (data1) {
16:                    data1++;
17:                    t1 = data1;
                  }
19:               synchronized (data2) {
20:                    data2 = data1+1;
21:                    t2 = data2;
                  }

24:               if (t2 != (t1+1))
25:                    throw new RuntimeException("bug found!");
       }
     }

     public class Runner extends Thread {
       TwoStage ts;
       ...
       public void run() {
            ts.writeAssert();
       }
     }
```

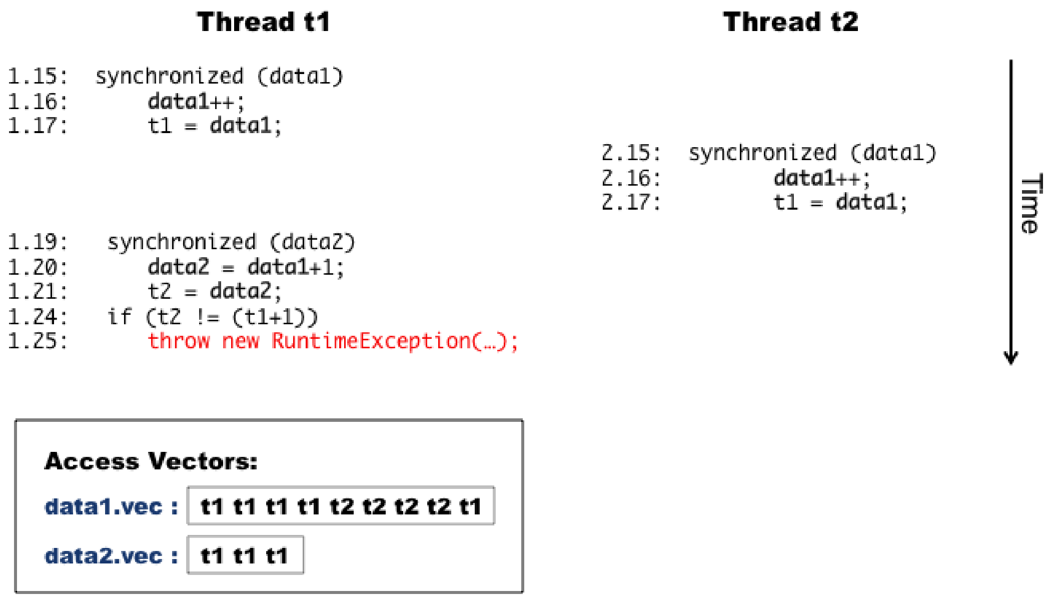
Figure 3.1: Example of code containing a two-stage bug.

races that may lead to an error, namely a two-stage access bug (Farchi, Nir, & Ur 2003). Operations to both `data1` and `data2` variables need to be protected because they can be concurrently accessed by different threads. However, it was wrongly assumed that separately protecting each operation is enough. In fact, both blocks of operations at lines 16-17 and 20-21 are synchronized, but a context switch may occur between the access to first and the second blocks and the value stored in `data1` may be changed by another thread. Hence, the condition at line 24 may be verified and an exception thrown.

Figure 3.2 illustrates a thread interleaving that triggers the bug. Assuming that, initially, `data1 = 0`, thread $t1$ will have its local variable `t1 = 1` at line 1.17. However, given that thread $t2$ increments `data1` at line 2.16, when thread $t1$ executes line 1.21, it will have `t2 = 3`. Therefore, an exception will be thrown at line 1.25.

A global-based technique would require 13 global synchronization[1] operations to record this schedule and replay the program. In turn, instead of a global vector, the local-order approach uses two access vectors (`data1.vec` and `data2.vec`) for the shared variables `data1` and `data2`, which record `<t1,t1,t1,t1,t2,t2,t2,t2,t1>` and `<t1,t1,t1>`, respectively. Therefore, this

[1]Each ++ operation is non-atomic, therefore is equivalent to two single operations.



Thread t1

```
1.15:  synchronized (data1)
1.16:      data1++;
1.17:      t1 = data1;

1.19:   synchronized (data2)
1.20:      data2 = data1+1;
1.21:      t2 = data2;
1.24:   if (t2 != (t1+1))
1.25:      throw new RuntimeException(…);
```

Thread t2

```
2.15:  synchronized (data1)
2.16:          data1++;
2.17:          t1 = data1;
```

Access Vectors:

```
data1.vec : t1 t1 t1 t1 t2 t2 t2 t2 t1
data2.vec : t1 t1 t1
```

Figure 3.2: Execution interleaving that triggers the bug.

technique requires no global synchronization operations and 12 local synchronization operations distributed for two groups that can be traced in parallel.

By losing global order enforcement, local-order recording relaxes faithfulness in the replay, allowing thread interleavings that are different from the original execution. For instance, in the example, line 1.19 could be executed before line 2.15. However, in Huang, Liu, & Zhang (2010) the authors claim that using this approach does not affect the error reproduction, and they formally prove the soundness of this statement.

3.1.2 Locating Shared Variable Accesses

To locate shared variables, LEAP uses a static escape analysis called *thread-local objects analysis* (TLO) (Halpert, Pickett, & Verbrugge 2007) from the Soot[2] framework. TLO classifies any field as thread-shared whenever it is possibly accessed by more than one thread at a time, otherwise it is considered as thread-local.

Given that accurately identifying shared variables is generally an undecidable problem, this technique computes a sound over-approximation, i.e. each shared access to a field is indeed identified, but some accesses which are actually not may also be classified as shared (Bodden &

[2]http://www.sable.mcgill.ca/soot

Havelund 2008). Despite that, LEAP authors have also proved that this over-approximation do not affect the correctness of the deterministic replay.

Another issue of TLO is its inability to distinguish between read and write accesses, therefore shared variables whose values never change after initialization are also considered. For these reason, LEAP further refines this technique to avoid recording accesses to shared immutable variables.

3.1.3 Variable Identification Across Executions

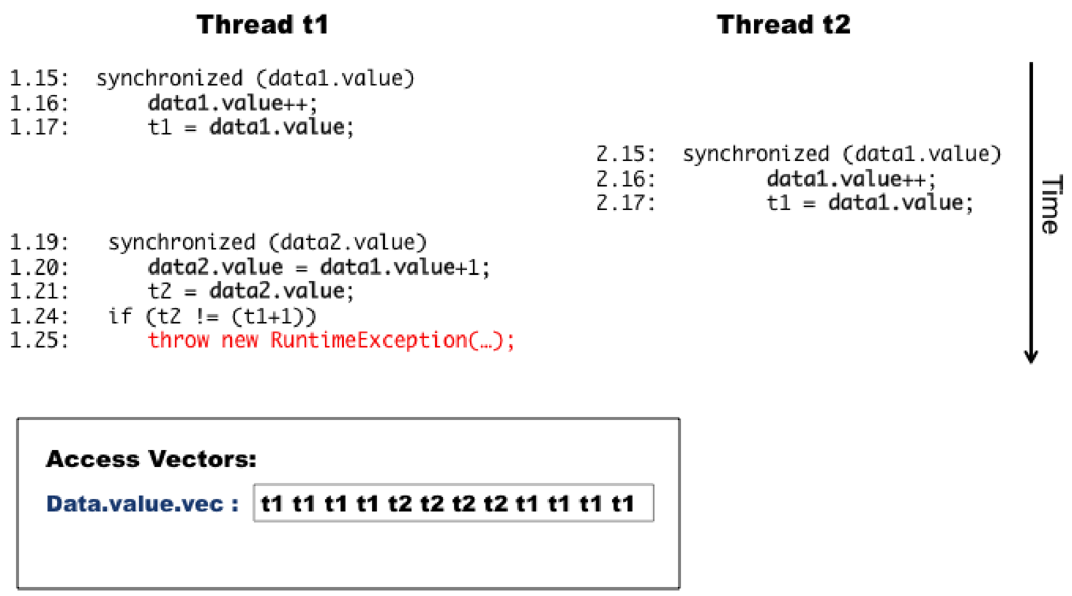
Given that the standard JVMs are not able to consistently identify objects across different runs, LEAP assigns offline a numerical index to each *shared program element* (SPE). Are considered as SPEs variables that serve as monitors (including Java monitors) and other shared field variables (including class and thread escaped instance variables). For example, in Figure 3.1, the two shared field variables `data1` and `data2` of the `TwoStage` class are assigned the numerical IDs 1 and 2.

The static field-based shared variable identification, besides remaining consistent across runs and imposing no runtime degradations, has the advantage of being more fine-grained than other object level identification approaches (LeBlanc & Mellor-Crummey 1987), because different fields of the same object are mapped to different indexes. As a result, one avoids the runtime serialization of accesses to different fields of the same object.

Unfortunately, this approach brings also some drawbacks, as referred below:

- Different instances of the same type are not statically distinguished. Consequently, the same access vector englobes all accesses performed to the same shared field variable of different instances of the same type. To illustrate this phenomenon, consider that the variables `data1` and `data2` (in Figure 3.1) instead of being of the type `Integer`, were instances of the following class `Data`:

```
public class Data {
    public int value;
    public Data(){value = 0;}
 }
```



```
1.15:  synchronized (data1.value)
1.16:      data1.value++;
1.17:      t1 = data1.value;
                                        2.15:  synchronized (data1.value)
                                        2.16:          data1.value++;
                                        2.17:          t1 = data1.value;

1.19:   synchronized (data2.value)
1.20:      data2.value = data1.value+1;
1.21:      t2 = data2.value;
1.24:   if (t2 != (t1+1))
1.25:      throw new RuntimeException(…);
```

Figure 3.3: Execution interleaving that triggers the bug when shared variables data1 and data2 are instances of a class Data.

Thereby, class `TwoStage` would now declare the two variables like `Data data1,data2` and execute the buggy interleaving as depicted in Figure 3.3. As one can see, all the accesses to both variables `data1.value` and `data2.value` are now recorded globally into the same access vector (`Data.value.vec`). Albeit this brings a slightly worse performance, it does not affect the correctness of the deterministic replay (Huang, Liu, & Zhang 2010).

- Scalar variables that are alias to shared array variables are not uniquely identified. To handle this issue, LEAP does an alias analysis for all of the scalar array variables in the program and uses the same SPE for all the aliases, neglecting the indexing operations. This solution decreases the degree of concurrency, but makes sure that nondeterminism caused by array aliases is correctly tracked.

- Different index positions of an array are not uniquely identified. In fact, this is an obvious consequence that follows from the usage of the an static analysis scheme. For instance, let us consider an array `arr` and two positions `arr[x]` and `arr[y]`. Basically, both positions address the same shared variable if `x == y` and this is not trivial to decide statically. As a consequence, independent shared array positions that could be recorded in separate, are logged into a single access vector.

3.1.4 Unique Thread Identification

As access vectors only contain thread IDs tracked during the production run, it is imperative to correctly recognize each thread in both recording and replay phases. LEAP achieves this by maintaining a mapping between the thread name and the thread ID during recording and using the same mapping for replay.

3.1.5 Handling Early Replay Termination

As mentioned earlier in Section 3.1.1, LEAP allows different global thread interleavings, as long as their individual state remains the same. This has the disadvantage of allowing early termination scenarios, i.e. a crash failure might be manifested before all SPE accesses are replayed. To deal with this problem, LEAP ensures that all the thread recorded actions are performed before the ending of the replay execution, which guarantees that its final state and the one seen in the original execution are identical.

3.1.6 Architecture

3.1.6.1 Overview

The overall infrastructure of LEAP, depicted in Figure 3.4, consists of three major components: the *transformer*, the *recorder*, and the *replayer*.

The transformer receives the Java program bytecode and employs two types of instrumentation schemes to produce the *record version* and the *replay version*, respectively.

The record version is then executed and the recorder component stores the accesses to each SPE in its correspondent access vector. When the production run ends, LEAP generates three different files: the access vectors, the thread ID map information, and the 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 and consults the thread ID map information file.

Each one of the main components is described with more detail in the following sections.

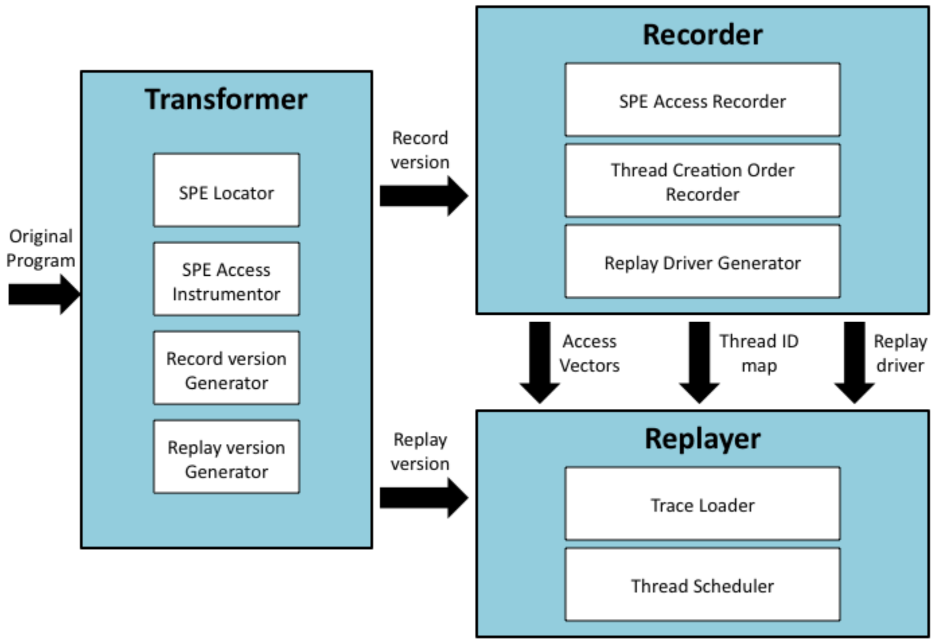


Figure 3.4: Overview of the LEAP architecture (adapted from the LEAP paper).

3.1.6.2 Transformer

The transformer module is responsible for instrumenting the Java bytecode and it is implemented with support of the Soot framework. Soot provides four intermediate representations for code: Baf, Jimple, Shimple and Grimp, each one corresponding to different levels of abstraction on the represented code. LEAP uses Jimple, which is a typed, three-address[3], statement based intermediate representation. Thereby, after the SPE locator identifies all the SPEs in the code, the transformer iterates over each Jimple statement instrumenting: *i)* SPE accesses, *ii)* thread creation information, and *iii)* recording end points.

In the first case, an API `accessSPE(int speIndex, long threadId)` call is placed before the Jimple statement to log both the thread ID and the SPE index. To guarantee that the correct SPE thread accessing order is recorded, LEAP uses a SPE-specific lock to encompass both the API invocation and the SPE access. Figure 3.5 illustrates this instrumentation scheme applied to line 17 from the example code in Figure 3.1 (`data1` index is considered to be 1 as in Section 3.1.3).

[3]Three-address code is a form of representing intermediate code, where each instruction has the form: *result := operand1 operator operand2* (three addresses). This allows every instruction to implement exactly one fundamental operation.

```
      thread_id = getThreadId();
      get_lock(1);
      accessSPE(1,thread_id);
1.17: t1 = data1;
      release_lock(1);
```

Figure 3.5: The instrumentation of SPE accesses.

In the particular case when the SPE is a Java monitor object, the monitoring invocation is inserted after the `monitorentry` and before the `monitorexit` operations. In addition, the call is also inserted before `notify/notifyAll/Thread.start` instructions and after `wait/Thread.join` instructions. If a thread accesses to SPEs multiple times in a method, its ID needs only to be captured once. Another optimization is do not instrument those SPEs whose accesses are always wrapped by the same monitor.

Regarding thread initialization, the thread identity information is recorded by instrumenting the `Thread` constructor with an API `threadStartRun(long threadId)` invocation, allowing LEAP to track the thread creation order, as described in Section 3.1.4.

Finally, LEAP instruments recording end points to inform the recorder to save the traced runtime information and to create the replay driver. Currently, three types of recording end points are provided. The first consists of adding a `ShutdownHook` to the JVM Runtime in order to perform the saving operations at the end of the program execution. The second is a try-catch block placed into the `main` thread and the `run` method of each Java Runnable class, followed by a `crashed(Throwable crashedException)` method invocation inside the catch block. This allows uncaught runtime exceptions to be also recording end points. Lastly, it also possible for users to do specific annotations of end points. In this case, during Jimple statements traverse, the transformer will replace the annotation with an API monitoring call, marking the end of recording.

All the above steps belong to the record version generation process. To create the replay version, the process is almost the same, with the following differences: *i)* the `accessSPE()` invocation has to be inserted *before* `monitorenter` and `wait` instructions, to avoid a deadlock with the order of synchronization operations enforced by the LEAP replayer; *ii)* additional API calls are inserted after each SPE access to check whether a thread has performed the total number of SPE accesses recorded in the original execution or not, as mentioned in Section 3.1.5.

3.1.6.3 Recorder

The recorder is responsible for monitoring the execution of the record version of the program. Every time the `accessSPE()` method is called, the recorder stores the executing thread ID into the access vector of the SPE. For this purpose, it is used a `Vector<Long>` array, named `accessVector`, where each position contains the access vector for the SPE with the correspondent index (e.g. accesses performed to the SPE 1 will be stored in the vector given by `accessVector[1]`).

Alongside, whenever a new thread is created, the thread information is added to the data structure that maps the thread name to the thread ID.

When a program end point is reached, the recorder saves both the recorded access vectors and the thread ID map data structures. In addition, the recorder also creates the replay driver, i.e. a Java file containing the code needed to execute the replay version of the program and initiate both the thread scheduler and the trace loader components.

3.1.6.4 Replayer

The replayer uses the replay driver to, first, start both the trace loader and the scheduler and, then, execute the instrumented version of the program targeted for replay.

Once started, the trace loader reads the files containing the saved access vectors and the thread ID map. This information is then used by the scheduler to control the thread execution interleaving and enforce a deterministic replay. The scheduler also assigns to each thread a semaphore maintained in a global data structure, in order to allow the thread suspension and resume whenever required.

Regarding SPE accesses, despite being also instrumented with an `accessSPE()` invocation, the implementation of the method differs from that of the recording phase. In the replayer, this method checks whether the thread should access the SPE or not, according to the order stablished in the SPE access vector. Thereby, if a thread is not allowed to access the SPE it is suspended. A thread also counts the number of times it has already accessed each SPE, and suspends itself if finds that it has already performed all the accesses recorded in the access vector, waiting for all other threads to end their execution.

3.2 CoopLEAP Architecture

CoopLEAP extends LEAP to support partial log combination. Although CoopLEAP shares most of its predecessor's components, its overall architecture is different. In this section, we focus on the extensions of CoopLEAP, describing its new features and components.

3.2.1 Overview

Figure 3.6 illustrates the overall architecture of CoopLEAP. During the instrumentation phase (Figure 3.6 -1), the *transformer* instruments the Java program bytecode to generate both the record version and the replay version, as done in LEAP. The nuance resides on the record version, where only a subset of all SPEs is actually instrumented (as this version will be further called as *partial record version*). It should also be noted that each client is assigned a different subset of SPEs to record, according to some defined criterion (see Section 3.3). The partial record versions are then sent to the clients, whereas the replay version is sent to the replayer.

Figure 3.6 -2 illustrates the record and replay phases. In CoopLEAP, there is a *recorder* module for each client. The recorder has exactly the same purpose as on LEAP, i.e. is responsible for collecting the access vectors for each SPE during the execution of the program. However, unlike the latter, CoopLEAP does not intend to record all SPEs' access vectors. Instead, each user logs accesses only to a part of the program's SPEs, as previously defined by the transformer.

Assuming that the program is executed by a large population of users, this mechanism allows to gather access vectors for the whole set of SPEs with high probability. By doing this, CoopLEAP aims at minimizing the performance overhead that would be required if one had to record all the access vectors at each client.

When the production run ends, each client sends its *partial log* to the developer site to be analyzed. This log consists of the access vectors recorded for a subset of the SPEs, the thread ID map, and also an additional bit indicating the success or failure of the execution (successful executions can be useful for the statistical analysis).

Here, the *statistical analyzer* will employ an heuristic (see Section 3.5) to explore and merge the received access vectors in order to generate a complete log of the faulty execution[4]. By

[4]In this work, we are assuming that all the partial logs refer to the same bug. Despite that, for different bugs

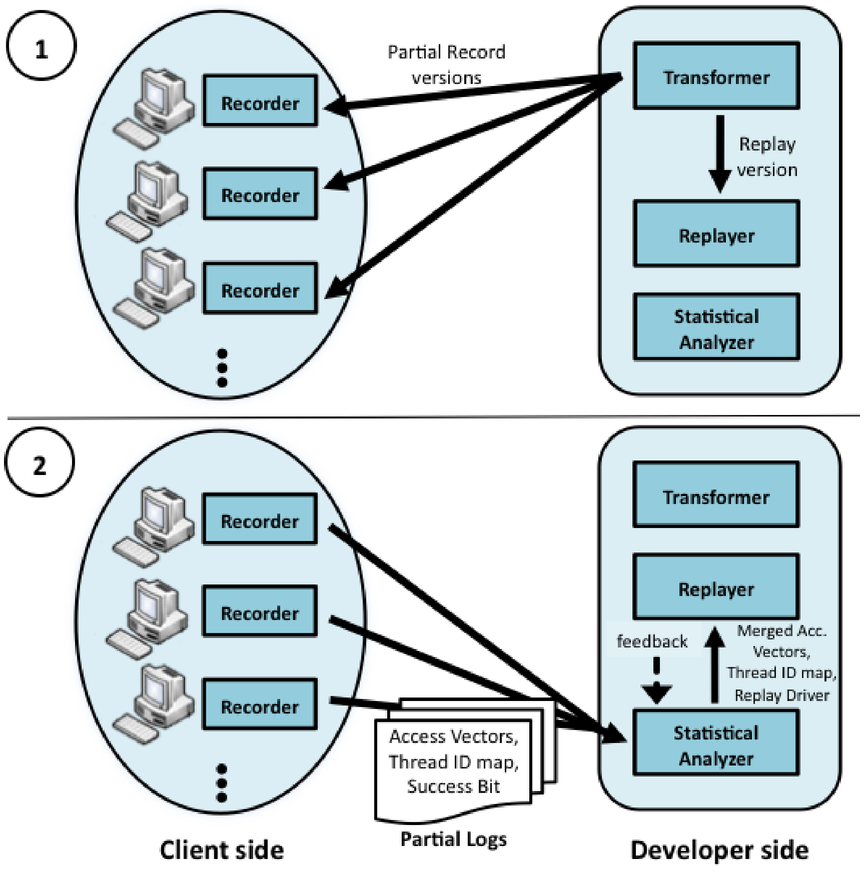


Figure 3.6: Overview of the CoopLEAP architecture: (1) Instrumentation phase; (2) Record and Replay phases.

pinpointing the most correlated partial logs, CoopLEAP plans to ease the task of inferring the original unrecorded access vectors.

Once the merge of partial logs is complete, the combination of access vectors is sent to the *replayer*, along with the thread ID map and the generated replay driver.

Finally, just like in LEAP, the replay driver will serve as an entry point for the replayer to control the replaying of the program execution. However, in CoopLEAP the replay phase is slightly different. Given that access vectors come from independent executions, the combined information can be incompatible. As a result, the execution replay will fail and the bug will not be reproduced. In this case, the replayer will send *feedback* to the statical analyzer communicating the replay failure, so the latter can investigate another access vector combination and produce a new complete log for replay. This process ends when the bug is successfully replayed

on the same program, some additional data could be used for distinction purposes, namely the line of code where the bug appeared.

or when the maximum number of attempts to do it is reached.

3.2.2 Statistical Analyzer

As previously referred, the statistical analyzer is responsible for analyzing and combining the information sent by the clients. Figure 3.7 depicts its internal structure.

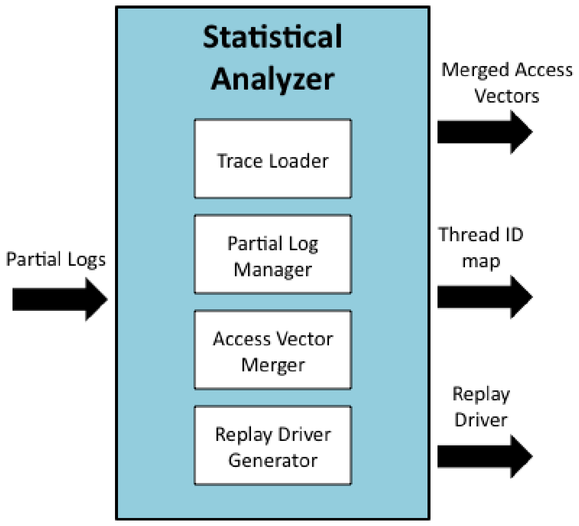


Figure 3.7: Statistical Analyzer component.

Once received, the access vectors are first loaded by the trace loader and then organized by the partial log manager. To facilitate the statistical analysis, the partial log manager hashes (using the Java `hashCode()` method) the information contained in the access vectors and builds the following control data structures:

- `PartialLogProfiles` - contains the loaded information regarding each partial log, with respect to its success bit and its SPEs' access vectors.
- `SPEProfiles` - maps each SPE to a list containing all its different access vectors recorded in the partial logs.
- `MatchLogs` - holds a profile referent to the comparison between each two partial logs, indicating their SPEs with equal access vectors and their degree of similarity.
- `StatisticalIndicators` - stores the values of the statistical metrics for each access vector (see Section 3.4.2).

Afterwards, the access vector merger module leverages on the above data structures to start the execution of the heuristic for merging the partial logs. Once the most correlated partial logs are located, their access vectors are merged and sent to the replayer in order to start the bug replay. Alongside, the generated replay driver and the thread ID map are also sent.

Every time the reproduction fails, the statistical analyzer proceeds with the heuristic and generates both a new access vector combination and a new replay driver. This process is carried on until the replayer reproduces the bug[5] or exceeds the stipulated maximum number of attempts.

3.3 Partial Log Recording

CoopLEAP introduces the novelty of recording the accesses to only a fraction of the entire set of the SPEs of the program. The subset of SPEs to be traced is defined at instrumentation time by the transformer. For this purpose, different criteria can be used, e.g. random selection, load balancing distribution, etc. However, in this work, we only consider the random selection of a certain percentage of the total number of SPEs of the program for partial recording. For this, CoopLEAP uses a static variable denoted `COVERAGE` to bound the percentage of the total number of SPEs to be instrumented. This variable consists of a `double`, whose value is the range [0,1].

Later, during the thread-local object analysis (see Section 3.1.2), whenever a new SPE is identified, a random number in the interval [0,1] is generated and compared to the coverage threshold. If the random number is less or equal to `COVERAGE`, then it is placed in a `toRecord` list, otherwise it is added to a `notToRecord` list. This mechanism allows to know whether a future reference to some SPE is to be instrumented or not, in case it already belongs to the `toRecord` list or to the `notToRecord` list, respectively.

It should be noted that this scheme is statistically fair, assuming that there is a significant number of users running the program. However, it is not granted to be optimal, as it does not always allow partial log overlapping, i.e. it may not exist SPEs in common for each two potential similar partial logs. For instance, let us consider a program with eight SPEs

[5]A bug is known to be reproduced by catching the thrown exception or by matching the produced output to that of the original faulty execution.

(`speIndex = {0..7}`) and a coverage of 50%. It could happen that two partial logs (l_1 and l_2) collected from two identical executions may not be considered as similar, since l_1 may only record the subset `{0..3}` and l_2 may only record the subset `{4..7}`, for example. This could be addressed by increasing the percentage of coverage (at the cost of greater overheads), or by defining a smaller fixed subset of SPEs to be logged by all users, thus always allowing the existence of points of comparison between the partial logs.

Moreover, one can also note that the overhead reductions may not be linear with the decrease of the coverage percentage. The reason is because some SPEs could be accessed more times than others, therefore, when instrumenting the code, the load balance may not be equally distributed among the users. A solution for this could be instrumenting the whole program and execute it one time to measure the number of accesses performed on each SPE, at the developer side. Later, when instrumenting the user versions, CoopLEAP could already take into account the SPEs workload.

The investigation of new partial recording schemes is scheduled as future work.

3.4 Merge of Partial Logs

The major challenge of our approach to provide low-overhead deterministic replay is *how to combine the collected partial logs in such a way that the access vectors used lead to a feasible thread interleaving during the replay.* In addition, that thread interleaving has to be also capable of reproducing the bug observed in the original execution.

In general, the following observations make the partial log merging difficult:

- the bug can be the result of several different thread interleavings;
- the probability of obtaining two identical executions of the same program can be very low (this probability is inversely proportional to the complexity of the program in terms of number of SPEs and the number of thread accesses);
- the combination of access vectors from partial logs of faulty executions may enforce a thread order that leads to a non-faulty replay execution;

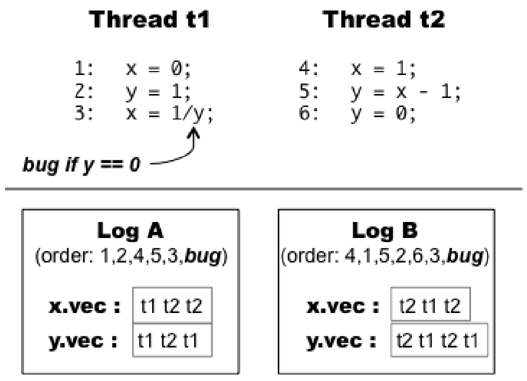


Figure 3.8: Example of a buggy program and two different logs that lead to a "division by zero" bug.

- the combination of access vectors from partial logs of faulty executions may enforce a thread order that leads to a impossible replay execution;

To better understand the case mentioned above where the combination of two faulty execution partial logs can lead to a non-faulty replay execution, let us present an example. Figure 3.8 illustrates a code snippet of a "naive" program that has a *division by zero* bug and shows two possible logs of executions that trigger the error (*LogA* and *LogB*).

Notice that both logs are complete, but let us consider a partial recording scenario where *LogA* contains only `y.vec` and *LogB* has `x.vec` alone. By merging the two access vectors, one gets a complete log with `x.vec = <t2,t1,t2>` and `y.vec = <t1,t2,t1>` that leads to the following execution order: `4,1,2,5,3,6`, where the program ends with success.

In addition, despite being a simple program, one can also see that there are other possible buggy execution orders (e.g. `1,4,2,5,6,3` and `4,5,1,2,6,3`) which generate different trace logs (`{x.vec = <t1,t2,t2,t1>, y.vec = <t1,t2,t2,t1>}` and `{x.vec = <t2,t2,t1,t1>, y.vec = <t2,t1,t2,t1>}`, respectively). This supports the claim that a bug can be result of several different thread interleavings.

Although the example used does not present an impossible execution replay situation, it can happen that the combination of access vectors lead to a deadlock scenario. Here, each thread wants to access a SPE whose access vector's first position contains the ID of the other thread. As a consequence, the threads block and wait for each other indefinitely. For Figure 3.8, an example could be `t1` wanting to access `x.vec = <t2,...>` and `t2` wanting to access `y.vec = <t1,...>`

at the same time.

Our approach to address these challenges and mitigate the incompatibility of the merged access vectors consists of employing some statistical metrics over the universe of collected partial logs to pick those which present more similarity. Our statistical metrics are divided in two types: *statistical metrics for partial log correlation* and *statistical metrics for bug correlation.*

The following sections begin by describing both types of statistical metrics and, then, present the *Similarity-Guided Merge* heuristic that systematically produces combinations of access vectors until the bug has been replayed.

3.4.1 Statistical Metrics for Partial Log Correlation

This section discusses the statistical metrics for partial log correlation, which are related to the partial logs as a whole and measure the amount of information that they have in common, so one can increase the probability of merging compatible access vectors.

In particular, the following statistical metrics are used to calculate the partial log correlation: *Similarity* and *Relevance.* Both metrics are described in detail below.

3.4.1.1 Similarity

The rationale behind the classification of the similarity between two partial logs is related to their number of SPEs with identical access vectors[6]. Hence, the more SPEs with equal access vectors the partial logs have, the better. The computation of this metric can come in two flavors: *Plain Similarity* and *Dispersion-based Similarity*, according to the weight given to the SPEs of the program.

To better define these metrics, let us first present some formal notation (see Table 3.1). With this, we can now define the metrics as follows:

Plain Similarity Let l_1 and l_2 be two partial logs, their *plain similarity* is given by the following equation:

$$PlainSimilarity(l_1, l_2) = \frac{\#Equal_{l_1,l_2}}{\#\mathcal{S}} \times \left(1 - \frac{\#Diff_{l_1,l_2}}{\#\mathcal{S}}\right) \tag{3.1}$$

[6]An access vector is considered to be identical to one another if both had recorded exactly the same thread interleaving.

Notation	**Description**
$\mathcal{S}$	Set of all the SPE identifiers of the program.
$\mathcal{S}_l$	Set of the SPE identifiers recorded only by the partial log l.
$\mathcal{AV}$	Set of the different hashes of the access vectors recorded by all the partial logs.
$\mathcal{AV}_l$	Set of the hashes of the access vectors recorded only by the partial log l.
$avecs(s) : \mathcal{S} \rightarrow \{\mathcal{AV}_1, \mathcal{AV}_2, ..., \mathcal{AV}_n\}$	Map that, for a given SPE identifier s, returns the set of the hashes of its different access vectors, recorded by all the partial logs.
$avec_l(s) : \mathcal{S}_l \mapsto \mathcal{AV}_l$	Function that maps a SPE identifier s to the hash of its access vector, recorded by the partial log l.
$Equal_{l_1,l_2} = \{s \mid s \in \mathcal{S}_{l_1} \cap \mathcal{S}_{l_2} \wedge avec_{l_1}(s) = avec_{l_2}(s)\}$	Set of the SPE identifiers, recorded by both partial logs l_1 and l_2, with identical access vectors.
$Diff_{l_1,l_2} = \{s \mid s \in \mathcal{S}_{l_1} \cap \mathcal{S}_{l_2} \wedge avec_{l_1}(s) \neq avec_{l_2}(s)\}$	Set of the SPE identifiers, recorded by both partial logs l_1 and l_2, with different access vectors.
$Sim_{l_0} = \{l_1, l_2, ..., l_k\}$	Set of the k partial logs more similar to l_0 (denoted as *group of similars of* l_0).
$Fill_{l_0,Sim_{l_0}} = \{s \mid s \in \mathcal{S}_{l_0} \cup \mathcal{S}_{l_1} \cup \mathcal{S}_{l_2} \cup ... \cup \mathcal{S}_{l_k} \wedge l_1, l_2, ..., l_k \in Sim_{l_0}\}$	Union of the sets of the SPE identifiers recorded by the partial log l_0 and by the partial logs of its group of similars Sim_{l_0}.

Table 3.1: Formal notation used to define the statistical metrics.

where $\#Equal_{l_1,l_2}$, $\#\mathcal{S}$, and $\#Diff_{l_1,l_2}$ denote the cardinality of the sets $Equal_{l_1,l_2}$, $\mathcal{S}$, and $Diff_{l_1,l_2}$, respectively.

It should be noted that this metric will only be 1 when both logs are complete and identical, i.e. they have recorded access vectors for all the SPEs of the program ($\mathcal{S}_{l_1} = \mathcal{S}_{l_2} = \mathcal{S}$) and those access vectors are equal for both logs ($avec_{l_1}(s) = avec_{l_2}(s), \forall s \in \mathcal{S}$). This implies that, for every two partial logs, their plain similarity will always be less than 1. However the greater this value is, the more probable is that the both partial logs come from the same production run.

Dispersion-based Similarity Let l_1 and l_2 be two partial logs, their *dispersion-based simi-*

larity is given by the following equation:

$$DispersionSimilarity(l_1, l_2) = \sum_{x \in Equal_{l_1,l_2}} weight(x) \times \left(1 - \sum_{y \in Diff_{l_1,l_2}} weight(y)\right) \quad (3.2)$$

where $weight(s)$ is a function of type $\mathcal{S} \rightarrow Double$ that maps each SPE identifier to a double value referent to its relative weight, in terms of *overall-dispersion*. Here, the overall-dispersion of a given SPE corresponds to the proportion of its different access vectors when compared to the total number of different access vectors collected for all the SPEs. Thereby, the weight function of a SPE identifier s can be calculated as follows:

$$weight(s) = \frac{\#avecs(s)}{\#\mathcal{AV}} \quad (3.3)$$

Notice that some other metrics could be defined if one consider other types of weights (e.g. the average number of accesses recorded for each SPE), but in this work we only use overall-dispersion.

Comparing the two metrics, one can see that the Plain Similarity considers that every SPE has the same importance, whilst the Dispersion-based Similarity assigns different weights to the SPEs. In general, both metrics allow to pinpoint the most similar partial logs, but the first is more useful when the overall-dispersion weight values are relatively well distributed for all the SPEs. On the other hand, the Dispersion-based Similarity is more suitable for cases when the access vector overall-dispersion weight is very high only for a small subset of SPEs of the program or when there are many SPEs whose access vectors are identical in every execution.

3.4.1.2 Relevance

It is easier to deterministically replay an execution if one starts from a base. Therefore, the *Relevance* metric allows to classify each partial log according to its likelihood of being completed with compatible information:

$$
\begin{aligned}
Relevance(l_0) = \alpha \times \frac{\#Fill_{l_0,Sim_{l_0}}}{\#\mathcal{S}} \\
+ (1-\alpha) \times \frac{\sum_{n=1}^{k} Similarity(l_0, l_n)}{k}, l_n \in Sim_{l_0}
\end{aligned}
\tag{3.4}
$$

where $Similarity(l_0, l_n)$ is one of the two possible types of Similarity metrics.

As one can see, the Relevance metric is the sum of two parcels with different importance (given by α). The first is related to the number of SPEs that is possible to fill joining the access vectors from the partial log l_0 and its group of similars Sim_{l_0}. This follows the rationale that the more missing SPEs of l_0 that can be filled with access vectors from similar partial logs, the better.

In turn, the second parcel gives the similarity ratio of all the partial logs in the group. This allows to pick, as the base partial log, the one whose group of similars is composed by partial logs with high similarity, thus increasing the probability of merging compatible information.

It should be noted that the maximum size k of the group of similars can be defined by the developer. Moreover, a partial log l_1 can only be part of Sim_{l_0} if $Similarity(l_0, l_1) \geq threshold$. This avoids the group of similars to be composed by partial logs with a very low value of similarity.

In our experiments, we found $\alpha = 0.7$, $k = 5$, $threshold = 0.3$ for Plain Similarity, and $threshold = 0.01$ for Dispersion-based Similarity, to be good values.

3.4.2 Statistical Metrics for Bug Correlation

Unlike the previous metrics, the statistical metrics for bug correlation are concerned with the correlation between the bug and each access vector individually, i.e. for the universe of access vectors collected for each SPE, these statistical metrics help to find the access vector which accounts for a greater number of failed executions. This also leverages information from successful executions and is specially useful when, even after merging the partial logs, there are still SPEs to be completed.

To compute these metrics, we adapt the scoring method proposed by Liblit et al (see CBI in Section 2). Thereby, access vectors are classified based on their *Sensitivity* and *Specificity*, i.e. whether they account for many failed runs and few successful runs. With this information, it is possible to define a third metric, denoted *Importance*, which identifies the access vectors that are simultaneously high sensitive and specific.

Let F_{total} be the total number of partial logs resulting from failed executions; for each access vector v, let $F(v)$ be the number of failed partial logs that have recorded v for a given SPE, and $S(v)$ be the number of successful partial logs that have recorded v for a given SPE. The three metrics are then calculated as follows.

$$Sensitivity(v) = \frac{F(v)}{F_{total}} \tag{3.5}$$

$$Specificity(v) = \frac{F(v)}{S(v) + F(v)} \tag{3.6}$$

$$Importance(v) = \frac{2}{\frac{1}{Sensitivity(v)} + \frac{1}{Specificity(v)}} \tag{3.7}$$

In summary, the higher the Importance value, the more correlated with the bug is the access vector.

3.5 Similarity-Guided Merge

To merge the partial logs and generate a complete log capable of replaying the faulty execution, we developed a heuristic denoted *Similarity-Guided Merge*. This section starts by describing the heuristic and then presents a case study to illustrate how it operates.

3.5.1 Algorithm

The Similarity-Guided Merge heuristic leverages on the statistical metrics discussed in Sections 3.4.1 and 3.4.2 to combine the information recorded by the partial logs with a high probability of compatibility. This heuristic operates in five steps, as described as follows:

1. ***Calculate the degree of similarity between the partial logs*** – the first step consists of calculating the similarity between each partial log and all the others from the universe

of partial logs received. To calculate the similarity, CoopLEAP applies the Equation 3.1 or the Equation 3.2 (whether one wants to use the Plain Similarity metric or the Dispersion-based Similarity metric, respectively) to each possible pair of partial logs.

For each combination of two partial logs, a profile (containing the result of the metric and a list of the SPEs with identical access vectors) is generated and stored in the `MatchLogs` data structure.

2. ***Identify the list of base partial logs*** – the next step consists of identifying the list of the partial logs that can be a potential good basis to start reconstructing the faulty execution. To build this list, CoopLEAP first calculates the relevance of each partial log (storing the result in its respective profile in the `PartialLogProfiles` data structure) and picks the n most relevant ones (we found $n = 10$ to be a suitable value for our experiments).

   It should be noted that this is a sorted list, where the partial logs are arranged in a descending order according to their relevance value. Thence, the first base partial log will be the one with the highest relevance value.

3. ***Complete the base partial log with information from the group of similars*** – having already chosen the base partial log, CoopLEAP can now start the merge of access vectors. For that, CoopLEAP identifies the unrecorded SPEs in the base partial log and completes them with the respective access vectors traced by the logs in the group of similars.

   If all SPEs become filled, the obtained complete log is sent to the replayer, along with the thread ID map and the generated replay driver. On the other hand, if there are still empty SPEs, the heuristic proceeds with the next step.

4. ***Complete the base partial log with information from partial logs "similar by transitivity"*** – when the access vectors from the group of similars are not sufficient to create a complete replay log, CoopLEAP tries to fill the missing SPEs with access vectors from the partial logs "similar by transitivity". These partial logs, although not belonging to the group of similars referred in the previous step, are part of the group of similars of those partial logs which are themselves similar to the base partial log. In other words, if $l_1 \in Sim_{l_0} \wedge l_2 \in Sim_{l_1} \Rightarrow l_2 \in Sim^2_{l_0}$, where $Sim^n_{l_0}$ contains the partial logs which are n^{th}-degree similar to l_0 (in this example, l_2 would be second-degree similar to l_0).

5. ***Complete the base partial log with statistical indicators*** – if it is still not possible to complete the log for replay (the union of the different groups of similars may not cover all the SPEs of the program), CoopLEAP searches the `StatisticalIndicators` data structure for access vectors of the missing SPEs. The `StatisticalIndicators` data structure contains the results of applying the metrics described in Section 3.4.2 to the universe of access vectors collected, thus allowing to know the ones with greater Importance (see Equation 3.7).

At the end of this process, CoopLEAP replays the merged log and verifies if the bug is reproduced. If it is, the goal has been achieved and the process ends. If it is not, CoopLEAP chooses the next partial log in the list of the most relevant to be the new base partial log, and re-executes the Similarity-Guided Merge from the step 3. It should be referred that, in the worst case scenario, where all the most important indicators failed to replay the bug, the heuristic switches to a *brute force* mode. Here, all the possible access vectors are tested for each missing SPE.

3.5.2 Case Study

In order to better understand how the Similarity-Guided Merge operates, this section illustrates its operation using a simple case study. Let us assume a buggy program with four SPEs: w, x, y, and z. Figure 3.9 depicts eight different faulty executions of the program and their respective partial logs (considering a recording of 50% of the SPEs). Notice that the symbol $[w]_1$ corresponds to the hash of the information contained in the access vector recorded for the SPE w, hence $[w]_1 \neq [w]_2$ and so on.

Executing the Similarity-Guided Merge heuristic, the first step is to measure the similarity between all partial logs. In this case, we will use the Dispersion-based Similarity metric, as it brings some additional complexity due to the weight of the SPEs. Thereby, one should start by

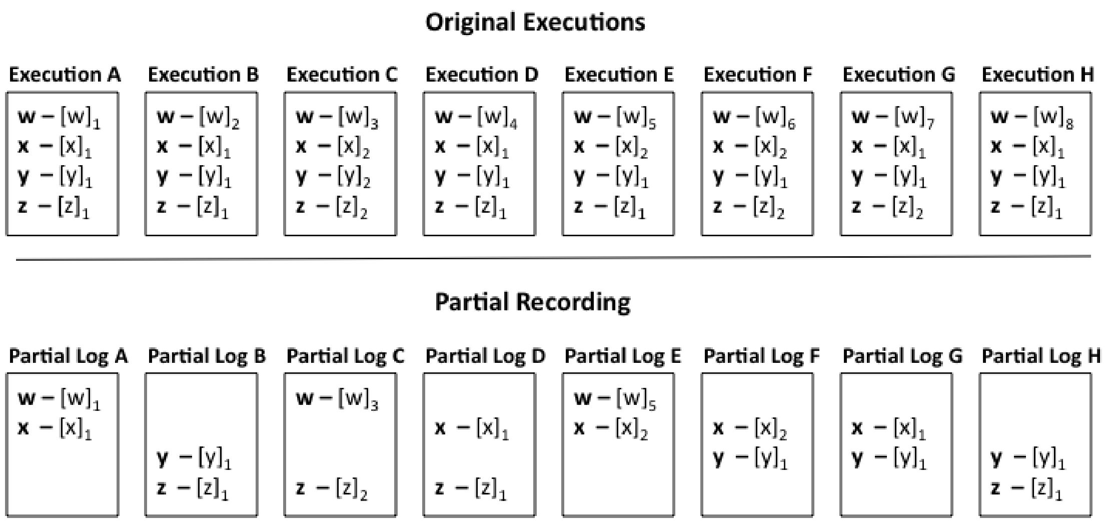


Figure 3.9: Examples of eight faulty executions and their respective partial logs (considering that each partial log records 50% of the SPEs).

calculating the weights:

$$
\begin{aligned}
\mathcal{S} &= \{w, x, y, z\} \\
\#\mathcal{S} &= 4 \\
\mathcal{AV} &= \{[w]_1, [w]_3, [w]_5, [x]_1, [x]_2, [y]_1, [z]_1, [z]_2\} \\
\#\mathcal{AV} &= 8 \\
weight(w) &= \frac{3}{8} = 0.375 \\
weight(x) &= \frac{2}{8} = 0.25 \\
weight(y) &= \frac{1}{8} = 0.125 \\
weight(z) &= \frac{2}{8} = 0.25
\end{aligned}
$$

Taking the partial log A as example, Figure 3.10 illustrates the comparison of access vectors between A and the other partial logs, and shows their values of similarity.

As one can see, among the universe of partial logs which have traced common SPEs with respect to A, only the partial logs D and C have recorded identical access vectors, namely for the SPE x. Given that for both partial logs x is the only SPE in common with A, their similarity takes the value of $weight(x)$ which is equal to 0.25. Therefore, one can say that $Sim_A = \{D, G\}$.

Calculating the similarity values for all the partial logs, one obtains the values shown in

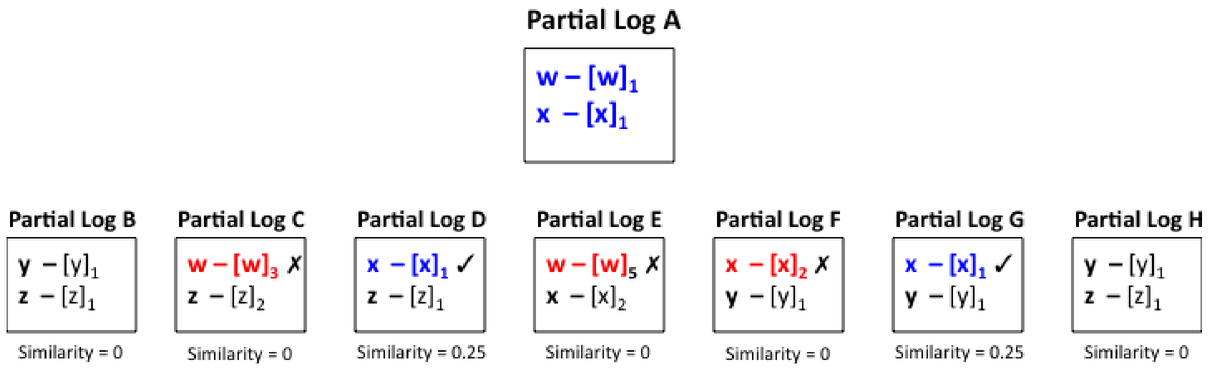


Figure 3.10: Similarity values between the partial log A and the other partial logs.

Table 3.2.

	B	**C**	**D**	**E**	**F**	**G**	**H**
A	0	0	0.25	0	0	0.25	0
B	-	0	0.25	0	0.125	0.125	0.375
C	-	-	0	0	0	0	0
D	-	-	-	0	0	0.25	0.25
E	-	-	-	-	0.25	0	0
F	-	-	-	-	-	0.094	0.125
G	-	-	-	-	-	-	0.125

Table 3.2: Similarity values between the partial logs.

Having the groups of similar logs, the next step is to identify the base partial log, which will be the one with greater value of relevance. Table 3.3 shows the values of relevance for the eight partial logs, as well as the intermediate values needed to calculate them. One can verify that the two partial logs with the highest relevance are then D and A. However, the first base partial log is considered to be D, as it has a greater k and, therefore, has more chances of being completed with compatible access vectors. Regarding the complete sorted list of potential base partial logs, it will be the following: $[D, A, G, F, E, B, H, C]$.

Having already a base partial log, one can move along to the third step, which consists of completing the base partial log with information from the group of similars. Figure 3.11 illustrates that process. In this case, the information from the group of similars is sufficient for fulfilling the base partial log and, therefore, one can already proceed to the replay phase.

Notice that this replay log is different from the original execution D (see Figure 3.9). However, curiously, it is identical to the original execution A, hence allows to deterministically

Log	Sim_{Log}	$Fill_{Log,Sim_{Log}}$	$\sum Sim_{Log}$	k	**Relevance**
A	{D,G}	4	0.5	2	**0.775**
B	{H,D,F,G}	3	0.875	4	**0.591**
C	{}	2	0	0	**0.35**
D	{A,B,G,H}	4	1	4	**0.775**
E	{F}	3	0.25	1	**0.6**
F	{E,B,H,G}	4	0.594	4	**0.745**
G	{A,D,B,H}	4	0.75	4	**0.756**
H	{B,D,F,G}	3	0.875	4	**0.591**

Table 3.3: Relevance values of the partial logs.

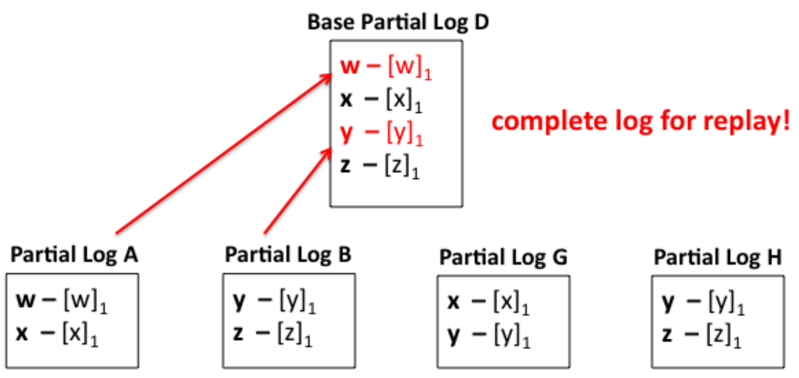


Figure 3.11: Process of completing the base partial log.

reproduce the bug.

In a normal execution of the Similarity-Guided Merge heuristic, the process would stop at this moment. However, for the sake of completion, let us use this case study to also better explain the final steps of the heuristic.

The fourth step is to complete the base partial log with information from partial logs "similar by transitivity". Taking the partial log E as example, one knows that $Sim_E = \{F\}$, which does not provide sufficient information to generate a complete replay log. As a consequence, it is necessary to complete the missing SPEs with access vectors from the partial logs belonging to Sim_F. The chosen partial log will then be the B, as depicted in Figure 3.12. Once more, one now gets a complete log for replay.

In turn, the fifth and final step of the heuristic consists of completing the base partial log with statistical indicators. Here, let us assume the existence of a new partial log $S = \{[w]_1, [x]_1\}$, which has resulted from a successful execution. Let us also take the partial log C as the base

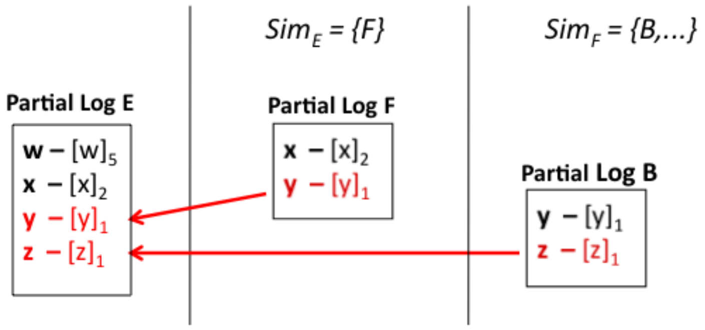


Figure 3.12: Completing the partial log E with information from partial logs "similar by transitivity".

partial log. Applying the statistical metrics described in Section 3.4.2 for each access vector recorded, one obtains the following table:

v	$F(v)$	$S(v)$	Sensitivity	Specificity	**Importance**
$[w]_1$	1	1	0.125	0.5	**0.2**
$[w]_3$	1	0	0.125	1	**0.22**
$[w]_5$	1	0	0.125	1	**0.22**
$[x]_1$	3	1	0.375	0.75	**0.5**
$[x]_2$	2	0	0.25	1	**0.4**
$[y]_1$	4	0	0.5	1	**0.667**
$[z]_1$	3	0	0.375	1	**0.545**
$[z]_2$	1	0	0.125	1	**0.22**

Table 3.4: Statistical indicators for all the recorded access vectors.

where $F_{total} = 8$. Since C has already recorded the SPEs w and z, one is only interested in filling the missing ones, i.e. x and y. Looking up in Table 3.4, one can see that the chosen access vectors will be $[x]_1$ and $[y]_1$, as they are the ones with greater Importance. The complete replay log will then be $\{[w]_3, [x]_1, [y]_1, [z]_2\}$.

Given that it is not granted that replaying this complete log reproduces the bug with success, it may be necessary to generate a new replay log. Assuming that, in this case, all the other base partial logs had already failed the bug replay, one would proceed with the brute force approach, producing a new test case with the other possible access vector, i.e. $[x]_2$.

3.6 Summary

This chapter presented the main components of CoopLEAP architecture, with the focus being on the modifications made to the LEAP system and the process of merging partial logs. The chapter started with an introduction of the standard LEAP system, presenting the local-order based approach for deterministic replay and the techniques used by LEAP to record and replay the SPEs of the program. It proceeded with the depiction of the Statistical Analyzer component and the introduction of the partial log recording scheme. The statistical metrics for partial log correlation, namely the two types of Similarity, were then described, as well as the statistical metrics for bug correlation. This was followed by the introduction of the Similarity-Guided Merge heuristic for combining the partial logs, which was then explained in detail using a case study. In the next chapter, the heuristic for deterministic replay presented is evaluated (using both types of Similarity) and compared to the standard LEAP scheme, using benchmarks and a real world application.

4 Evaluation

This chapter reports the results of the experimental study aimed at evaluating and comparing the performance of CoopLEAP to the one of standard LEAP. It begins by describing the experimental settings and the criteria used in the evaluation. Then, Section 4.3 assesses the bug replay capacity of the Similarity-Guided Merge using both types of Similarity metrics (Plain Similarity and Dispersion-based Similarity) for our micro-benchmark, the IBM ConTest benchmark, and the real world application Tomcat. Section 4.4 compares the results regarding the performance and space overheads imposed by CoopLEAP and LEAP. Finally, Section 4.5 presents a brief discussion of the results obtained.

4.1 Experimental Setting

All the experiments were conducted in a machine Intel Core 2 Duo at 2.26 Ghz, with 4 GB of RAM and running Mac OS X. CoopLEAP prototype was implemented over a LEAP public version[1]. In order to get comparative figures, this standard version of LEAP was also used in the experiments.

Regarding partial logging, three different configurations were employed:

- CLEAP-25% – each partial log traces 25% of the SPEs of the program.
- CLEAP-50% – each partial log traces 50% of the SPEs of the program.
- CLEAP-75% – each partial log traces 75% of the SPEs of the program.

For each one, 500 partial logs from failed executions were used, plus more 50 of successful runs. To get a fairer comparison of the three recording schemes, the partial logs were generated from 500 complete logs, picking randomly the SPEs to be stored according to the scheme's

[1]Available at http://sites.google.com/site/leaphkust/

percentage. For the Plain Similarity we used a threshold of 0.3 and for the Dispersion-based Similarity we used a threshold of 0.01 (given that the weights of some SPEs may be very low). Regarding the maximum number of attempts of the heuristic to reproduce the bug, it was set to 500.

4.2 Evaluation Criteria

Three main criteria were used to evaluate CoopLEAP, namely: *i)* the bug replay capacity, *ii)* the performance overhead, and *iii)* the size of the partial logs produced. The first consists of the number of attempts of the heuristic to replay the bug, therefore, the less number of tries, the better. This metric was applied to compare the heuristic execution with both types of Similarity, assessing which type performs better in which case. The second criterion is measured in terms of the percentage of runtime degradation of the original execution of the program imposed by the instrumentation. The last criterion is related to the size of the partial logs generated to trace the access vectors of the SPEs. It should be noted that the two latter criteria were applied to both CoopLEAP and standard LEAP, in order to evaluate the benefits and the limitations of our solution.

4.3 Bug Replay Capacity

4.3.1 Micro-Benchmark

4.3.1.1 Description

In order to evaluate the correction of the deterministic replay capacity of CoopLEAP, we started by developing a micro-benchmark that allows to easily tune the number of threads and SPEs used in the experiments[2]. This micro-benchmark consists of an application that simulates transfers between bank accounts. Since the threads concurrently update the accounts with no synchronization, the final balance may not be correct. In this application, each thread runs over five iterations of a cycle, alternating between account transfer operations and check operations.

[2]Code available at http://www.gsd.inesc-id.pt/∼nmachado/bank_micro.java

Every time a thread finds an inconsistent balance, it raises an exception that prints a message indicating that a bug has occurred.

Besides the bug being non-deterministic, even when occurs, it may be the result of different thread interleavings (because the bug exceptions may not always be raised by the same threads at the same point in every execution), which further hampers the partial log merge.

To assess how the complexity of the program impacts the bug replay capacity, the micro-benchmark was executed with eight different configurations. These configurations were obtained by varying the number of SPEs of the program (in practice, this corresponds to the number of accounts in the bank) and also by varying the number of threads of these configurations. Thereby, we configured the micro-benchmark for 32 SPEs and 64 SPEs. For each of these configurations, we ran the program with 8, 16, 32, and 64 threads.

4.3.1.2 Results

The number of the attempts required by the heuristic to replay bug for the configurations mentioned above are shown in Table 4.1.

SPEs	**Similarity**	**8 Threads**			**16 Threads**			**32 Threads**			**64 Threads**		
		25%	**50%**	**75%**	**25%**	**50%**	**75%**	**25%**	**50%**	**75%**	**25%**	**50%**	**75%**
32	Plain	2	1	1	X	X	2	X	X	1	X	X	X
	Dispersion	53	1	1	X	X	5	X	X	1	X	X	X
64	Plain	5	1	1	X	1	1	X	X	X	X	X	X
	Dispersion	X	1	1	X	1	1	X	X	X	X	X	X

Table 4.1: Number of the attempts required by the heuristic to replay bug in the micro-benchmark (the X indicates that the heuristic failed to replay the bug in the maximum number of attempts stipulated).

Effect of the Increase in the Number of Threads: From the analysis of Table 4.1, one can see that the heuristic fails to reproduce the bug when the program executes with 64 threads. Moreover, for 32 threads, it can only be replayed by recording 75% of the SPEs. This matches our observation that the more complex the program, the more difficult is to replay the bug. This is mainly due to the fact that, with more threads running, there are more SPE accesses and more possible interleavings, what hinders the existence of similar executions. Table 4.2 and Figure 4.1 depict this phenomenon.

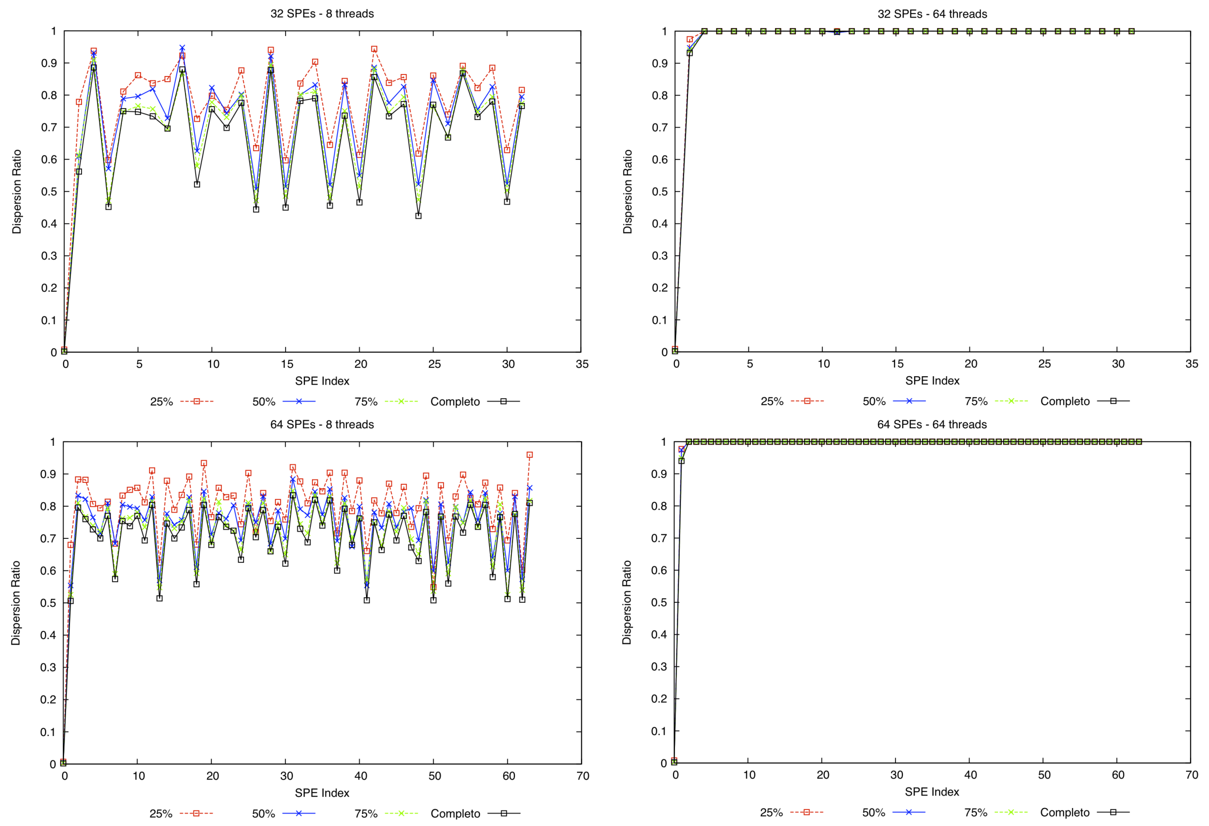


Figure 4.1: SPE dispersion ratios for the four logging configurations (Cleap-25%, Cleap-50%, Cleap-75%, and complete) in both cases of 32 and 64 SPEs, running with 8 and 64 threads.

Table 4.2 indicates, for different configurations scenarios, the sum of all thread accesses performed to the SPEs during the execution of the micro-benchmark application. As expected, the overall number of accesses increases when there are more threads running.

The plots of Figure 4.1 show the *dispersion ratio* of each SPE index of the program, for both 32 and 64 SPEs, running with 8 and 64 threads. Here, the dispersion ratio indicates *how disperse is the SPE*, i.e. whether many different access vectors were recorded for it or not. The dispersion ratio is computed by dividing the number of *different* access vectors recorded for the SPE by the total number of access vectors recorded for that SPE[3]. Thereby, it should be noted that, for the complete recording scheme, the total number of accesses vectors recorded for every SPE is always 500. However, this value can be lower when decreasing the percentage of the recording scheme, as not all SPEs are traced by all the partial logs.

[3]Recalling the case study of Figure 3.9, the *dispersion ratio* of x would be $\frac{2}{5}$, as there are two different access vectors ($[x]_1$ and $[x]_2$) in all the five access vectors recorded for x (referring to partial logs A,D,E,F,G)

SPEs	**Threads**			
	8	**16**	**32**	**64**
32	3570	7138	14281	28576
64	4080	8159	16326	32663

Table 4.2: Overall number of thread accesses for the different configuration scenarios.

Observing Figure 4.1, it can be observed that the dispersion ratio for both cases with 64 threads running is almost always equal to 1, which means that all the access vectors collected are different. In contrast, with 8 threads, the dispersion ratios vary within the interval [0.45;0.94] and [0.51;0.96], for 32 SPEs and 64 SPEs, respectively. A special note for the SPE 0, which is identical for all the executions and, therefore, has a very low dispersion ratio.

The plots also indicate that the dispersion ratio usually increases with the decrease in the percentage of SPE recording. This is explained by the fact that most of the access vectors traced are different, thus being more likely that CoopLEAP picks non-identical access vectors when building the partial logs. As a result, the Similarity-Guided Merge heuristic will probably have less opportunities to combine compatible information and replay the bug when records a smaller percentage of the SPEs.

Returning to Table 4.1, there is a curious case when the program has 32 SPEs: the execution of the heuristic for 32 threads slightly outperforms the one for 16 threads, using CLEAP-75%. The reason is related to the fact that, for the 32 threads case, the heuristic could immediately generate a complete replay log by combining information from only one partial log of the group of similar logs. On the other hand, for the 16 threads case, the replay log was always generated by merging access vectors from other three logs of the similar group, and that information happen not to be compatible at the first attempts.

Effect of SPE Number Increase: Table 4.1 shows that, in general, the increase of SPEs did not bring significant impairments for the Similarity-guided Merge heuristic. In fact, observing Table 4.2, one can note that the increase in the number of accesses is not proportional to the increase in the number of SPEs. This is due to the fact that our micro-benchmark simulates a fixed number of bank transfers, doing computations that use the operation `%bank.numAccounts` to determine both source and destination account indexes. Given that the number of SPEs is related to the number of accounts, having more SPEs results in bank transfers within a sparser universe of accounts. Hence, with 64 SPEs the number of transfers for each account is smaller than with 32 SPEs, so as the number of accesses in the correspondent access vector.

Curiously, for the case with 16 threads, the heuristic performed better with 64 SPEs than with 32 SPEs, specially when using CLEAP-50%. This because, in the latter case, there were SPEs with up to 734 accesses, while in the case with 64 SPEs the maximum number of accesses recorded for a single SPE was only 224. Thereby, the overall dispersion-ratio resulted to be higher for 32 SPEs.

On the other hand, the number of SPEs was very influent when running the program with 32 threads. For 64 SPEs, all the few group of similars created were composed by only one partial log, which was not sufficient to generate a complete replay log. Consequently, the missing access vectors ended being filled with incompatible information.

Plain Similarity vs Dispersion-based Similarity: Comparing the two types of similarity metrics, one can see that the Plain Similarity achieved better results, especially for CLEAP-25%. This follows our idea that Plain Similarity is more suitable for situations where there are not significant disparities in the weight distribution across the majority of the SPEs, just as in this case.

For the configurations with 8 threads and using CLEAP-25%, the problem with the Dispersion-based Similarity was that the similarity threshold was too low. This allowed to create group of similars composed by partial logs with a similiarity degree close to the threshold, but capable of filling many SPEs when combined. Hence, the partial logs in these conditions were classified as the most relevant (see Equation 3.4), in contrary to what happened when running the heuristic with Plain Similarity. In the particular case of this configuration for 64 SPEs, we tested again with a higher threshold (= 0.07), and the heuristic was able to replay the bug at the 81st attempt. However, once more the groups of similars were slightly different to those of Plain Similarity, since the SPE weights were not completely identical (for instance, having SPE 0 in common was irrelevant for the computation of the Dispersion-based Similarity between partial logs).

Finally, it should be noted that the addition of successful logs ended to be irrelevant. As the error occurred with many different execution interleavings, there were always a lot of access vectors with the same statistical importance to fill the missing SPEs.

4.3.2 ConTest Benchmark

4.3.2.1 Description

To further evaluate the bug capacity replay, some programs from a third-party benchmark were also used. The IBM ConTest benchmark suite (Farchi, Nir, & Ur 2003) contains programs with many types of concurrency bugs. The ones used in our experiments are described in Table 4.3, in terms of its number of SPEs, the total number of SPE accesses, and the bug-pattern according to Farchi, Nir, & Ur (2003).

Program	**SPEs**	**Total Accesses**	**Bug Description**
BubbleSort	10	49964	Not-atomic
Manager	4	30240	Not-atomic
TwoStage	4	27103	Two-stage
ProducerConsumer	8	997	Orphaned thread
Piper	6	347	Missing condition for Wait

Table 4.3: Description of the ConTest benchmark bugs used in the experiments.

4.3.2.2 Results

Table 4.4 shows the number of attempts of the heuristic (using both Plain Similarity and Dispersion-based Similarity) to replay the ConTest benchmark bugs, when recording 25%, 50%, and 75% of the SPEs.

Program	**Plain Similiarity**			**Dispersion-based Similarity**		
	25%	**50%**	**75%**	**25%**	**50%**	**75%**
BubbleSort	1	1	1	1	1	1
Manager	X	X	X	X	X	X
TwoStage	34	13	X	7	1	1
ProducerConsumer	5	2	1	1	1	1
Piper	2	1	1	1	1	1

Table 4.4: Number of the attempts required by the heuristic to replay bug in the ConTest benchmark (the X indicates that the heuristic failed to replay the bug in the maximum number of attempts stipulated).

Analyzing the results, one can verify that the Similarity-Guided Merge heuristic only failed to replay the bug in program `Manager`. Besides that, the `TwoStage` bug was also not reproduced in the particular case of Plain-Similarity for CLEAP-75%. These results are also coherent with the statement that executions containing a higher number of total SPE accesses are more unlikely to be successfully reproduced using partial logging strategies. However, an overall high number of accesses per se is not an indicator that the heuristic will fail, as shown by the results obtained for program `BubbleSort`. The main reason for the failure of the heuristic is related to how the accesses are distributed between the SPEs and how that influences the SPE dispersion ratio. Figure 4.2 supports this claim by depicting the SPE dispersion ratios for the ConTest benchmark programs (for the sake of readability and to ease the comparison, Figure 4.2 only presents the values for the complete logging configuration).

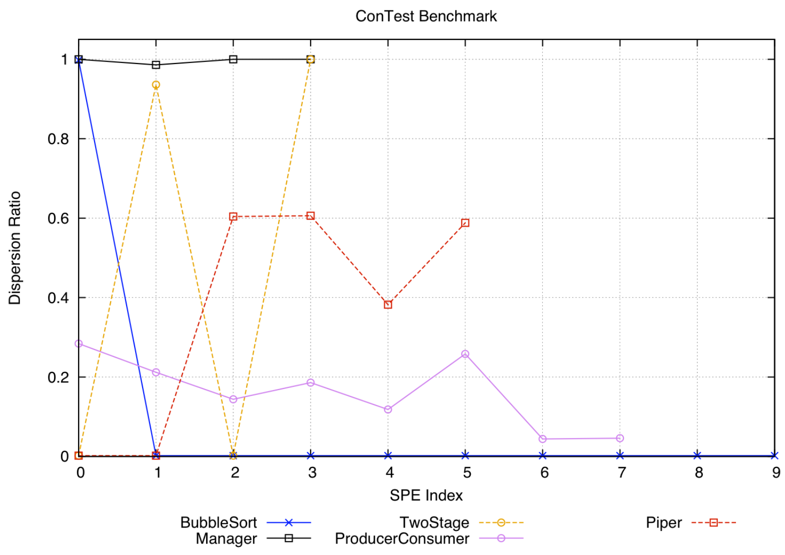


Figure 4.2: SPE dispersion ratios for the ConTest benchmark programs, when logging all the SPEs of the program.

As one can note, the `BubbleSort` program has only one SPE with a very high dispersion ratio (which also accounts for about 99% of the total accesses), while the remaining SPEs always present the same access vector across all the executions. For this reason, the partial log combination ended to be trivial, since it was easy for the Similarity-Guided Merge heuristic to combine compatible information.

On the other hand, the `Manager` program has all its SPEs with a dispersion ratio of 1 or closer, which means that almost all the recorded executions had a different thread interleaving. These clearly represent unfavorable conditions for the partial logging approach, which in fact failed to replay the bug, as indicated in Table 4.4.

Regarding the `TwoStage` application, it presents unusual results when using Plain Similarity, since the bug was not replayed when the partial logs recorded more information. The explanation for this is related to the SPE dispersion ratios. As one can observe in Figure 4.2, from the four SPEs of the program, two were always identical (SPE 0 and 2), one had very few equal access vectors (SPE 1), and the last one was always different (SPE 3). Let us further discuss the three partial logging scenarios when using Plain Similarity:

- **Cleap-75%** – with this configuration, each partial log was composed by three SPEs. Hence, the list of base partial logs ended being composed by the partial logs whose group of similars contained only other partial logs matching in the SPEs 0 and 2. As a consequence, the access vectors combined for filling either SPE 1 or 3 were incompatible.
- **Cleap-50%** – with this configuration, each partial log was composed by two SPEs. Here, the list of base partial logs was filled with the partial logs that have other ones matching access vectors for the SPE 1. This because all the partial logs containing only SPEs 0 and 2, albeit having many other similar partial logs, could not generate a complete replay log just by combining information from their group of similars. Therefore, their relevance was lower (see Equation 3.4). The same did not happened for the partial logs containing SPE 1 and the bug was replayed by trying different access vectors for filling SPE 4.
- **Cleap-25%** – with this configuration, each partial log was composed by a single SPE. Since there were no intersection points between the partial logs, the Similarity-guided Merge heuristic picked random partial logs to act as base to generate the replay log. Then, it tried to replay the error by successively filling the missing SPEs with the access vectors indicated by the statistical indicators. As can be verified, the bug was successful replayed at the 34th attempt.

On the other hand, when using Dispersion-based Similarity, the heuristic could easily reproduce the bug, because the SPEs had different importances. Thence, the partial logs with the same access vector for SPE 1 were immediately identified as the best base partial logs and used to generate a complete replay log.

As final remark, once more it could be observed that the addition of successful logs did not impact the results. The reason is the same referred for the micro-benchmark, i.e. when it was necessary to fill missing SPEs, there were always many different access vectors with the same degree of correlation to the bug.

4.3.3 Tomcat

4.3.3.1 Description

The real-world application used in the experiments was Tomcat[4], which is an open source software implementation (developed by the Apache Software Foundation) of the Java Servlet and JavaServer Pages specifications from Oracle Corporation. The bug replay capacity of the Similarity-Guided Merge heuristic was tested with bug `#37458` [5] of Tomcat v5.5. This error consists of a `NullPointerException`, resulting from a data race, and was already used in the work of Huang, Liu, & Zhang (2010) to test the LEAP solution.

In terms of SPEs and number of shared accesses, this application bug requires the recording of 15 SPEs, which account for a total of 61 accesses. This is due to the fact that we used a test unit (JUnit) to trigger the bug, therefore the transformer only instruments the SPEs accessed during the execution of the JUnit class. The use of a driver can be considered as an useful asset, since it allows to circumscribe the really needed SPEs to replay the bug, which is better in terms of scalability (one avoids to instrument all the unnecessary SPEs of the program).

4.3.3.2 Results

Table 4.5 shows the number of attempts of the Similarity-Guided Merge heuristic (using both Plain Similarity and Dispersion-based Similarity) to replay the Tomcat#37458 bug, when recording 25%, 50%, and 75% of the SPEs.

Program	**Plain Similiarity**			**Dispersion-based Similarity**		
	25%	**50%**	**75%**	**25%**	**50%**	**75%**
Tomcat#37458	2	1	1	1	1	1

Table 4.5: Number of the attempts required by the heuristic to replay Tomcat#37458 bug.

[4]http://tomcat.apache.org/
[5]https://issues.apache.org/bugzilla/show_bug.cgi?id=37458

From the analysis of Table 4.5, it can be verified that our heuristic could easily replay the bug. In fact, one can say that this is a relatively simple error in terms of complexity, as can be proved by the SPE dispersion ratios illustrated in Figure 4.3. This means that practically all the 500 execution logs collected resulted from production runs originating very similar thread interleavings.

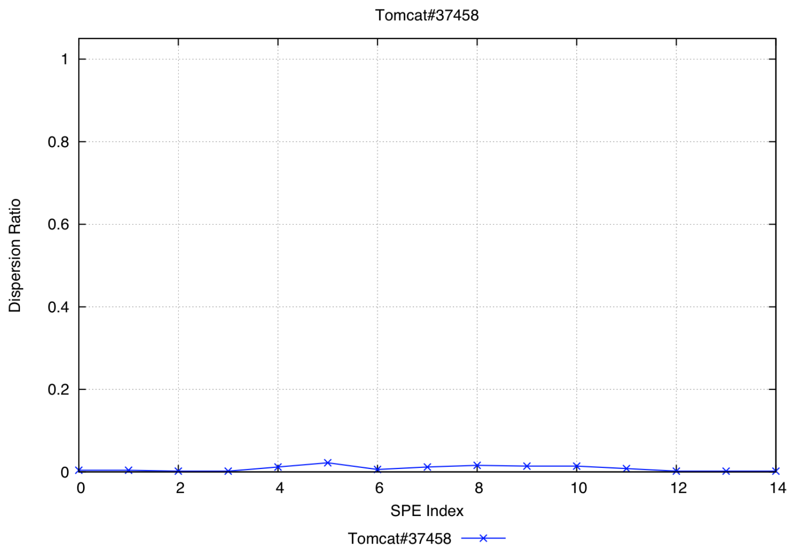


Figure 4.3: SPE dispersion ratios for the Tomcat#37458 bug, when logging all the SPEs of the program.

4.4 Overheads

This section analyzes the overheads related to the performance degradation and to the size of the logs generated, during the recording phase. Both types of overheads are measured for the programs of both ConTest and Java Grande Forum benchmarks, as well as for the Tomcat bug. It should be noted that, to measure the performance overhead, the program was instrumented three times for each partial recording scheme (in order to test with different subsets of SPEs recorded) and, for each of these times, the average value of three samples of the execution time was collected. In turn, the log size was measured by computing the average size of ten partial log samples for each recording scheme.

4.4.1 ConTest Benchmark

4.4.1.1 Performance Overhead

Figure 4.4 depicts the performance overheads of the tested programs. As one can note, by using partial recording, CoopLEAP achieved always lower runtime degradation than standard LEAP (which corresponds to the recording configuration of 100% in Figure 4.4).

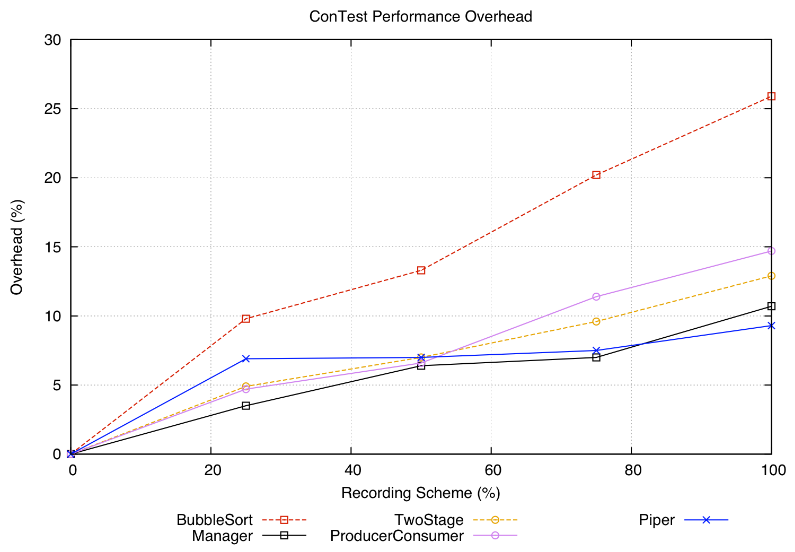


Figure 4.4: Performance overheads for the ConTest benchmark programs. Standard LEAP corresponds to the recording configuration of 100%.

As one can see, the overhead with respect to the original full recording configuration is decreasing as expected by a factor almost equal to 75%, 50%, and 25%, according to the homonym partial recording configuration, respectively. The decreases are just not completely linear because some SPEs are accessed more times than others. Given that the instrumentation of the code is performed statically, the load balance in terms of thread accesses may not be equally distributed among the users, as previously referred in Section 3.3. This implies that the impact of logging $x\%$ of the SPEs will not necessarily mean a reduction of $x\%$ in both performance overhead and log size. In fact, sometimes the reduction may be greater than expected, but other times may be lower.

Another observation is that the advantage of using partial logging is obviously higher in those scenarios where the usage of full logging has a higher negative impact on performance.

The most notorious case is `BubbleSort`, which is the application with more overall accesses (see Table 4.3). In `BubbleSort`, LEAP imposed a performance overhead of 26%, while CoopLEAP only slowed down the execution time by 20.2%, 13.3%, and 9.8%, when recording 75%, 50%, and 25% of the SPEs, respectively. In turn, `Piper` is the program with less number of accesses and where the gains were lower. For `Piper`, CoopLEAP achieved a runtime overhead of 6.9% for Cleap-25%, whilst LEAP made the program 9.3% slower.

Similarly, for the remaining programs, the best improvements were achieved when recording only 25% of the SPEs. With this configuration, comparing to LEAP, CoopLEAP obtained execution overheads 3.13x, 3.06x, and 2.63x better, for the programs `ProducerConsumer`, `Manager`, and `TwoStage`, respectively. Finally, from an overall analysis of the table, it can be verified that, when using Cleap-25%, the runtime penalties are always less than 10%, which is an acceptable value for real world applications, according to previous works in the topic (Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009; Jin, Thakur, Liblit, & Lu 2010).

4.4.1.2 Log Sizes

Regarding the size of the logs produced in the recording phase, Figure 4.5 depicts the results obtained for the ConTest benchmark applications tested, in terms of the ratio between the size of the logs generated by the partial recording configurations and the size of the logs generated by LEAP.

It can be noted that the log size ratios also follow the linear decreasing trend observed in the performance overhead plots. Here, `Manager` was the program in which CoopLEAP shown a reduction ratio more similar to the one expected by decreasing proportionally the recording percentage (the log size ratios were 0.22, 0.52, and 0.83 when recording 25%, 50%, and 75% of the SPEs, respectively). The same applies to `ProducerConsumer` for the latter two configurations, with log size ratios of 0.51 and 0.74, respectively for Cleap-50% and Cleap-75%.

In turn, `BubbleSort` was the program where the log sizes decreased faster for Cleap-50% and Cleap-75% (the log size ratios for these configurations, with reference to LEAP, were 0.36 and 0.62, respectively). However, the reduction obtained when moving from Cleap-50% to Cleap-25% was not too significant, as Cleap-25% had a log size ratio of 0.35. This is due to the fact that the largest fraction of accesses is confined to a single SPE, which was recorded the same number of times for the ten logs measured for these configurations.

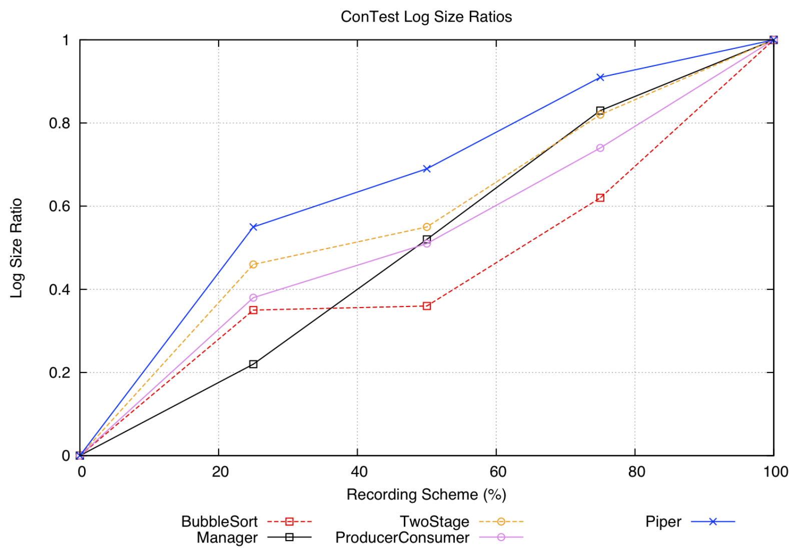


Figure 4.5: Log size ratios for the ConTest benchmark programs with respect to the log size generated by LEAP. Standard LEAP corresponds to the recording configuration of 100%.

On the other hand, `Piper` was the application where decreasing the percentage of logged SPEs led to smaller reductions of the log sizes. This is again explainable by the heterogeneity in the size of the access vectors associated with the various SPEs. In this application, recording 25% of the SPEs only led to a log size ratio of 0.55.

4.4.2 Tomcat

4.4.2.1 Performance Overhead

Figure 4.6 illustrates the performance overhead for Tomcat. Observing the figure, one can conclude that once more CoopLEAP reduced the performance overhead proportionally to the percentage of logged SPEs. It only required 0.9%, 1.4%, and 2.8% of additional execution time, respectively when using Cleap-25%, Cleap-50%, and Cleap-75%, whilst LEAP degraded the performance time in 4.7%.

4.4.2.2 Log Sizes

Given that replaying bug Tomcat#37458 implies very few thread accesses to shared variables, the logs obtained with partial recording are only a couple of bytes smaller than the logs

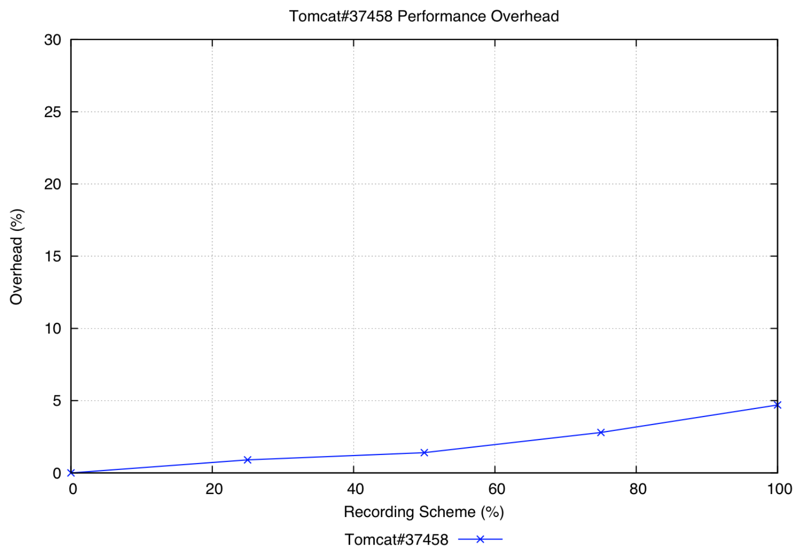


Figure 4.6: Performance overheads for the Tomcat program. Standard LEAP corresponds to the recording scheme of 100%.

generated by LEAP. Hence, one obtained log size ratios of 0.90, 0.91, and 0.95 when recording 25%, 50%, and 75% of the SPEs, respectively. This depends on the fact that the data structures used to store the SPEs have a minimum fixed cost associated with the access vectors' metadata, which is independent of the actual number of entries stored in the access vector. Given that with this benchmark, the number of entries stored in each access vector is extremely limited (5 on average), the size of the log is dominated by the metadata. Therefore, the impact of partial logging ended to be less significant.

4.4.3 Java Grande Forum Benchmark

4.4.3.1 Description

The Java Grande Forum[6] was a community initiative to promote the use of Java for so-called "Grande" applications, which are typically computationally intensive science and engineering applications requiring high-performance computers. In this line, it was developed a suite of benchmarks to measure different execution environments of Java against each other and native code implementations.

[6]http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/

Given that Java Grande Forum Benchmark does not have known bugs, it only was used in our experiments to assess the benefits and limitations of CoopLEAP when compared to LEAP, on demanding computing environments. Table 4.6 describes the benchmark programs used in terms of number of SPEs and the overall number of times that they are accessed. For the sake of readability, in the next sections, the results of the tests performed are presented in tables, since the values obtained vary within a large scale.

Program	**SPEs**	**Total Accesses**
Raytracer	16	2.56×10^9
SparseMatmult	8	5.08×10^7
SOR	8	1.99×10^6
Montecarlo	15	1.50×10^5
Series	8	2.00×10^4

Table 4.6: Description of the Java Grande Forum benchmark programs used in the experiments.

4.4.3.2 Performance Overhead

Table 4.7 contains the experiments with respect to the performance overhead measured when tracing the SPEs with the previous logging configurations.

Program	**Performance Overhead**			
	25%	**50%**	**75%**	**LEAP**
Raytracer	9566.7%	17452.1%	44610.6%	92908.4%
SparseMatmult	598.2%	1725.5%	2505.1%	2606.7%
SOR	1.1%	2.0%	2.4%	2.7%
Montecarlo	1.5%	2.3%	3.7%	7.3%
Series	0.1%	0.4%	2.3%	6.5%

Table 4.7: Performance overheads for the Java Grande Forum benchmark programs.

Once more, one can verify that there is always a decrease of the performance overhead of CoopLEAP when compared to standard LEAP. The most preponderant case is `Series`, where CoopLEAP achieved a runtime degradation 65x and 16x smaller than LEAP, for Cleap-25% and Cleap-50%, respectively. These reductions are explained by the fact the majority of the accesses are confined to only two of the eight SPEs of the program. Hence, when those specific SPEs are not traced, the imposed overhead is automatically lower. However, for this program, even the worst case overhead was not very significant.

On the other hand, for `Raytracer`, one can note that both CoopLEAP and LEAP still incur in a heavy performance overhead, as a result of the high number of accesses performed to the SPEs. Nonetheless, once more CoopLEAP brought visible improvements, reducing LEAP penalties by 9.7x, 5.3x, and 2.1x when logging 25%, 50%, and 75% of the SPEs, respectively. This scenario is similar to that of `SparseMatmult`, where CoopLEAP achieved decreases of 4.4x and 1.5x (for CLEAP-25% and CLEAP-50%, respectively) when compared to the runtime degradation of LEAP. This trend also holds for the remaining programs, however with less significant overheads.

4.4.3.3 Log Sizes

Concerning the log size ratios with respect to LEAP, the results are shown in Figure 4.7.

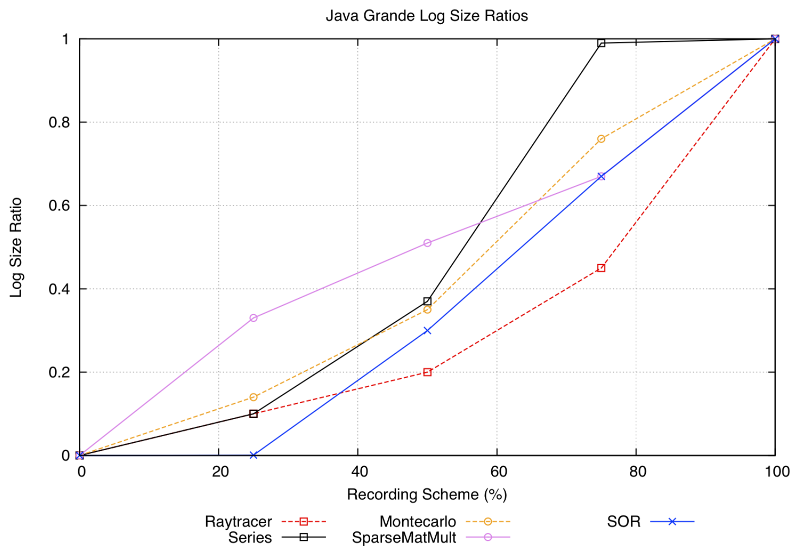


Figure 4.7: Log size ratios for the Java Grande benchmark programs with reference to log size generated by LEAP (which corresponds to the recording scheme of 100%).

From the figure analysis, the benefits of partial logging are clear. The most evident case is `SOR`, where the log sizes when using CLEAP-25% account for only 0.1% of LEAP's log size. For `SOR` with both CLEAP-50% and CLEAP-75%, the ratios were 0.33 and 0.67, respectively, which is even smaller than the expected. In fact, for all the benchmark programs, there was a high heterogeneity in the size of the access vectors of the program SPEs, which significantly influenced the actual reduction in the log sizes. As already discussed in Section 4.4.1.1, this can lead to higher (as in `Raytracer` and `SparseMatMult`) or lower (as in `Series` when using Cleap-75%) reductions of the log size, and motivates future research in how one can equally distribute the information to be recorded among the different clients.

4.5 Discussion

From the evaluation described in the previous sections, one can conclude that, in general, the Similarity-Guided Merge heuristic is capable of replaying concurrency errors in the presence of a significant universe of partial logs. However, the bug replay capacity is clearly hampered when dealing with complex applications, i.e. applications with many SPEs and, more important, with a large amount of shared accesses. In fact, when almost each partial log, out of the 500 different logs examined, presented a different execution interleaving (which led to SPEs with a dispersion ratio close to 1), our heuristic was not able to successfully reproduce the bug. However, this might be addressed by having a larger population of users, where the probability of finding similar logs will be higher. We schedule this experiments as future work.

Regarding the two types of Similarity metrics, Dispersion-based Similarity outperformed Plain Similarity when the dispersion weight was not well distributed across all the program SPEs, i.e. when there were few SPEs that accounted for many different access vectors, while the others exhibited almost always the same execution interleaving, as observed in the `TwoStage` application of the ConTest benchmark. In turn, Plain Similarity achieved better results when the weight distribution does not shown relevant disparities across the SPEs of the program. It should be also remarked the importance of the heuristic threshold, to avoid the creation of groups of similars with little compatible partial logs. Our experiments highlight the need of having a better way of setting the threshold in place of being a static value, since it may vary according to the application. A possible solution is to define the threshold at runtime, by analyzing the SPEs' weight and defining the minimum acceptable log similarity according to it.

In terms of the overhead imposed by both CoopLEAP and standard LEAP, one can conclude that the former always presented smaller degradations, both in terms of log sizes and performance. Furthermore, the smaller the recording percentage, the smaller were the overheads. However, the reduction was not always proportional to percentage decrease, as the SPEs had different number of accesses.

Unfortunately, for computationally intensive applications, both the time and space overheads imposed by CoopLEAP are still very high to be acceptable. This brings the need of finding new solutions to further minimize the recording penalties. A possible line of research could be the partial recording of the access vectors content, in addition to the SPEs.

4.6 Summary

This chapter presented the experimental study based on different benchmarks and a real world application. It began with a in-house developed micro-benchmark, which served two main purposes, namely, assessing the impact of increasing the number of threads and SPEs on the bug replay capacity of the Similarity-Guided Merge heuristic. Next, some applications from the IBM ConTest benchmark and a real bug from Tomcat were used to further test the bug replay capacity of our heuristics. In addition, all these applications served also as a basis to evaluate the benefits of both metrics of Similarity (Plain Similarity and Dispersion-based Similarity).

Then, a comparative analysis between CoopLEAP and LEAP was made, regarding both the performance overheads imposed and the size of the logs generated by the two solutions. For this analysis, we used the ConTest benchmark, the Tomcat bug example, and the Java Grande Forum benchmark. Finally, the results obtained were briefly discussed.

5 Conclusion

5.1 Conclusions

With the advent of multi-processors, it becomes appealing to develop parallel software programs that take full advantage of the available computing resources and achieve better performance. However, writing and debugging concurrent programs is a very challenging task because of their non-deterministic nature, i.e. running the same program several times may lead to different outcomes for each run. The deterministic replay technique addresses this problem, as it provides a faithful reproduction of the original run. Unfortunately, deterministic replay comes with very expensive overheads, since it requires recording all sources of non-determinism to achieve the original program behavior.

To address this problem, this thesis proposes a scheme to reduce the amount of information that a given program instance is required to store to support deterministic replay. To this end, the thesis introduced CoopLEAP, a system based on LEAP (Huang, Liu, & Zhang 2010) that provides fault replication of concurrent programs, based in cooperative recording and partial log combination. To avoid a brute force approach to find a compatible combination of partial logs, capable of successfully replaying the bug, we also developed a heuristic, denoted Similarity-Guided Merge. The Similarity-Guided Merge heuristic employs statistical metrics adapted from previous work on statistical debugging (Liblit, Naik, Zheng, Aiken, & Jordan 2005), but it also introduces novel ones. Specifically, two novel statistical metrics were presented in this thesis, namely Similarity (accounts for the similarity degree between two partial logs) and Relevance (accounts for the likelihood of a partial log being completed by compatible information). In the particular case of Similarity, two versions were also proposed: Plain Similarity and Dispersion-based Similarity, according to whether the shared program elements (SPEs) are considered to have the same importance or not, respectively.

With the objective of assessing the bug replay capacity of the Similarity-Guided Merge heuristic, an experimental study was performed using bugs from benchmarks and from a real world application. In the tests, three recording schemes were used, corresponding to recording 25%, 50%, and 75% of all SPEs. The experimental study was also aimed at evaluating CoopLEAP against standard LEAP in terms of the runtime overheads imposed.

From the evaluation results, the benefits from partial recording are clear, as CoopLEAP could always reduce both performance degradations and log sizes produced by LEAP. Furthermore, it was shown that the Similarity-Guided Merge heuristic can successfully replay concurrency bugs by combining information traced by different partial logs. Unfortunately, in the presence of more complex programs (where almost every faulty execution presents a different thread interleaving), the heuristic exhibited limitations in replaying the bug within the maximum number of attempts set.

Concerning the two types of Similarity metrics, one can conclude that Dispersion-based Similarity is more suitable for programs where only a fraction of the SPEs presents high dispersion (i.e. many different thread access interleavings across the executions), while the remainder fraction stays identical. On the other hand, when dealing with applications with no relevant disparities in the dispersion of the SPEs, Plain Similarity is possibly the best approach.

5.2 Future Work

The bug replay capacity and the performance of the Similarity-Guided Merge heuristic presented in this thesis should be further experimented and evaluated in more complex and realistic scenarios, with also a larger number of partial logs collected. Further evaluation should mainly focus on the impact of applications with longer execution times and several thousands of lines of code, which produce larger and more complex logs. In this context, the implementation of a lightweight checkpointing mechanism would also be an useful asset, since it might not be convenient to replay the whole program execution concerning the long replay time and the large log size. Thereby, using checkpoints, one could only replay the program from the last checkpoint to the recording end point.

Regarding partial logging schemes, in this thesis, we only relied on recording a certain percentage of SPEs of the program (selected randomly). Thereby, an interesting direction for future work is to study and develop new partial logging schemes. For instance, to take into account load balancing, to define a small subset of fixed SPEs to be recorded by all users, and/or to set pre-defined subsets of SPEs for different users (allowing to get a quorum when collecting the partial logs), would all be interesting criteria to star with. In terms of the similarity degree between partial logs, this thesis covered statistical metrics that consider two access vectors to be similar only when their content is strictly the same. However, this can be a too restrict criterion for all cases, hence new statistical metrics should be studied (e.g. euclidean or edit distances between access vectors, to determine *how much distinct they are*).

Finally, the results of the experimental study suggest that using static thresholds when building the groups of similar partial logs is not the best approach. In this line, one should consider on using adaptive thresholds, which would leverage on SPE dispersion or on other metric that allow to better capture the reproducibility complexity of the program.

Bibliography

Altekar, G. & I. Stoica (2009). Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pp. 193–206. ACM.

Bacon, D. F. & S. C. Goldstein (1991). Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD'91, pp. 194–206. ACM.

Bodden, E. & K. Havelund (2008). Racer: effective race detection using aspectj. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA'08. ACM.

Castro, M., M. Costa, & J.-P. Martin (2008). Better bug reporting with better privacy. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'08, pp. 319–328. ACM.

Chilimbi, T. M., B. Liblit, K. Mehra, A. V. Nori, & K. Vaswani (2009). Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE'09, pp. 34–44. IEEE Computer Society.

Choi, J.-D. & H. Srinivasan (1998). Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT'98, pp. 48–59. ACM.

Cornelis, F., A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, & K. D. Bosschere (2003). A taxonomy of execution replay systems. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet.*

Dunlap, G. W., D. G. Lucchetti, M. A. Fetterman, & P. M. Chen (2008). Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'08, pp. 121–130. ACM.

Farchi, E., Y. Nir, & S. Ur (2003). Concurrent bug patterns and how to test them. In

Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS'03, pp. 286–293. IEEE Computer Society.

Feldman, S. I. & C. B. Brown (1988). 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, pp. 112–123. ACM.

Fonseca, P., C. Li, V. Singhal, & R. Rodrigues (2010). A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'10, pp. 221–230. IEEE Computer Society.

Georges, A., M. Christiaens, M. Ronsse, & K. De Bosschere (2004, May). Jarec: a portable record/replay environment for multi-threaded java applications. *Software Practice and Experience 40*, 523–547.

Godefroid, P. & N. Nagappan (2008, May). Concurrency at microsoft – an exploratory survey. *Larus*, 1–4.

Hall, A. (2007). Realising the benefits of formal methods. *Journal of Universal Computer Science 13*(5), 669–678.

Halpert, R. L., C. J. F. Pickett, & C. Verbrugge (2007). Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT'07. IEEE Computer Society.

Huang, J., P. Liu, & C. Zhang (2010). 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, pp. 385–386. ACM.

Jin, G., A. Thakur, B. Liblit, & S. Lu (2010). 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, pp. 241–255. ACM.

Lamport, L. (1978, July). Ti clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 558–565.

LeBlanc, T. J. & J. M. Mellor-Crummey (1987, April). Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36*, 471–482.

Li, Z., L. Tan, X. Wang, S. Lu, Y. Zhou, & C. Zhai (2006). 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, pp. 25–33. ACM.

Liblit, B., A. Aiken, A. X. Zheng, & M. I. Jordan (2003). Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI'03, pp. 141–154. ACM.

Liblit, B., M. Naik, A. X. Zheng, A. Aiken, & M. I. Jordan (2005). Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 05, pp. 15–26. ACM.

Lu, S., S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, & Y. Zhou (2007). 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, pp. 103–116. ACM.

Lu, S., S. Park, E. Seo, & Y. Zhou (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'08, pp. 329–339. ACM.

Machado, N., P. Romano, & L. Rodrigues (2011). Reprodução de faltas em programas concorrentes através da combinação de múltiplos históricos parciais. In *Actas do Terceiro Simpósio de Informática*, INForum'11.

Musuvathi, M., S. Qadeer, T. Ball, G. Basler, P. A. Nainar, & I. Neamtiu (2008). Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 267–280. USENIX Association.

Narayanasamy, S., G. Pokam, & B. Calder (2005). Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA'05, pp. 284–295. IEEE Computer Society.

Netzer, R. H. B. (1993). Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD*, pp. 1–11. ACM.

of Standards, N. I. & D. o. C. Technology (2002, June). Software errors cost u.s. economy $59.5 billion annually. *NIST News Release 2002-10*.

Omar, A. A. & F. A. Mohammed (1991, April). A survey of software functional testing methods. *SIGSOFT Softw. Eng. Notes 16*, 75–82.

Orso, A., D. Liang, M. J. Harrold, & R. Lipton (2002). Gamma system: Continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'02, pp. 65–69. ACM Press.

Pablo Montesinos, L. C. & J. Torrellas (2008). Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA'08, pp. 123–134. IEEE Computer Society.

Park, S., Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, & S. Lu (2009). Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pp. 177–192. ACM.

Parnas, D. L. (2010, January). Really rethinking 'formal methods'. *Computer 43*, 28–34.

Patil, H., C. Pereira, M. Stallcup, G. Lueck, & J. Cownie (2010). Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEEACM International Symposium on Code Generation and Optimization*, CGO'10, pp. 2–11. ACM.

Pokam, G., C. Pereira, K. Danne, L. Yang, & J. Torrellas (2009, January). Hardware and software approaches for deterministic multi-processor replay of concurrent programs. *Intel Technology Journal 13*, 20–41.

Sorin, D. J., M. M. K. Martin, M. D. Hill, & D. A. Wood (2002). 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, pp. 123–134. IEEE Computer Society.

Srinivasan, S. M., S. Kandula, C. R. Andrews, & Y. Zhou (2004). Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC'04, pp. 29–44. USENIX Association.

Steegmans, E., P. Bekaert, F. Devos, G. Delanote, N. Smeets, M. van Dooren, & J. Boydens (2004). Black & White Testing: Bridging Black Box Testing and White Box Testing. In *Software Testing: Beheers Optimaal de Risico's van IT in Uw Business*, pp. 1–12. Sterck, P.

Wang, R., X. Wang, & Z. Li (2008). Panalyst: privacy-aware remote error analysis on commodity software. In *Proceedings of the 17th Conference on Security Symposium*, pp. 291–306. USENIX Association.

Xu, M., R. Bodik, & M. D. Hill (2003). A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual International Symposium on Computer Architecture*, ISCA'03, pp. 122–135. ACM.

Zamfir, C. & G. Candea (2010). Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, pp. 321–334. ACM.