# Record and Replay of Multithreaded Applications

Aliaksandra Sankova
aliaksandra.sankova@ist.utl.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

**Abstract.** With the advent of multi-core processors, there is a strong motivation to write concurrent programs, that can fully exploit the parallelism offered by the hardware. Unfortunately, developing concurrent programs is a difficult task, prone to bugs that are difficult to find and to correct. Common concurrent programming paradigms, such as those relying on shared memory and explicit synchronization, can lead to subtle data races that generate errors just for some particular thread interleavings. Furthermore, these interleavings may be hard to reproduce. Due to these reasons, there is a growing interest in developing tools that help the programmers reproduce concurrency bugs. One of the main techniques to achieve this goal is what has been called *record and replay*: it consists of logging relevant information during the execution of a program that allows the interleaving that causes the bug to be reproduced later. Unfortunately, the logging operation can introduce a substantial slowdown in the execution of the application. In this report, we survey related work and discuss how the current state of the art may be improved.

## 1 Introduction

In this report we address the problem of reproducing bugs in concurrent programs that rely on the shared memory paradigm. In these programs, different sequencial threads execute in parallel, quite often in different cores of a multiprocessor, and communicate with each other by reading and writing in shared variables. Threads use explicit synchronization, such as *locks* or *semaphores*[1], to coordinate among each other. In particular, when accessing shared data, the logic of the program must ensure that the threads use the required synchronization to avoid *data races*[2]. A data race occurs when different threads access a shared data structure without synchronization and, at least, one of those accesses is a write. Data races can be avoided by the correct use of synchronization primitives but, unfortunately, these primitives are hard to master. An incorrectly placed or missing synchronization primitive may not eliminate the race and can even create other bugs, such as *deadlocks*[3]. Furthermore, the interleavings that cause the bug may happen only in some rare circunstancies, and be very hard to reproduce. This makes the debugging of concurrent programs an extremely difficult and tedious task. In this context, we aim to study techniques and tools that simplify the reproduction of concurrency bugs.

One of the main techniques to achieve the reproduction of concurrency bugs is what has been called *record and replay* [4–13] (or *deterministic replay*). Record and replay relies in instrumenting the application in order to record all sources of non-determinism at runtime, including inputs, interrupts, signals, and scheduling decisions. For multi-core environment, it is also necessary to record the order by which different threads have accessed shared variables. This way, deterministic replay can be achieved by re-executing the application while enforcing all points of non-determinism to comply with the information stored in the log.

Unfortunately, faithfully logging a concurrent program's execution requires inserting additional synchronization (to ensure that the thread interleaving is correctly traced). This, in combination with the large amount of information that may be required to be captured, can induce an unacceptable slowdown in the application [5, 6, 9, 10]. To address this issue, some approaches attempted to reduce the amount of synchronization used to register the log, the amount of information included in the log, or even both. For instance, one can trace only partial information during the production run and then, at replay time, use search techniques to infer thread interleavings that are compliant with the (partial) log information [8, 13]. However, since reducing the amount of information logged hinders the replay determinism, the challenge lies in identifying the best trade-off between recording cost and inference time.

In this report, we present a survey of the most relevant literature on record and replay, including systems that attempt to fully capture the order of accesses to shared variables by different threads, and systems that combine partial or inaccurate information with search techniques to reproduce the bug. Based on the analysis of these techniques, we conjecture a hypothesis on how the state of the art can be improved. Finally, we describe a plan to develop and evaluate the proposed ideas.

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

## 2 Goals

Conservative record and replay solutions introduce a significant amount of additional synchronization to ensure that accurate information regarding thread interleavings is stored in the log. Recent work proposed a number of techniques to reduce this amount of synchronization. Our work also explores this path and will research additional techniques that may further reduce the cost of logging, while still allowing for bug reproduction. In detail:

> *Goals*: We aim at experimenting a relaxation of the logging procedure, that will simply avoid synchronization when recording concurrent

accesses to shared variables. This will allow for imprecise information to be stored in the log. To compensate for this fact, we will devise replay mechanisms that can look for different interleaving that are compliant with the information stored, in order to reproduce the bug in affordable time.

In order to achieve the goal above we will start by changing the logging mechanisms of STRIDE[12], to also relax the recording order of shared write operations, in addition to reads. Then, we will devise and implement a module capable of deriving an execution ordering, from the relaxed logs, that replays the error. In summary, we will design and implement an optimistic record and replay solution where the record of thread interleavings will be performed without any synchronization.

The project is expected to produce the following results.

*Expected results:* The work will produce *i)* a specification of the algorithms to trace, collect and analyze information in order to replay concurrent executions; *ii)* an implementation of a prototype of this system, and *iii)* an extensive experimental evaluation using real-world applications and third-party benchmarks with concurrency bugs.

## 3 Related Work

Recording and replaying multithreaded applications has been an object of study from more than a decade. In this section, we review the most important related work and discuss the key concepts that have been introduced in the literature. Section 3.1 presents an overview of the record and replay approach, along with its main challenges. Section 3.2 describes the design choices and trade-offs that one has to take into account when implementing this kind of systems. Section 3.3 presents an overview of the most relevant record and replay solutions. Finally, Section 3.4 summarizes the related work.

### 3.1 Record and Replay

The goal of the record and replay (or *deterministic replay*) technique is to allow a given execution of a program to be faithfully reproduced. This requires the ability to reproduce not only the inputs provided to the program, but also all non-deterministic events, such as interrupts and thread interleavings. Deterministic replay has several applications, namely simplifying debugging, support performance analysis of programs and architectures, exploit security vulnerabilities, among others[14]. In general, techniques that provide deterministic replay operate in two distinct phases:

1. *Record phase*, which consists in logging, at runtime, relevant data about the execution, such as values read from inputs and shared variables.

2. *Replay phase*, which consists in re-executing the program in a controlled manner, consulting the log to ensure that the reproduction follows the same steps as the original run.

Although simple in theory, building a deterministic replay system poses several issues. In the following, we discuss the most relevant challenges that need to be addressed in the context of record and replay:

**Non-determinism:** There are many sources of non-determinism in a system[15]. At the user level, sources of non-determinism include certain system calls, signals, special architecture instructions[14]. At the system level, major sources of non-determinism are signals from I/O devices, hardware interrupts, and writes performed by Direct Memory Access[14]. No system deals with all forms of non-determinism[16] and different systems target different levels of abstraction.

**Runtime overhead:** The record phase may induce a non-negligible overhead during the execution of a program. The more information is included in the log, i.e., the finer the granularity of logging, the easier it is to ensure deterministic replay, but the larger it is the recording overhead. Approaches that opt to log less information, i.e., that perform logging at a coarser granularity, are less expensive, but make it harder to reproduce the original schedule in reasonable time. As such, there is an inherent tradeoff between *logging accuracy* and *replay efficiency*[16].

**Log size:** On the record phase we are saving information to some trace file or log. This log should have adequate size and the amount of space, dedicated to store that log, should be minimal. Hence, another relevant metric of quality of deterministic replay systems is the *space efficiency* during the record phase[16].

**Security issues:** When logging information during production runs, one must take into account that some collected data may be sensitive. This raises security and privacy issues if the log needs to be shared with others (for instance, if the log is sent to the developer). Therefore, the degree of user data disclosure during recording is also considered to be important when devising record and replay solutions. Despite that, in our work, we do not explicitly address this concern, because it is somewhat orthogonal to the techniques that we plan to experiment with.

## 3.2 Design Choices

As it will become clear in the next sections, a large diversity of techniques have been proposed to perform record and replay. To help in discussing the related work, we will often resort to a brief taxonomy, where the following design choices have been identified:

**Implementation.** Deterministic replay techniques are usually classified according to the amount of specialized support that is built in hardware for this particular purpose. In particular, we consider three different types of systems:

- *Hardware-only approaches*[17, 18] are able to provide deterministic replay with little recording overhead, but require expensive non-commodity hardware and tie the approach to a particular architecture.
- *Hybrid approaches*[19], which combine hardware and software techniques, look for a sweet-spot between hardware cost and efficiency but, like pure hardware solutions, tie the approach to specific architectures.
- *Software-only approaches*[5, 6, 4, 7, 20, 21, 10, 11, 13, 10, 9, 8, 12] are usually not as efficient as the ones above, but have the major advantage of being more general. Therefore, they can be applied to many different off-the-shelf architectures.

For self-containment purposes, in Section 3.3, we will make an overview of systems using the three techniques above. However, our work we will be mainly focusing on software-only approaches.

**Record and Replay Approach.** Another way of classifying record and replay systems consists in looking at the type of information that is recorded in the log. Here, it is possible to distinguish approaches as content-based, order-based, and search-based[16]:

- *Content-based approaches*, also called data-driven, log the values of all the relevant inputs and shared variables read by each thread, such that the exact same values can be used during replay. The major drawback of this approach is that it generates very large logs and may induce severe slowdowns in the execution of the program.
- *Order-based approaches*, also called control-driven, instead of tracking the values of variables track the control flow of the program (such as timing of interactions with program files or I/O channels) from a given initial state. According to this method, it is not necessary to record every instruction to replay an execution, which allows to reduce the amount of traced data. However, read and write accesses to shared memory locations still need to be tracked in order to support the reproduction of the thread interleaving. Such tracking is called the *exact linkage* between reads and writes into shared memory.
  A main challenge with this approach is to ensure that the initial state of the program is exactly the same in both the original and the reproduction run. Unfortunately, the initial state may often depend on the availability of external resources such as cores in multicore processor, that could affect the internal state of the program being replayed. Furthermore, to ensure that the log faithfully captures the thread read-write linkage, it is generally necessary to introduce additional synchronization during the record phase. Thus, even if order-based approaches represent an improvement over pure content-based approaches, most of systems implemented with this approach still induce a large overhead at runtime[9, 10] .
- *Search-based approaches* do not log the exact thread read-write linkage. Instead, they rely on a post-recording phase to construct a feasible interleaving

from a *partial log*[9, 8, 12] or infer the missing information. This way, it is possible to substantially reduce the recording overhead at the cost of smaller determinism guarantees and a potentially longer replay. However, since the search space increases exponentially with the amount of missing information regarding the ordering of thread shared accesses, search-based approaches need to carefully balance the inference time and the recording overhead.

Since pure content- and order-based systems incur an overhead too high to be practical, most recent solutions have adopted a search-based approach. Our solution follows this trend, as well.

**Range in Time.** This criterion indicates whether the system requires the replay start point to be predetermined (*static time range*) or if it may be changed (*dynamic time range*). Systems with static time range usually rely on checkpoints, whereas systems with dynamic time range allow to replay an execution backwards.

**Multi-processor and Data Race Support.** In a single-processor system, it suffices to record the synchronization order and the thread scheduling decisions to deterministically replay any concurrent program [6]. The reason is because, in these systems, parallelism is actually an abstraction: only one thread physically executes and accesses memory at a given point in time.

However, in a multi-processor environment (SMP and multicores), providing deterministic replay becomes much more challenging. As threads do actually execute simultaneously on different processors, logging the thread scheduling in each processor is no longer enough to know the exact access ordering to shared memory locations. A way to address this issue is to capture the full thread schedule, which is not easy do to efficiently. This is also the reason why replaying data races is a major challenge for multi-processor deterministic replay systems.

**Immediate Replay.** This criterion indicates whether a system is able to replay an execution at the first attempt or not. For instance, it is common for search-based systems to relax immediate replay guarantees in order to reduce the recording overhead.

## 3.3 Record and Replay Systems

In this section, we overview some of the most relevant record and replay systems, proposed over the years. To ease the description, we present the solutions divided according to the way they are implemented.

**Hardware-only Systems.** Many of the early record and replay systems have been designed considering the use of hardware support to detect data races, often piggybacking on the cache coherence protocol. For instance, Bacon and Goldstein

[22] used a snooping bus to record all cache coherence traffic. In general, there are two main approaches to capture the information in hardware-only systems:

- *Data-driven:* Memory is logically divided into blocks (that consist of one or memory words) and a timestamp is associated with each block. Every time a processor accesses a block, the timestamp is recorded and updated.
- *Path-driven:* The log identifies sequences of data accesses that have been executed without interference from other threads. Each entry in the log, also called a *chunk*, stores the address of all words that have been accessed in the sequence. For conciseness, the set of identifiers that are part of the chunk is stored as a *Bloom Filter*[14]. Data accesses are added to the current chunk until a conflict exists. At that point, the chunk is recorded and a new chunk is initiated.

The major advantage of hardware-only systems is that they allow to achieve deterministic replay with little runtime overhead. Unfortunately, they are impractical in general due to the significant cost and amount of hardware modifications they require. Also, in many cases, one is not interested in logging the entire program but only parts of it. However, supporting selective logging would require even more changes to the hardware.

We briefly summarize some of the most relevant hardware systems in the next paragraphs:

*Flight Data Recorder (FDR)* [18], whose name is inspired in the mechanisms used in avionics to trace flight data, is a system designed to continuously log data about a program's execution with the goal of supporting the replay of the last second of a (crashed) execution. Since the system traces a substancial amount of information, the state of the application is periodically checkpointed, such that the log only needs to preserve the data accesses after the last checkpoint. FDR leverages on previous work for implementing the checkpointing operation, namely on the SafetyNet mechanism[23]. After a checkpoint, FDR logs accesses to the cache. For this purpose, the system maintains an *instruction counter* (IC) for each core and a *cache instruction count* (CIC) field to each cache line; the CIC stores the IC of last instruction that accessed the cache line[14]. FDR implements a number of optimizations[24] to avoid storing information that can be inferred from previous entries. With these optimizations, the authors claim that FDR can induce less than 2% slowdown in the execution of a program. In terms of space overhead, FDR produces logs of 35 MB in an interval of 1,33s; for a longer period, such as 3h of replay, the size of the log would amount to 320 GB.

*ReRun* [25] is a path-driven approach, that identifies and logs sequences of data accesses that do not conflict with other threads. Such sequences, called *episodes*, are stored in a compressed format (using bloom filters as described before) along with a timestamp that identifies the order of the episode with regard to other episodes. The bloom filter that encodes the data accesses during the episode, denoted *episode signature*, is used to detect conflicts among episodes. A conflict exists when one episode tries to access a data item that has been accessed by another episode. ReRun uses Lamport Clocks as timestamps for episodes, i.e., each core keeps a logical clock that is updated, according to Lamport's rules, to keep track of causal dependencies between reads and writes. During replay, episodes are executed according to the order of their timestamps. Although ReRun aimed at reducing the size of logs (when compared to FDR), it requires logical clocks to be piggybacked with every cache coherence operation, which is an additional source of overhead. In particular, the authors claim that ReRun's overhead is approximately 10%.

*DeLorean* [17] is another path-driven hardware system for deterministic replay. As ReRun[25], DeLorean also logs chunks using signatures, but in a different way: in DeLorean's multi-processor execution environment, cores are continuously executing chunks that are separated by register checkpoints. To detect a conflict, the system compares chunks' signatures. However, the updates of a chunk can only be seen after chunk commits. If a conflict is found, the chunk is squashed and re-executed. As with ReRun, replay determinism is achieved by replaying chunks according to their timestamp order.

**Hybrid Systems.** Hybrid systems combine hardware and software support for record and replay. They aim at supporting a wider range of scenarios while reducing the costs associated with building dedicated hardware. In the following paragraph, we refer to one of such systems.

*Capo* [19] uses hardware support to record the interleaving of threads and software support to trace other sources of non-determinism. It also provides support for logging only a subset of the entire program (for instance, the user code but not the operating system code). This is achieved by defining an abstraction named *Replay Sphere*, that encapsulates the set of threads whose operation need to be logged, and by defining the explicit and implicit transitions that allow the core to enter and leave the replay sphere. Capo generates a combined log of 2.5 bits to 3.8 bits per kilo-instruction of the program, which results in a slowdown of the system execution in the order of 21% to 41%.

**Software-only Systems.** There is a great variety of software-only techniques that provide record and replay. In the following, we briefly describe some of the most relevant software-based solutions proposed over the years.

*Instant Replay* [7] can be considered a pioneer software-only record and replay system. It follows an order-based approach: the system records the history of accesses to all the shared objects with respect to a particular thread. As this requires to uniquely identify shared objects dynamically, each object is extended with a custom version number that gets incremented after every write access during both record and replay. The computation model of Instant Replay is based on the CREW (Concurrent-Reader-Exclusive-Writer) protocol, which assigns two possible states to each shared object:

– *Concurrent Read*: implies a state where no processor allowed to write, but all processors are allowed to read the value with no restrictions.
– *Exclusive Write*: implies a state where only one processor (called the *owner*) is allowed to read and write, whereas the processors are deprived of access.

This method makes the record phase quite costly and, due to the small granularity of shared memory accesses, tends to create enormous trace files. It has been reported that performance suffers an overhead up to 10x times.

*DejaVu* [4] is another order-based system, designed at IBM ten years after Instant Replay, with the purpose of providing deterministic replay for Java programs. This system is based on capturing the total order of thread accesses, thus allowing to deterministically replay non-deterministic executions. To this end, DejaVu uses a global timestamping scheme, which exists in two categories:

– *Critical events*, which encompass synchronization points and shared memory accesses, relevant to the record and replay process.
– *Non-critical events*, which are those that only influence the thread where they get executed, so their scheduling is not utterly relevant.

This approach becomes less appealing when the number of threads and cores of a processor increases, as the overhead to capture the global order of all thread events becomes extremely high.

*RecPlay* [6] is a successor of DejaVu, but, unlike the latter, uses Lamport clocks instead of a global clock. RecPlay is based on the assumption that most programs do not have data races, and that the synchronization races are intentional and beneficial. As such, this solution traces threads accesses to only synchronization variables (such as monitor entries and exits). Since these Lamport timestamps are stored in the trace in a compressed from, the runtime slowdown is not very large. In the replay phase, the trace is consulted for every synchronization operation[26]. The drawback of this approach is the loss of determinism (as many shared memory accesses may not be synchronized) and the impossibility to replay problematic runs in the presence of data races[27].

*JaRec* [5] is a portable record and replay system, designed especially for Java applications. This system operates at the bytecode level. Its working principle is very similar to RecPlay[6], as it is also based on assumption that applications are data-race free. In particular, JaRec tracks only the lock acquisition, thus it is not able to reproduce buggy execution caused by data races as well. In other words, this system provides deterministic replay, but only until the first data race condition. This proviso makes JaRec and its predecessor RecPlay[6] unattractive in practice. In terms of performance degradation, JaRec's recording overhead lies between 10% to 125%, depending on the scale of the benchmark used.

*iDNA* [21] is an instruction level tracing framework, based on dynamic binary instrumentation. This system addresses non-determinism by tracking and restoring changes to registers and main memory. In order to do so, it maintains a copy of the user-level memory, which is implemented as a direct mapped cache. iDNA monitors the data values at every dynamic instance of instructions during the execution, and tracks the order of synchronization operations, which means that it does not support the reproduction of data races. An interesting feature of iDNA is the possibility to replay threads independently, as each of them maintains its own copy. But this can also be seen as a source of large log files. In fact, iDNA produces, on average, dozens of mega-bytes per second of trace sizes and incurs runtime overhead of 11x.

*LEAP* [10] is a deterministic replay solution for concurrent Java programs on multi-processors. LEAP is based on the observation that there is no need to track the global ordering of thread accesses to shared memory locations. Instead, it suffices for each shared variable to track only the thread interleaving that it sees (denoted *access vector*). Therefore, this solution produces a set of vectors containing the local access order with respect to the shared memory locations, rather than a single vector with the global order. As each shared variable has its own synchronization, this approach allows accesses to different variables to be recorded in parallel, thus imposing lower runtime overhead. Despite providing slightly weaker determinism guarantees, the authors prove that LEAP's technique does not affect the correctness of the replay.

One of the limitations of LEAP is that does not distinguish different instances of the same class, which creates false dependencies between different objects and a consequent increase on the recording overhead. Nevertheless, the experimental evaluation in [10] showed that LEAP is up to 10x times faster than global order approaches (e.g. Instant Replay [7] and DejaVu [4]) and 2x to 10x faster than JaRec [5], albeit it still incurs huge overhead for applications with many shared accesses. As for space efficiency, trace sizes range from 51 to 37760 KB/sec.

*ORDER* [11] was developed in order to record and reproduce non-deterministic events inside the Java virtual machine (JVM). This system follows an order-based approach and is based on two main observations: *i)* good locality at the object level with respect to thread accesses, and *ii)* frequent object movements due to garbage collection.

ORDER literally records the order of threads accessing shared objects, eliminating unnecessary dependencies introduced by moving objects within memory during garbage collection. It also implements an offline log compression algorithm, used to filter out remaining unnecessary dependencies from thread-local and assigned-once objects, caused by imprecise static compiler analysis. To this end, ORDER extends the header of each object with following five meta-data fields:

– *Object Identifier* (OI) that works as an unique hash of the object.

– *Accessing thread identifier* (AT) and *access counter* (AC), which are used to maintain the current status of the object's access time-line. Every time-line recorded by ORDER can thus be interpreted as *"the object OI is accessed by thread AT for AC times"*.
– *Object-level lock* that is used to protect the whole object and synchronize the recording of the accesses to it.
– *Read-Write flag*, which records whether the current time-line record is read-only or read-write, for future log compression.

In both record and replay phases, ORDER relies on instrumentation actions added to the JVM. At *record* time, the system compares the AT in the object header with the identifier of the current accessing thread (CTID). If the access belongs to the same thread, ORDER increments the corresponding AC. Otherwise, ORDER appends the tuple *(AT, AC)* to the log and proceeds with the execution. During *replay*, the process is similar: the system checks if AT == CTID and if the requesting thread is the expected one (according to the log), it decrements the AC allowing thread to continue executing. Otherwise, the thread gets blocked until its turn.
Performance evaluation results show that ORDER is 1.4x to 3.2x faster than LEAP.

*CARE* [13] is a very recent application-level deterministic record and replay technique for Java concurrent programs. CARE employs an order-based approach that leverages thread locality of variable accesses in order to avoid recording all read-write linkages. Concretely, during the record phase, CARE assigns each thread with a software *cache*. This cache is updated every time the thread reads or writes on a shared variable, and queried whenever the thread performs a read operation. Write operations are always synchronized and recorded into the trace file whenever a new thread writes on a given shared variable, whereas read operations are only logged in the presence of a *cache miss*. A cache miss occurs when the value read from the shared variable differs from the one previously buffered in the cache, meaning that another thread must have written on this variable before. At this point, CARE logs the exact read-write linkage by redoing the read action again with synchronization.

In the replay phase, CARE does not try to determine all non-recorded read-write linkages. Instead, it simulates the behavior of all caches and overrides read values from memory by values buffered in thread-local caches. This provides value-determinism guarantees at replay. Evaluation shows that CARE resolved all missed linkages for sequentially consistent replay, and exhibited 3.4x reduction on runtime overhead and 7x reduction on log size when compared to LEAP.

All the aforementioned systems employ an order-based approach. In the following, we describe some of most relevant search-based solutions.

*PRES* [8] is a search-based record and replay system for C/C++ concurrent programs. Its underlying idea consists of minimizing the recording overhead during production runs, at the cost of an increase in the number of attempts to replay the bug during diagnosis. To this end, PRES records solely a partial trace of the original execution, denoted *sketch* (the authors have explored five different sketching techniques that represent a trade off between recording overhead and reproducibility). Later, in order to reconstruct the non-recorded information, PRES relies on an intelligent offline replayer to search the space of possible thread interleavings, choosing one that fits the sketch. As the search space includes all possible schedules, it grows exponentially with the number of data-race conditions. To address this issue, PRES leverages on feedback produced from each failed attempt to guide the subsequent one and on heuristics to explore the search space efficiently. Most of the times, this mechanism allows to successfully replay bugs in a few number of attempts (1-28 tries according to the experiments). In terms of performance slowdown, authors report an overhead from 28% (for network applications) to several hundred times (for CPU-bound applications.

*ODR* [9], similarly to PRES, relaxes the need for generating a high-fidelity replay of the original execution by inferring offline an execution that provides the same outputs as the production run. In other words, ODR provides the so-called *output determinism*, which authors claim to be valuable for debugging due to: *i)* the reproduction of all output visible errors, such as core dumps and various crashes, *ii)* the assurance of memory-access values being consistent with the failure, and *iii)* the no obligation for the values of data races to be identical to the original ones.

However, this system provides no guarantees regarding the non-output properties of the execution, which makes replaying data races very challenging. To address this, ODR uses a technique, called *Deterministic Run Inference (DRI)*, to infer data-race outcomes, instead of recording them. Once inferred, the system substitutes these values in future replays, thus achieving output-determinism.

As an exhaustive search of the space data races is unfeasible for most programs, DRI employs two facilitating techniques: *i)* guiding the search, which allows to prune the search space by leveraging the partial information recorded at runtime, and *ii)* relaxing the memory-consistency of all possible executions in the search space, which allows to find output-deterministic executions with less effort.

According to ODR's evaluation, while recording causes a modest slowdown of 1.6x, the inference time at the replay phase ranges from 300x to over 39000x the original application time (for ODR's low-recording overhead mode), with some searches not completing at all. Authors do not provide information about trace sizes.

*STRIDE* [12] is a state-of-the art search-based solution that records bounded shared memory access linkages instead of exact ones. Under the sequential consistency assumption, STRIDE infers a failure-inducing interleaving in polynomial time. Its recording scheme logs read operations without adding extra synchronization, which reduces the runtime overhead with respect to pure order-based approaches [10]. Write operations, on the other hand, are still captured in a synchronized fashion. To allow the reconstruction of the global thread schedule, STRIDE logs read operations with write fingerprints. More concretely, STRIDE associates a version number to all the recorded write operations and, for each read operation, STRIDE records a pair consisting of the value returned by the read operation and the latest version of write that read can possibly link to, i.e., the *bounded linkage*. This bounded linkage is later leveraged by STRIDE's search mechanism to quickly find the correct match between reads and writes.

Since STRIDE is the base system of the work in this thesis, we describe its architecture in slightly more detail than the previous systems. The essential concepts of STRIDE are then the *execution log*, the *memory model* and the *legal schedule*. The *execution log* is divided into three parts:

– *LWx*, which is a vector containing the local total order of the writes performed by different threads on the shared variable *x*.
– *LAi*, which registers the order of lock/unlock operations on lock *l*. It allows to reproduce deadlocks.
– *TRi*, which corresponds to the read log of thread *i*.

The *memory model*, in turn, defines the set of values committed by writes that are allowed to be returned by a read. STRIDE uses the most strict memory model for concurrent programs – *sequential consistency* [28]. As defined by Lamport, under sequential consistency, the result of any concurrent execution is the same as if the operations on all the processors are executed in some sequential order and the operations of each individual thread appear in the program order. In this context, the *legal schedule* represents a total order of the read-write operations that conform with the memory behavior rules of STRIDE.

Using these concepts, STRIDE is able to, from an imprecise execution log, generate a feasible thread access ordering such that all read and write operations conform to the sequential consistency memory model. The main advantage of this approach is the low runtime overhead due to the avoidance of additional synchronization when logging reads on shared variables. Authors claim that STRIDE incurs on average of 2.5x smaller runtime slowdown and 3.88x smaller log than LEAP. On the other hand, this approach has the downside of losing some determinism guarantees.

## 3.4 Overall analysis and conclusion

Table 1 summarizes the record and replay systems presented above. The systems are classified according to their implementation type and record and replay approach, along with some additional relevant criteria, namely the range in time, multi-processor support, ability to replay data-races, and ability to immediately replay without a state exploration stage. We also add a short reference to the main features of each system.

It is important to highlight that doing a precise comparative analysis of these systems is almost impossible, since their evaluations have been performed on different benchmarks and using distinct criteria.

| **System** | **Implementation** | **R&R Approach** | **Main feature of System** | **Range in Time** | **Multi-processor support** | **Data Races support** | **Immediate Replay** | **Year** |
|---|---|---|---|---|---|---|---|---|
| Flight Data Recorder | hardware | point-to -point | continuous logging | static (checkpoints) | ✓ | ✓ | ✓ | 2003 |
| ReRun | hardware | chunk | isolate episodes (non-conflicting access sequences) | static | ✓ | ✓ | ✓ | 2008 |
| DeLorean | hardware | chunk | logs chunks using signature | static (checkpoints) | ✓ | ✓ | ✓ | 2008 |
| Capo | hybrid | hybrid | Replay Sphere | static (checkpoints) | ✓ | ✓ | ✓ | 2009 |
| Instant Replay | software | order | full history of accesses | dynamic | ✓ |  | ✓ | 1987 |
| DeJavu | software | order | captures thread schedule | dynamic |  | ✓ | ✓ | 1998 |
| RecPlay | software | order | uses Lamport clock | dynamic | ✓ |  | ✓ | 1999 |
| JaRec | software | order | logs lock acquisitions | dynamic | ✓ |  | ✓ | 2004 |
| iDNA | software | order | tracks changes to registers | dynamic | ✓ |  | ✓ | 2006 |
| LEAP | software | order | uses access vectors for shared variables | dynamic | ✓ | ✓ | ✓ | 2010 |
| ORDER | software | order | mitigates influence of garbage collection | dynamic | ✓ | ✓ | ✓ | 2011 |
| CARE | software | order | uses cache scheme | dynamic | ✓ | ✓ | ✓ | 2014 |
| PRES | software | search | different recording schemes | dynamic | ✓ | ✓ |  | 2009 |
| ODR | software | search | Deterministic Run Inference | dynamic | ✓ | ✓ |  | 2009 |
| STRIDE | software | search | bounded linkage scheme | dynamic | ✓ | ✓ |  | 2012 |

**Table 1.** Summary of the presented systems.

Despite that, it is possible to conclude that, in general, hardware solutions impose very small overheads (up to 2%), but require expensive hardware modifications. Hybrid solutions, in turn, rely on less hardware modifications and are still able to achieve a modest performance overhead (between 20% and 40%). However, they still suffer from compatibility issues, which makes this approach unattractive.

On the other hand, software-only solutions seem to be the most attractive due to the possibility of being easily deployed on commodity machines. Despite that, they require a clear trade-off between strong determinism guarantees and time and space overhead. In single-processor systems, it suffices to log the thread preemption points made by the scheduler to achieve deterministic replay, whereas, for multi-processor systems, racing memory accesses may affect the execution path, hence making deterministic replay much more challenging. As such, since DejaVu records solely scheduling invocations in the Java Virtual Machine, it only provides deterministic replay of multithreaded Java applications on uni-processors.

Among the solutions that strive to provide record and replay for multi-core systems, it is possible to see the variety of a trade-offs when observing Table 1. For instance, Instant Replay aims to provide a global order of operations adding extra synchronization to enforce replay. iDNA, in turn, records all the values read from or written to a memory cell as well as the thread synchronization order, while JaRec and RecPlay abolish the idea of global ordering and use Lamport clocks to maintain partial thread access orders. Unfortunately, they assume that programs are race-free and, therefore, are only able to ensure deterministic replay up until the first race. As for LEAP and Order, two state-of-the-art order-based techniques, they record the exact order of shared memory accesses, but incur high performance slowdown. PRES and ODR achieve smaller recording overhead by means of searching heuristics that explore the non-recorded thread interleaving space, but at the cost of more replay attempts. Finally, STRIDE, a very recent search-based system, introduces a novel technique that relies on bounded linkages to relax the logging of read operations, albeit still requiring synchronization for tracking writes. Regarding time and space overhead, one can argue that, over the years, solutions have been improving these indicators. Nowadays, one considers the a recording technique as efficient if the overhead is generally less than 10% [8].

## 4 Architecture

This works aims to design and implement a prototype of a record and replay system, capable of reproducing concurrency bugs in multi-processors, while incurring low time and space overhead. To achieve this, we plan to optimistically log both read and write operations on shared variables during production runs. In other words, no extra synchronization will be added to the program. This reduces the recording overhead, but allows the captured read-write linkage to be imprecise (because the actual event and the recording operation do not happen atomically). However, since most programs tend to exhibit a strong locality with respect to thread accesses [12], we believe that it will be possible to infer a legal schedule in a reasonable amount of time (polynomial time in practice, with respect to the number of threads and shared accesses). To devise the necessary record and replay algorithms, we will draw on the techniques described in STRIDE [12]. In the following sections, we justify the choice of STRIDE as the basis of this work, as well as describe the architecture of the proposed system.

As a final remark, our work will target concurrent programs written in Java, but we believe that our ideas may be applied to programs written in other languages.

## 4.1 STRIDE Properties

Similarly to STRIDE, our solution will follow a search-based approach, that relaxes the need to record the exact read-write linkage and yet is able to reconstruct a schedule equivalent to the original one. The key aspects that motivated our choice of designing our solution on top of STRIDE are as follows:

– *Software-based approach*: STRIDE is a software-only system and, therefore, does not require expensive hardware modifications nor raises compatibility issues.
– *Bounded linkage*: all systems presented in Table 1, except STRIDE, either log the exact shared read-write linkages at the cost of a large runtime overhead or use off-line heuristics to search for a feasible thread interleaving in an exponential space. STRIDE, on the other hand, was the first system to leverage write fingerprints to record inexact read-write linkages at runtime and still reconstruct the original execution in polynomial time.
– *State-of-the-art overhead indicators*: comparing to the previous deterministic replay solutions, STRIDE was able to, on average, reduce the runtime overhead in 2.5x and to produce 3.88x smaller logs, with respect to pure order-based solutions (e.g. LEAP [10]).

Despite that, we claim that STRIDE is not perfect and that there is still some space left for improvement. For instance, STRIDE requires locking writes (locally) for each shared memory location, which incurs 10x overhead for programs with workloads where 30% of the operations are writes. This cost is still too high for production usage.

In this context, the following question arises: *Is it possible to further reduce the recording overhead by relaxing the logging of shared write operations in addition to read operations, while maintaining the ability to reconstruct the schedule in a reasonable amount of time?* We believe so. Tracing the write order without synchronization may cause conflicts in the write versioning mechanism (e.g. two writes may be recorded with the same version number). However, as showed by the experiments in [12], the read/write operation and the corresponding recording operation tend to occur contiguously for most programs. As such, one should only have to resolve a few write conflicts in order to be able to apply STRIDE's replay algorithm as is. If this is not the case and the number of write conflicts is high, one could also introduce some adaptive mechanism in the record phase to switch between exact and relaxed write logging, thus allowing to automatically tune the recording scheme to the application nature. The next section further explains this idea, by presenting the architecture envisioned for our system.

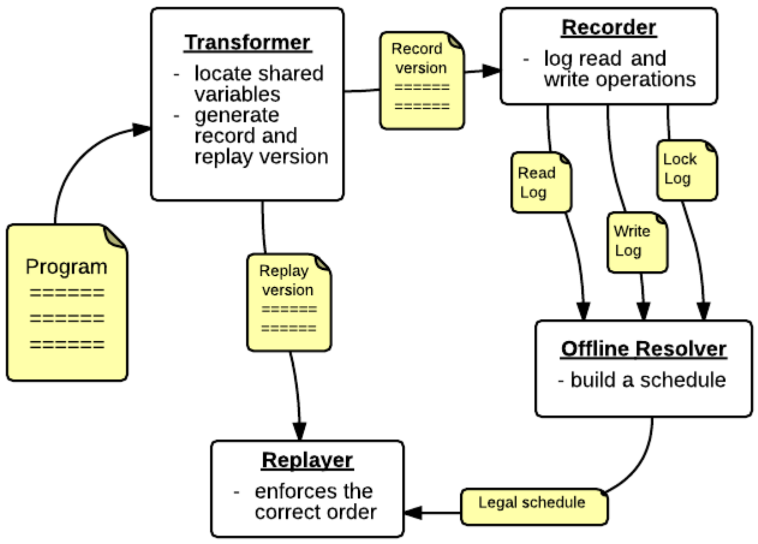

**Fig. 1.** Components of proposed system

## 4.2 Proposed Architecture

Figure 1 illustrates the architecture of the proposed system. In particular, the system consists of four components, namely the *transformer*, the *recorder*, the *offline resolver*, and the *replayer*. We describe each component as follows.

– *Transformer*. This component is responsible for the preliminary analysis of program. In addition to locating the shared variables in the program, the transforms also instruments two versions of the code: a *record version* and a *replay version*, that will serve as input to the recorder and the replayer, respectively.
– *Recorder*. This component executes the record version of the program (previously instrumented by the *transformer*) and stores the relevant events into a trace file. Concretely, the *recorder* produces three types of logs, containing the write operations, the read operations, and the lock acquisition order,

respectively. Unlike STRIDE, we will not introduce any kind of synchronization to log write operations, so writes with the same version will be allowed, for example.

– *Offline Resolver.* This component processes logs produced by the *recorder*, with the goal of producing a feasible execution ordering of events, i.e., the *legal schedule.* To this end, the offline resolver will have to not only use the bounded linkages to infer the correct match between reads and writes, but also resolve potential versioning conflicts among write operations. To address the first issue, we can leverage the techniques employed by STRIDE: for every read operation we will look for the write operation with the same value and a version not higher than the bounded linkage. As for the second issue, we intend to devise novel algorithms.
– *Replayer.* This component is responsible for replaying of the program execution by enforcing the order of thread accesses indicated in the legal schedule, previously generated by the *offline resolver.*

## 5 Evaluation

The evaluation of the proposed solution will be performed experimentally, based on a running prototype. In particular, we are interested in evaluating our system according to the following criteria:

– *Recording overhead*, both in terms of execution slowdown and log sizes.
– *Bug Reproducibility*, to assess whether the system is able to reconstruct the original execution and produce a legal schedule that triggers the concurrency bug.
– *Inference Cost*, to evaluate whether our system will be able to find the correct linkage in a reasonable amount of time. The authors of STRIDE report that its matching algorithm runs in time $O(Kn)$, where $K$ is the number of threads and $n$ the total length of the execution log. However, they claim that STRIDE is able to find the corresponding write for each read in time $O(1)$ for most test cases. As a consequence, we expect that our algorithm will run with time complexity $O(c)$, in practice, where $c$ is the number of write conflicts for a given version number. This because each read can only be matched to one of the $c$ writes conflicting. Nevertheless, in the future, we plan to support this claim with a more thorough complexity analysis.
– *Ratio of Write Conflicts*, in order to understand the impact of tracing write operations without synchronization and assess how the likelihood of finding write versioning conflicts varies with the nature of the application (e.g. the percentage of write operations, the number of threads, and the shared accesses).

We plan to compare our results against STRIDE (and against other systems, if time allows), in order to assess the benefits and limitations of our solution. As test subjects, we will use benchmarks from the *Dacapo*[1] and *Java Grande*[2] suites,

[1] http://www.dacapobench.org
[2] http://www.javagrande.org

as they were already used to evaluate STRIDE. Since these benchmark suites contain programs with different read-write percentages and number of shared accesses, we expect to obtain a good notion of the favorable and unfavorable conditions for our system. As for the bug reproducibility, we plan to use bugs from real-world applications and benchmark programs, such as the IBM ConTest suite [29].

## 6 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, as well as the necessary algorithms to resolve write versioning conflicts.
- March 30 - May 3: Perform an experimental evaluation of the results.
- May 4 - May 23, 2014: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15, 2014: Deliver the MSc dissertation.

## 7 Conclusions

In this report, we have surveyed some representative approaches to record and replay of multithreaded programs, as well as presented the current state of the art by describing several systems. We discussed the main challenges and performance metrics that have to be taken into account. We proposed a new approach to make the logging phase more efficient, using a relaxed logging procedure and offline algorithms to link operations and produce a replay schedule. In addition, we presented the overall architecture of a system able to implement this approach, and described the evaluation methodology that will be used to assess the results.

**Acknowledgments** We are grateful to N. Machado, M. Bravo and H.Miranda for the fruitful discussions and comments during the preparation of this report. This work was partially supported by EUROEAST, an Erasmus Mundus Action 2 project funded by the European Commission.

## References

1. Dijkstra, E.W.: Cooperating sequential processes. Springer (2002)
2. Netzer, R., Miller, B.P.: Detecting data races in parallel program executions. University of Wisconsin-Madison, Computer Sciences Department (1989)
3. Coffman, E.G., Elphick, M., Shoshani, A.: System deadlocks. ACM Computing Surveys (CSUR) **3**(2) (1971) 67–78
4. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, ACM (1998) 48–59

5. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: Jarec: a portable record/replay environment for multi-threaded java applications. Software: practice and experience **34**(6) (2004) 523–547
6. Ronsse, M., De Bosschere, K.: Recplay: a fully integrated practical record/replay system. ACM Transactions on Computer Systems (TOCS) **17**(2) (1999) 133–152
7. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. Computers, IEEE Transactions on **100**(4) (1987) 471–482
8. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009) 177–192
9. Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009) 193–206
10. Huang, J., Liu, P., Zhang, C.: Leap: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ACM (2010) 207–216
11. Yang, Z., Yang, M., Xu, L., Chen, H., Zang, B.: Order: Object centric deterministic replay for java. In: USENIX Annual Technical Conference. (2011)
12. Zhou, J., Xiao, X., Zhang, C.: Stride: Search-based deterministic replay in polynomial time via bounded linkage. In: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press (2012) 892–902
13. Jiang, Y., Gu, T., Xu, C., Ma, X., Lu, J.: Care: cache guided deterministic replay for concurrent java programs. In: ICSE. (2014) 457–467
14. Pokam, G., Pereira, C., Danne, K., Yang, L., King, S., Torrellas, J.: Hardware and software approaches for deterministic multi-processor replay of concurrent programs. Intel Technology Journal **13**(4) (2009)
15. Tanenbaum, A.S., Austin, T., Chandavarkar, B.: Structured computer organization. Pearson (2013)
16. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., Bosschere, K.: A taxonomy of execution replay systems. In: Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, Citeseer (2003)
17. Montesinos, P., Ceze, L., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. In: Computer Architecture, 2008. ISCA'08. 35th International Symposium on, IEEE (2008) 289–300
18. Xu, M., Bodik, R., Hill, M.D.: A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In: Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on, IEEE (2003) 122–133
19. Montesinos, P., Hicks, M., King, S.T., Torrellas, J.: Capo: a software-hardware interface for practical deterministic multiprocessor replay. In: ACM Sigplan Notices. Volume 44., ACM (2009) 73–84
20. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. ACM Sigplan Notices **44**(3) (2009) 97–108
21. Bhansali, S., Chen, W.K., De Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: Proceedings of the 2nd international conference on Virtual execution environments, ACM (2006) 154–163
22. Bacon, D.F., Goldstein, S.C.: Hardware-assisted replay of multiprocessor programs. Volume 26. ACM (1991)

23. Sorin, D.J., Martin, M.M., Hill, M.D., Wood, D.A.: Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on, IEEE (2002) 123–134
24. Netzer, R.H.: Optimal tracing and replay for debugging shared-memory parallel programs. Volume 28. ACM (1993)
25. Hower, D.R., Hill, M.D.: Rerun: Exploiting episodes for lightweight memory race recording. In: ACM SIGARCH Computer Architecture News. Volume 36., IEEE Computer Society (2008) 265–276
26. Ronsse, M., Christiaens, M., De Bosschere, K.: Cyclic debugging using execution replay. In: Computational Science-ICCS 2001. Springer (2001) 851–860
27. Ronsse, M., De Bosschere, K., Christiaens, M., de Kergommeaux, J.C., Kranzlmüller, D.: Record/replay for nondeterministic program executions. Communications of the ACM **46**(9) (2003) 62–67
28. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. Computers, IEEE Transactions on **100**(9) (1979) 690–691
29. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing. IPDPS'03, IEEE Computer Society (2003) 286–293