INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Ditto – Deterministic Execution Replay for the Java Virtual Machine on Multi-processors

João Pedro Marques Silva

Dissertation submitted to obtain the Master Degree in

Information Systems and Computer Engineering

Jury

Chairman:	Prof. Luís Eduardo Teixeira Rodrigues
Supervisor:	Prof. Luís Manuel Antunes Veiga
Members:	Prof. António Paulo Teles de Menezes de Correia Leitão

October 2012

Acknowledgements

This thesis is the result of a year-long journey of many frustrations, successes and discoveries. I could not have made it on my own to the end of this arduous but fulfilling road.

First and foremost, I would like to thank my advisor Prof. Luís Veiga, for offering me guidance throughout my first endeavour in research. I offer him my sincerest gratitude for pointing me to the problem addressed in the thesis, for lending me his knowledge, experience and insight, for nudging me towards ideas I would never have thought of on my own, and for doing all this while allowing me room to work in my own way.

I owe much to the Jikes RVM community, whose dedicated contributors built the platform on top of which I developed my work and promptly answered any questions posted to the researchers mailing list.

I would also like to thank my colleagues at Instituto Superior Técnico, especially Eugénio, Francisco, Amaral and Ricardo, for being much more than workmates during the past five years. I have learned much from all of them.

To my family I owe everything. I thank my father Luís, the best role model for excellence and unshakable values one could have, for taking the time to proof read this document. My mother Eugénia, the embodiment of dedication and unconditional love, for always caring for me when I was too busy to care for myself. My brother Nuno, for patiently sitting through my many rants and frustrations. Last but not least, my grandparents Faustino and Conceição, for being a second set of parents to me and for their dedication to family.

I am grateful to my girlfriend Lisa, for making me forget work when I needed it the most, while remaining patient and understanding when our time together was sacrificed for the sake of the thesis. I thank her for being my ambition when I lacked it, for her invaluable advice throughout the year, for sharing my highs and lows, and for always believing in me.

Lastly, I thank FCT (Fundação para a Ciência e a Tecnologia) for partially supporting my work, by granting me a scholarship under project PTDC/EIA-EIA/113613/2009, and to projects PTDC/EIA-EIA/102250/2008 and PEst-OE/EEI/LA0021/2011.

Lisbon, November 24, 2012
João Silva

”THERE IS AS YET INSUFFICIENT DATA
FOR A MEANINGFUL ANSWER.”
–Isaac Asimov, The Last Question

Resumo

Juntamente com a ascensão das máquinas multi-processador durante a última década, os modelos de programação concorrentes tornaram-se próximos de ubíquos. Programas desenhados segundo estes modelos são vulneráveis ao aparecimento de falhas com pré-condições raras, resultantes de interações não antecipadas entre tarefas paralelas. Além disso, as metodologias convencionais de depuração não se adaptam bem a falhas não determinísticas, levando a esforços de depuração ineficientes em que a maioria dos recursos são gastos em tentativas de reprodução da falha. A técnica de reprodução determinística ataca este problema através da gravação de execuções e uso do resultante rastreio para gerar execuções equivalentes. Os reprodutores conseguem ser eficientes em máquinas uni-processador, mas têm dificuldades com *overhead* excessivo em multi-processadores.

Apresentamos o Ditto, um reprodutor determinístico para aplicações concorrentes da JVM executadas em máquinas multi-processador. Integrando técnicas originais e *state-of-the-art*, o Ditto consegue ser consistentemente mais eficiente que anteriores reprodutores de programas Java, em termos de *overhead* de gravação, *overhead* de reprodução e tamanho do ficheiro de rastreio. A principal contribuição do Ditto é um par de algoritmos originais para gravação e reprodução que (a) gerem as diferenças semânticas entre leituras e escritas da memória, (b) rastreiam acessos à memória ao nível dos campos de instâncias individuais, (c) usam redução transitiva parcial e remoção de restrições baseada na ordem de programa, e (d) tiram partido de análise estática TLO, análise de escape e optimizações do compilador da JVM para identificar acessos à memória locais à tarefa.

Abstract

Alongside the rise of multi-processor machines in the last decade, concurrent programming models have grown to near ubiquity. Programs built using these models are prone to bugs with rare pre-conditions, arising from unanticipated interactions between parallel tasks. Moreover, conventional debugging methodologies are not well suited to deal with non-deterministic faults, leading to inefficient debugging efforts in which most resources are consumed in reproduction attempts. Deterministic replay tackles this problem by recording faulty executions and using the traces to generate equivalent ones. Replayers can be efficient on uni-processor machines, but struggle with unreasonable overhead on multi-processors.

We present Ditto, a deterministic replayer for concurrent JVM applications executed on multi-processor machines. By integrating state-of-the-art and novel techniques it manages to consistently out-perform previous deterministic replayers targeted at Java programs, in terms of recording overhead, replaying overhead and trace file size. The main contribution of Ditto is a novel pair of recording and replaying algorithms that (a) leverage the semantic differences between load and store memory accesses, (b) serialize memory accesses at the instance field level, (c) employ partial transitive reduction and program-order pruning on-the-fly, and (d) take advantage of TLO static analysis, escape analysis and JVM compiler optimizations to identify thread-local accesses.

Palavras Chave
Keywords

Palavras Chave

Reprodução Determinística

Concorrência

Depuração

JVM

Keywords

Deterministic Replay

Concurrency

Debugging

JVM

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This thesis aims to deliver a partial solution to the problem associated with debugging concurrent programs, through the development of a deterministic replay system for the JVM.

1.1 Problem Statement

1.1.1 Ubiquitous Concurrency

For most of computer science's short history, developers have enjoyed very significant software performance boosts with each new generation of processors, thanks to faster clocks and better instruction-level parallelism. During this period, processor improvements translated directly into shorter software execution times; no effort was needed on the developer's part. As a roadblock on processor speed was reached in the past decade, CPU manufacturers were forced to find new ways of improving performance, resulting in a push towards multi-core processors and multi-processor machines. Nowadays, processor replication is the largest source of new computing power. Unfortunately, the latter cannot be directly translated into increased software efficiency; developers must now identify tasks that can be performed concurrently and attributed to different processors. Hence, with the advent of the multi-processor machine, new concurrent programming models have grown to near ubiquity.

1.1.2 The Challenge of Concurrent Programs

The transition from the old sequential paradigm of software development to the new concurrent paradigm has not been the easiest, as software developers struggle to grasp all the implications of parallelism. While sequential programs derive state solely from their inputs, the state of concurrent programs depends on the order in which tasks access shared resources. As a result, concurrent programs are inherently non-deterministic, with even coarse-grained concurrency leading to an amount of possible program states too large for us to fully visualize. Not only are concurrent programs much harder to develop than their sequential counterparts, they are arguably even more challenging to debug. On the one hand, the difficulty involved in reasoning about concurrency makes programs built under the new paradigm very prone to bugs arising from unanticipated interactions between tasks. Furthermore, the pre-conditions of

concurrent bugs are often of so rare an occurrence that even extensive program testing efforts may not reveal them.

On the other hand, though taking on the role of detective to debug complex systems is an all-too-familiar task for programmers, the methodologies developed in the past to deal with sequential programs are not very well suited for non-deterministic concurrent programs. Cyclic debugging is a widely used methodology in which the programmer (a) postulates a set of possible causes for the fault, (b) adds statements to the program which enable the observation of previously hidden program state, (c) uses the ensuing observations to update the set of possible causes, and (d) reiterates until the fault is located [30]. The effectiveness and efficiency of this process depends on one assumption: that the program be deterministic in regards to the fault, i.e., given the same input, different executions of the program should all lead to the fault's manifestation. Although the assumption is reasonable for executions of sequential programs, the same cannot be said for concurrent ones. Indeed, the programmer may need to perform many executions to complete a single iteration of cyclic debugging, significantly lowering its efficiency. Moreover, adding trace statements to the program may contribute to the fault's evasiveness through a phenomenon called probe effect, which may render cyclic debugging ineffective in the worst case, and less efficient in the best. In practice, debugging becomes too time- and resource-consuming, mostly due to the effort associated with bug reproduction. Indeed, a recent study on real world concurrency bugs has shown that the time to fix a concurrent bug is mainly taken up by the task of reproducing it [28].

In summary, debugging concurrent programs using methodologies developed for sequential programs is a far from perfect solution, making debugging efforts on such systems too time- and resource-consuming. This issue is relevant in face of the ubiquity of concurrent programming models resulting from the shift towards multi-processor machines observed in the past decade.

1.1.3 Deterministic Replay

Most solutions that tackle the problem developed in the previous sections are based upon the idea of eliminating non-deterministic behavior in order to re-enable conventional debugging methodologies. Deterministic replay has long been suggested as one such solution, operating in two phases: a record phase, performed during a faulty execution, in which the outcomes of non-deterministic events are traced; and a replay phase, in which a replayer forces other executions of the same program to experience outcomes to non-deterministic events identical to those previously traced. A fitting metaphor for deterministic replay is that of a time machine, because it grants a debugger the ability to inspect past states of a particular execution of a non-deterministic program [23].

To properly understand deterministic replay we must first distinguish between input and memory non-determinism. Input non-determinism results from variations in data used by the program and provided by external sources, present in both concurrent and sequential programs. This kind of non-determinism is tolerated by cyclic debugging whenever input can be reproduced without too much effort, which is not always the case. Efficient software solutions for deterministically replaying input non-determinism have been developed, as evidenced by Flashback [48], Jockey [43], among others.

Memory non-determinism results from the occurrence of data races in concurrent programs, i.e., unsynchronized accesses to the same memory location in which at least one is a write operation. Not all data races lead to faulty states; a subset of them, the synchronization races, are used to implement synchronization primitives that allow threads to compete for access to shared resources. By making some non-trivial assumptions about programs and/or the machines they execute on, deterministic replayers can maintain replay correctness by tracing only the outcomes to synchronization races, whose number is orders of magnitude smaller than that of data races. This technique has been used to develop efficient software deterministic replayers of memory non-determinism that either assume executions occur on single processor machines or that executions are data race free. Examples of such systems are Instant Replay [27], DejaVu [8], RecPlay [41] and JaRec [16].

1.1.4 Deterministic Replay on Multi-processors

The introduction of multi-processor machines makes the problem of deterministic replay much more challenging. The reason is that while parallelism on uni-processors is an abstraction provided by the task scheduler, in multi-processors it has a real-world significance. As a result, replaying scheduling decisions, i.e., outcomes to synchronization races, is no longer sufficient to guarantee a correct replay. One must now record the outcomes of all data races. Because the instructions that can potentially result in a data race make up a significant percentage of a typical program, monitoring them imposes unreasonable overhead. There are currently four distinct approaches to this still open research problem:

1. Replaying solely synchronization races, guaranteeing a correct replay up until the occurrence of a data race [41, 16]. We believe the assumption that programs are perfectly synchronized severely limits the effectiveness of this solution as a debugging tool;

2. Introducing specialized hardware to passively record accesses to shared memory [51, 31]. While efficient, this technique has the drawback of requiring special hardware;

3. Using probabilistic replay techniques that explore the trade-off between recording overhead reduction through partial execution tracing and relaxation of replay guarantees

[37, 4]. These techniques show a lot of potential as debugging tools, but are unable to put an upper limit on how long it can take for a successful replay to be performed;

4. Employing static analysis to identify memory accesses that are actually involved in inter-thread interactions, hence reducing the amount of monitored accesses [20].

Only the first and fourth of these approaches have been employed in the context of Java programs.

1.2 Objectives

We aim to develop a deterministic replay system for the JVM that uses state-of-the-art and novel techniques to achieve the following properties.

Low recording overhead Time and space overheads should be an improvement over existing solutions. The imposed overhead should preferably be low enough for the system to be active in production runs, without severely crippling performance.

Correct user-level replay The system must record and replay non-deterministic events triggered in application code. The replay must be logically equivalent to the recorded execution.

Multi-processor and data race support The system ought to replay executions of applications running in multi-processor machines. Moreover, this property should not compromise the ability to reproduce the outcomes of data races. Indeed, given that data races are at the root of many concurrency bugs, we feel that limiting the system to perfectly synchronized programs is unreasonable.

Easy and inexpensive deployment The system should operate on unmodified binaries, precluding the need to have access to source code, which may be hard to modify or unavailable.

1.3 Contributions

The deterministic replayer developed along this thesis offers the following contributions:

- A novel pair of logical clock-based recording and replaying algorithms that:
  - Leverage the semantic differences between load and store memory accesses to reduce trace data and increase concurrency during replay;
  - Serialize memory accesses at the finest possible granularity, distinguishing between distinct static, instance and array fields;

  - Employ a modified version of transitive reduction to reduce the amount of traced data on-the-fly;
  - Take advantage of thread-local objects static analysis, escape analysis and JVM compiler optimizations to reduce the set of monitored memory accesses;

- A trace file format and a set of optimizations that highly reduce the size of logical clock-based traces.

1.4 Results

In the course of this thesis, we designed a deterministic replayer for the JVM, and implemented it inside the open-sourced Jikes RVM. We named the replayer Ditto after its responsibility to repeat what has happened in the past. Our experimental results show that Ditto consistently outperforms previous state-of-the-art deterministic replayers targeted at Java programs in terms of record-time overhead, trace file size and replay-time overhead. It does so across multiple dimensions, such as number of threads, number of processors, load to store ratio, among others.

1.5 Publications

The work in this thesis is partially described in the following publications:

1. João M. Silva and Luís Veiga, Reprodução Probabilística de Execuções na JVM em Multi-processadores, INFORUM 2012 - Simpósio de Informática, Sep. 2012 [22] (accepted and presented at the conference);
2. João M. Silva and Luís Veiga, Ditto – Deterministic Execution Replay for the Java Virtual Machine on Multi-processors, submitted to ACM EuroSys 2013, European Conference on Computer Systems (under evaluation).

1.6 Document Roadmap

The rest of this document is organized as follows: Chapter 2 surveys and classifies related work on the topic of deterministic replay; Chapter 3 presents Ditto, its architecture, algorithms and data structures; Chapter 4 goes into detail about some aspects of Ditto's implementation in Jikes RVM; Chapter 5 presents the methodology used to evaluate Ditto and the experimental results thereof; and Chapter 6 summarizes the document, listing the main ideas and results to keep in mind, and our thoughts on what future work should focus on.

2 Related Work: Deterministic Replay

Our work builds mainly on the knowledge acquired and reported by previous researchers who delved into the topic of deterministic replay. Research efforts on concurrent debugging, data-race detection and static analysis of parallelism are also relevant to our purposes, but to a much smaller degree. Though a reasonable amount of research has focused on deterministic replay over the past two to three decades, we have found no survey on the topic offering an analysis that is as up-to-date and in-depth as we believe the field deserves. There is a lot a variety in the approaches taken to solve this problem, but previous surveys focus on a very closed set of criteria. Most classify systems in terms of algorithm type (data- vs. order-based) or implementation substrate (software vs. hardware), considering very few additional properties [39, 21, 9, 40, 10]. Thus, one of the contributions of our work is a new taxonomy for deterministic replay systems which we develop along this chapter. We believe this taxonomy enables a better understanding of the subject than previous classification efforts do.

2.1 Deterministic Replay

The main purpose of a deterministic replay system is to eliminate the non-determinism associated with a particular execution of a target program by reproducing the outcomes of its non-deterministic events in subsequent executions. It is important to note that non-deterministic behavior is present in the original execution itself, but not in the latter ones, as there are alternative approaches that make every execution deterministic by default, even if concurrency is present [36]. Replaying a particular execution is usually done in two phases: (i) a record phase, in which the outcomes of non-deterministic events are traced; and (ii) a replay phase, in which the replayer forces another execution of the same program to experience equal outcomes to its own non-deterministic events.

A considerable amount of researchers have worked on deterministic replay systems, but replaying executions in multi-processor machines remains an open problem. Current solutions either make functionality-limiting assumptions or introduce specialized hardware. Since every access to shared memory in multi-processors can be involved in a data-race and lead to non-deterministic behavior, each has to be monitored, introducing a crippling amount of overhead in most applications. Even tough this is the problem tackled by Ditto, the survey and taxonomy

that follow are broader. Indeed, we study many different replay systems, including those that deal with input non-determinism and uni-processor executions.

2.2 Usage Models

Deterministic replay systems have been developed and deployed to enable a wide range of applications.

Debugging. The vast majority of deterministic replay systems have been developed with the purpose of eliminating fault non-determinism that keeps conventional debugging methodologies, such as cyclic debugging, from being effective when dealing with concurrent programs [27, 5, 35, 42, 8, 41, 24, 49, 51, 16, 48, 34, 43, 23, 33, 15, 31, 17, 4, 37, 20]. Some attempt to facilitate debugging by providing mechanisms that offer the illusion of reverse execution [23].

Fault tolerance. Deterministic replay can be used as an efficient means for a fault-tolerant system to maintain replicas and recover after experiencing a fault [7].

Security. Deterministic replay has been used to find exploits for vulnerabilities, run security checks [39] and examine the way attacks on systems had been carried out [11] by enabling system administrators to inspect the state of the system before, after and during an attack.

Trace collection. Trace collection can be made efficient and inexpensive by using deterministic replay technology to compress large execution traces [53].

General replay. Some deterministic replayers have been developed with no particular usage model in mind [12, 32, 52, 19].

2.3 Abstraction Level

One of the most important design decisions to make when creating a deterministic replay system is the level of abstraction at which the execution of the target system will be recorded and replayed. The choice defines not only the scope and power of the replayer, but also the specific sources of non-determinism it will have to face, which we discuss in detail in Section 2.4. Furthermore, the choice will place constraints on the techniques one can use to implement the replay system.

In practice, the abstraction level appears to be either the user or the system levels of the software stack. Among the replay systems that operate at user-level, some replay only application code [42, 49, 48, 15, 17, 4, 37, 27, 35, 8, 41, 24, 16, 20], while others replay shared library code as well [34, 43, 32]. System-level replayers enable reproduction of executions of whole systems, including OS code [5, 51, 33, 52, 12, 31, 19, 7, 11, 23, 53].

Deciding between a software-only or hardware-assisted implementation is also highly conditioned by the abstraction level at which executions are to be replayed. Indeed, if one wants to replay at system-level, the options are either a hardware- or a virtual machine-based replayer. User-level replayers, on the other hand, can and are mostly implemented completely in software.

2.4 Types of Non-determinism

To achieve faithful deterministic replay, one has to record every single source of non-determinism that might cause two executions of the same program to diverge. Sources of non-determinism can be divided into two sets: (1) input non-determinism, which amounts to any kind of input that a program receives from external sources; and (2) memory non-determinism, which arises from the interleaving of parallel tasks and the resulting interleaving of shared memory accesses. The techniques used in recording and replaying these two different sources of non-determinism are very distinct. In fact, we found that the type of non-determinism a system deals with is the most distinguishing criterion when attempting to characterize it. For instance, most properties of memory non-determinism replay systems are not applicable to input non-determinism replay systems. Therefore, in our framework of deterministic replay, systems are first assigned one of two categories: systems that replay input non-determinism or systems that replay memory non-determinism. Some systems do deal with both sources of non-determinism, in which case their two facets will be discussed separately. Because the techniques used to handle the two kinds of non-determinism are quite distinct, they can easily be discussed independently without hindering their analysis or understanding.

2.4.1 Input Non-determinism

Input non-determinism occurs in both sequential and concurrent executions. It can arise from any input delivered to a software layer that is not generated by the same layer. Moreover, input events may be non-deterministic with respect to both their data and timing. For instance, a system call is only non-deterministic in relation to the data it returns or manipulates, since its timing can be derived from program order. On the other hand, interrupt and DMA (Direct Memory Access) operations are additionally non-deterministic with respect to timing, due to their asynchronous nature.

The actual instances of non-determinism are dependent on the level of abstraction with which we consider the target system. Most deterministic replay systems record and replay at either the user [42, 49, 48, 15, 17, 4, 37, 34, 43, 32] or the system level [7, 11, 51, 23, 53, 12, 31], having to deal with very distinct input.

User level At user-level, most input is generated by the OS. Deterministic replay systems must handle the following sources of non-determinism [39]:

- *System calls.* There are a reasonable amount of system calls that are non-deterministic. A prominent and easily understandable example is the UNIX system call `gettimeofday`, which is dependent on timing-related external conditions. System calls that read from disk or a network card are also non-deterministic, because the data present in these devices may change between executions. The data read from networks is particularly difficult to reproduce manually, when compared with data from the disk. Even memory allocation system calls like UNIX's `malloc` are non-deterministic, because their return value is dependent on the current internal state of the OS.

- *API calls.* Sometimes, applications do not invoke system calls directly. Instead, they invoke an high-level API. Java programs, for example, access the system through the Java API. Native programs may also access the system through a library, such as libc. Thus, one may replay the input to the program by recording at the API level instead of the system call level.

- *Signals.* The OS delivers asynchronous signals to applications in order to notify them of a variety of events. The occurrence and timing of such a signal makes the control flow of an execution non-deterministic.

- *Non-deterministic user-level architectural instructions.* The Instruction Set Architecture (ISA) of a processor may contain non-deterministic instructions available in user-mode. The `rdtsc` x86 instruction is one example, since it reads the CPU's timestamp/cycle counter.

- *Stack and dynamic library locations.* Even though not technically an input to the program, the memory locations of both the program stack and dynamically loaded libraries may cause non-determinism.

System level At the system level, input is generated by the hardware itself. The following sources of non-determinism must be handled [39]:

- *I/O.* Any information read by the system from an I/O device is potentially non-deterministic. As examples, the data in a hard drive may be rewritten and the data provided by a network card is clearly timing dependent. When communication with these devices is done through memory mapped I/O, any data read from the assigned addresses must be recorded and reintroduced during replay.

- *Interrupts.* Hardware interrupts cause the processor to stop whatever it is doing, save its context and branch to a routine that handles the particular interrupt being raised.

Such an operation effectively modifies the control flow of the execution. Furthermore, interrupts are asynchronous, meaning the point at which the processor stops cannot be predicted. Thus, both the timing and the contents of each interrupt have to be recorded and reintroduced at the same point upon replay. Note that, in contrast, traps do not have to receive this treatment, because they are raised by the processor itself as a result of a faulty condition when executing an instruction. This means that both the timing and contents of a trap are dependent solely on the particular instruction and its operands. If the replay is successful up to the execution of the instruction, the operands should be the same during replay as during recording and an equal trap will be naturally raised by the processor.

- *Direct Memory Access (DMA).* Direct memory accesses allow devices to write directly to memory, bypassing the processor. Therefore, in the processor's point of view, they are asynchronous events. Just like interrupts, both their timing and written values must be recorded.

- *Non-deterministic architectural instructions.* The results of executing any non-deterministic instruction featured in the processor's ISA also have to be recorded.

2.4.2 Memory Non-determinism

In contrast with input non-determinism, memory non-determinism is unique to concurrent executions. The phenomenon behind this kind of non-determinism is called a data race, which occurs whenever (a) there are two unsynchronized accesses to the same shared memory location and (b) at least one of those accesses is a write operation.

Synchronization races are a subset of data races used to implement synchronization primitives and are, thus, intentional and beneficial. They allow for competition between threads to access a critical region or lock a mutex, for example. Removing synchronization races would turn a concurrent execution into a sequential one. Nevertheless, the outcome of these races must be recorded. In fact, recording and replaying synchronization races is enough to replay any concurrent program running on a uniprocessor system. This is the case because, in such systems, the parallelism between threads is just an abstraction, as only one thread may execute and access memory at a given point in real time. Thus, the outcome of all data races can be derived solely from the task scheduling decisions.

Data races, on the contrary, are the reason behind many concurrent bugs. They usually arise from faulty or non-existent synchronization between accesses to shared memory. Data races make the job of a deterministic replay system a lot harder on multi-processor machines, because when multiple processors exist, parallelism is no longer a simple matter of abstraction,

but a real physical phenomenon. Therefore, the outcome of data races stops being solely dependent on scheduling decisions; knowing which tasks are executing on each processor at a given point in time is not enough to know which one will win a particular data race. We are now face-to-face with a situation in which any pair of accesses to shared memory is a potential data race, the outcome of which must be recorded. This is the major problem that deterministic replay systems face today, since monitoring every memory access incurs crippling time and space overhead, especially for software-only solutions.

Any replay system capable of reproducing the outcome of data races can also reproduce the outcome of synchronization races. Nonetheless, the concept of a synchronization race is beneficial for deterministic replayers that make assumptions about the target system. If the latter is either executing on a uni-processor machine or is free of data races, replaying outcomes to synchronization races is enough for deterministic re-execution. Since synchronization operations make up a much smaller fraction of total executed instructions than shared memory accesses do, the distinction enables the creation of efficient software-only replayers for multi-processor executions.

2.5 Replay Start Point

Before delving into the record and replay techniques that enable deterministic replay of input and memory non-determinism, let us consider the starting point of a replay. The simplest approach is to start replaying the execution from the very beginning, which is fine when it is short-lived or the events of interest occur early on. For long executions, however, this might become a problem. If a bug only manifests after a program has been running for three days, a programmer would take at least the same three days to complete an iteration of cyclic debugging. This encouraged replay system designers to develop and/or deploy checkpointing techniques (e.g., transparently and incrementally [44]) to allow a replay to start at arbitrary points of an execution. One can think of a checkpoint as a compressed execution trace, allowing the replay system to fast-forward through parts of the execution that are known to contain no events of interest. Additionally, checkpoints are mandatory if a functional requirement is to provide the illusion of reverse execution in an efficient way. We now survey the techniques used in deterministic replay systems to create checkpoints and enable multiple replay starting points.

Flashback [48] is a user-level replayer that uses shadow processes to create checkpoints and efficiently roll back the state of a target process. The user (or automated debugger) can request a checkpoint at any point during the execution. Flashback then creates a snapshot of the process, stores it as a shadow process structure in the kernel, and immediately suspends it. The user can then go back in time to the point when the shadow process was created and Flashback is

able to replay from that point forward. The overhead of restoring checkpoints is reduced by restoring pages using a copy-on-write policy.

liblog [15] is a user-level, library-based replay system with a checkpointing mechanism based on libckpt [38]. This library writes allocated memory regions in a checkpoint file. A bootstrap application reads this file and overwrites its own memory with its contents, effectively becoming a copy of the program at the point the checkpoint was created.

Jockey [43] is another user-level, library-based replayer. It creates checkpoints using a technique based on Flashback and libckpt. The target process is forked and the child creates the checkpoint file, while the original process continues running.

FDR [51] is a hardware-based replayer that enables whole-system deterministic replay of the second leading up to a fault. Due to its somewhat unique goal, checkpointing is mandatory and must be performed often. FDR was tested with the SafeyNet [47] checkpointing mechanism. The checkpoint itself contains the architectural state of all processors, an image of physical memory and I/O state. Because the image of physical memory is large, FDR incrementally creates logical checkpoints by saving the values of memory locations that are overwritten. A checkpoint can then be recovered from the system's final state by undoing changes made to memory. Due to the nature of FDR, checkpoints can be discarded when a new one is created. Thus, even though multiple checkpoints are created as the system executes, FDR can only start replaying from the most recent onwards.

BugNet [34] is another hardware-based replayer, but it records at user-level. It uses the notion of a checkpoint interval to achieve replay of concurrent programs. At the beginning of an interval, the architectural state (program counter and register values) is saved. From then on it saves the values of memory locations only when they are accessed for the first time since the interval began. This amounts to a reduction in both log size and hardware cost over FDR.

TTVM [23] is a virtual machine-based, system-level replayer. Its checkpoints comprise a complete state of the virtual machine: CPU registers, physical memory and virtual disk, among others. To improve efficiency, a copy-on-write policy is used on both memory and disk. TTVM is able to undo or redo operations, enabling time-travel between checkpoints.

2.6 Replaying Input Non-determinism

Though sequential programs do not exhibit memory non-determinism, they are still exposed to input non-determinism. How, then, have programmers gotten away with debugging them using techniques like cyclic debugging for so long? The reason is that, in many cases, input can be reproduced with relative ease – files can be restored, network activity can be created manually and signals or interrupts are rarely a problem. It is only once reproduction of in-

Table 2.1: Overview of input non-determinism deterministic replay systems.

	Bressoud & Schneider [7]	jRapture [49]	ReVirt [11]	FDR [51]	Flashback [48]	Jockey [43]	TTVM [23]	liblog [15]	ReTrace [53]	SMP-ReVirt [12]	DeLorean [31]	R2 [17]	Capo [32]	ODR [4]	PRES [37]
Abstraction Level															
System	×		×	×			×		×	×	×				
User + Library						×							×		
User		×			×			×				×		×	×
Type of Inputs															
System Calls					×	×		×					×	×	×
API Calls		×				×		×				×			
Signals						×		×					×	×	×
Non-deterministic Instructions	×		×			×	×		×	×				×	
I/O	×		×	×			×		×	×	×				
Interrupts	×		×	×			×		×	×	×				
DMA	×		×	×			×		×	×	×				
Start Point															
Static	×	×	×	×								×	×	×	
Dynamic					×	×	×	×	×	×	×				×
Implementation															
Hardware				×							×				
Software															
Library-based		×				×		×				×			
Binary Instrumentation															×
OS Modifications					×								×	×	
VM Modifications	×		×				×		×	×					
Usage Model															
Debugging		×		×	×	×	×	×			×	×		×	×
Fault-Tolerance	×														
Security			×												
Trace Collection									×						
General Replay										×			×		

put is impossible or a resource heavy task that deterministic replay of input non-determinism becomes relevant. To achieve this goal replayers must make sure the program perceives no difference in its interaction with external resources during re-execution. The solution is to trace the inputs listed in Section 2.4.1 during the record phase and inject them back upon replay.

The following sections survey real systems that have the goal of replaying input non-determinism at user-level and system-level. Table 2.1 summarizes the surveyed systems according to the criteria used in our taxonomy to classify input non-determinism replayers.

2.6.1 User-level Replay

At this abstraction level, the major sources of non-determinism are system calls and signals. We focus on techniques that enable their replay. Non-deterministic instructions are a somewhat

lesser problem.

Flashback [48] records system call level input using kernel modifications. More specifically, system calls are hijacked by replacing the default handler for each one with a wrapper function that handles the logging and replaying. The results and side-effects of each system call are logged when recording and injected back in during replay. System calls that affect the application's state only, such as `gettimeofday` or `getpid`, are the easiest to replay. They need not be re-executed by the OS, meaning the call can be bypassed and the program's state is simply modified according to the logged results. Other system calls change the state of the OS itself and need to be re-executed during replay in a way that ensures the same state modifications for the application that had occurred during the record phase. The syscalls `malloc` and `fork` are examples of this latter case. Creating the wrappers for all system calls is a tedious and far from general solution, as each recorded routine must be paid special attention.

Flashback does not handle signals. Nevertheless, the authors propose using the approach described by Slye and Elnozahy [45] to achieve deterministic signal reproduction. In this approach, a signal is annotated with the increment suffered by an instruction counter, available in some architectures, since the last asynchronous event occurred. This would uniquely identify the timing of the signal.

Capo [32] is a hybrid software-hardware system that aims at replaying application and shared library code. It can reproduce both input and memory non-determinism, with the former being a responsibility of the software part of the system. CapoOne, its prototype implementation, takes advantage of small kernel modifications and uses the Linux `ptrace` process tracing mechanism to control the target processes. The mechanism for dealing with system calls is equal to the one used in Flashback. It improves upon Flashback in that it also replays signals, but the mechanism for capturing their exact timing is not specified the respective publication.

ODR [4] records system calls and non-deterministic instructions by having a signal delivered to itself (in the context of the process being executed) by a small set of kernel modifications whenever the target system performs such actions. System calls are replayed by around 200 manually written stubs, but, unlike Flashback, these are executed when handling the signals delivered by the modified OS, not by substituting the default system call routines.

Jockey [43] differs from Flashback and Capo in that a runtime user-mode library is injected into the target application to enable deterministic replay. Jockey logs a mix of system calls, libc calls and non-deterministic instructions by replacing them with calls to stubs that handle the recording and replaying. The correctness of Jockey can be broken due to the way the timing of signals is recorded. These are associated with the closest successive stub call, instead of a point in time. During replay, the delivery of the signal to the application is, thus, delayed until that next stub call completes. In practice this approach may have a very high probability of reproducing program state, but it breaks full correctness nonetheless.

liblog [15] is another library-based replay system. It uses very similar techniques to the ones employed by Jockey to record system calls, libc calls and signals. It does not seem to record non-deterministic instructions.

jRapture [49] is a replay tool for Java programs. It operates by recording at the Java API level, as most interactions between a Java application and the system are done through this interface. It is implemented as a set of modified versions of the Java API classes. Just like system calls, methods in these classes can have side-effects that span further than their return value. As a result, each modified class must be written by hand. jRapture can also replay native methods that are called through the Java Native Interface (JNI) at the cost of becoming platform dependent.

R2 [17] provides a different approach to library-based replay systems. Jockey, liblog and jRapture all log statically defined interfaces: system calls and libc calls for the two former and the Java API for the latter. In contrast, R2 enables recording and replaying of a user-defined interface. Anything above the interface is re-executed during replay, while anything below is bypassed and the results read from a log. Choosing the right interface is a trade-off between the amount of information that is logged and the detail of the replay. The higher the level is, the lesser detail the replay has, as big chunks of execution are bypassed. This allows for a curious phenomenon in which bug symptoms are reproduced, but not the bug's root cause. Since the interface is user-defined, the stubs for each chosen function must be created on-the-fly. *R2* provides the user with an annotation language through which the side-effects of a function can be made explicit, enabling the creation of the stubs.

PRES [37] is a very recent system focused on replaying concurrent programs on multiprocessors. It replays system calls and signals, at least. The handling of input non-determinism is downplayed in its publication and is present for completeness reasons. This is indicative of the fact that replaying input non-determinism is considered a solved problem.

2.6.2 System-level Replay

Replaying at system-level imposes more constraints on the implementation options of deterministic replay systems. It is not possible to record all system-level events using a software-based solution that runs in user mode. Thus, it comes as no surprise that systems replaying at this abstraction level are implemented as either hardware modifications or inside virtual machines. The input that needs to be recorded includes non-deterministic instructions, I/O, interrupts and DMA operations.

Bressoud & Schneider [7] pioneered the idea of using virtual machine technology to achieve deterministic replay of whole systems. They use execution replay to enable a high-availability primary-backup system in which the primary machine is recorded and the backup systems use deterministic replay to mimic the primary. With this setup, the backups accompany the state

changes of the primary with absolute faithfulness and are ready to take over in the event of failure. Space overhead is not a problem, because when all backups have replayed a certain part of the execution, the corresponding log portion can be discarded.

To record the timing of asynchronous events like interrupts or DMA operations, their system uses the *recovery register* of the HP's PA-RISC architecture. This register is decremented whenever an instruction is executed and an interrupt is delivered to the Hypervisor when it becomes negative. This behavior allows the system to regain control at a very specific point in the execution and deliver a virtual interrupt to the backup, for example.

ReVirt [11] also takes advantage of a virtual machine, but with the goal of allowing the system administrator to inspect the execution during an attack to the system. Input from external devices, non-deterministic system calls to the host OS and non-deterministic instructions are logged. The point at which a virtual interrupt is delivered to the guest is uniquely identified by combining the program counter with the hardware retired branches counter. The instruction counter points to a specific instruction in the system's code, but not a specific execution of that instruction, since subsequent branches may lead to it being executed multiple times. However, in combination with the retired branches counter, which is incremented whenever a branch instruction executes, it uniquely identifies a specific point in an execution.

TTVM [23] and ***SMP-ReVirt*** [12] are two more virtual machine-based replay systems used to debug operating systems and general replay, respectively. They handle input non-determinism with the same techniques that ReVirt uses.

ReTrace [53] was developed by *VMWare* to reduce the overhead of collecting arbitrarily complex traces of production executions using deterministic replay. It records all input to which a virtual machine is subjected. Asynchronous events are associated with a point in time by keeping track of the number of instructions executed.

FDR [51] is a hardware-based replayer. It records I/O by storing load values, and interrupts by using an instruction counter to uniquely identify their timing. As for DMA writes, FDR models each DMA interface as a pseudo-processor and uses the same algorithm that handles memory races (discussed in Section 2.7.5). This is possible because DMA operations use the same directory protocol to maintain cache coherence as processors. The trace is kept in hardware buffers, as FDR only replays the second previous to a system crash.

DeLorean [31] is another hardware-assisted replayer based around the notion of a chunk, a set of instructions that execute atomically. It uses a shared DMA log and two per-processor logs for I/O and interrupts. Like FDR, it models DMA interfaces as pseudo-processors and makes them go through the same chunk commit protocol that processors use to record memory non-determinism (details in Section 2.7.5). Interrupt timing is identified by the `chunkID` of the chunk that initiates execution of its interrupt handler.

2.6.3 Software vs Hardware Approaches

Hardware approaches like *FDR* and *DeLorean* achieve the lowest performance overhead. For instance, when recording an execution of the Apache web server, *FDR* has a performance overhead below 2%, including memory non-determinism logging. In spite of this superiority, software approaches have been shown to enable deterministic replay of input non-determinism quite efficiently, achieving performance overheads below 10% during the record phase [23, 48, 53, 11].

Comparing approaches in regards to space overhead is not straightforward. Replay systems differ in events logged, compression schemes and use different benchmarks in their evaluation. Nonetheless, *Flashback* claims its log size grows linearly with the number of system calls the target program issues. We would expect most other systems' logs to grow in a similar fashion: linearly with the number of events they log.

The slightly better recording performance does not seem to justify the cost involved in employing hardware support. This conclusion is supported by the fact that only two of the surveyed systems that handle input non-determinism are implemented in hardware. Furthermore, both support input replay for completeness purposes, as their reason to be is memory non-determinism replay.

2.7 Replaying Memory Non-determinism

Recording input only enables deterministic replay of sequential programs or concurrent programs in which tasks do not interact. When tasks communicate with each other through whichever means, they become dependent on each other and exhibit memory non-determinism.

In this section we will first present the criteria we use to classify systems targeted at memory non-determinism in regards to (1) the model of its target system/program, (2) the recording mechanism and (3) the replaying mechanism. We then survey real replay systems that handle memory non-determinism. Table 2.2 summarizes the characteristics of each system by classifying them according to our taxonomy.

2.7.1 Target System Model

2.7.1.1 Multi-processor Support

There is a very significant difference between a concurrent system execution on a uniprocessor and on a multi-processor. In a single processor machine parallelism is only an abstraction, as

Table 2.2: Overview of memory non-determinism deterministic replay systems.

	Instant Replay [27]	Bacon & Goldstein [5]	Netzer's TR [35]	Russinovich & Cogswell [42]	DejaVu [8]	RecPlay [41]	FDR [51]	JaRec [16]	BugNet [34]	Strata [33]	liblog [15]	RTR [52]	SMP-ReVirt [12]	DeLorean [31]	ReRun [19]	Capo [32]	ODR [4]	PRES [37]	LEAP [20]
Target System Model																			
Multi-processor Support	×	×	×			×	×	×	×	×		×	×	×	×	×	×	×	×
Data Race Support		×	×	×	×		×		×	×	×	×	×	×	×	×	×	×	×
Dynamic Task Creation	×	×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Abstraction Level																			
System		×	–				×			×		×	×	×	×				
User + Library			–						×							×			
User	×		–	×	×	×		×			×						×	×	×
Record Mechanism																			
Traced Events																			
Shared-memory Accesses		×	×		×		×		×	×		×	×				×	×	×
Synchronization Ops.	×				×	×		×									×	×	×
Schedule				×							×								
Conflict-free Intervals														×	×	×			
Algorithm Type																			
Data-based									×										
Order-based	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×
Sharing Identification																			
High-level Constructs	×			–	×	–		–			–								
Dynamic		×	×	–		–	×	–	×	×	–	×	×	×	×	×	×	×	
Static				–		–		–			–								×
Trace Optimization		×	×		×	×	×	×	×	×		×	×		×			×	×
Replay Mechanism																			
Determinism																			
Value	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			×
Output																	×		
Conditional																		×	
Optimistic/Probabilistic																	×	×	
Dynamic Start Point									×		×	×	×	×				×	
Implementation																			
Hardware		×	–				×		×	×		×		×	×	×			
Software																			
Library-based	×		–			×					×								
Binary Instrumentation			–					×									×	×	×
OS Modifications			–	×	×														
VM Modifications			–										×						
Usage Model																			
Debugging	×	×	×	×	×	×	×	×	×	×	×			×			×	×	×
General Replay												×	×		×	×			

the one processor cannot execute two instruction streams simultaneously. With more than one processor, the opposite is true: tasks may execute concurrently in a real, physical sense. Thus, while in a uniprocessor the interleaving of tasks is only dependent on the points at which tasks are swapped by others, in a multi-processor tasks can interleave in very complex ways. Indeed, since the processors offer no guarantees about the time taken to execute a portion of code, knowing when each task was swapped in is not enough to derive the interleaving.

Due to the increased complexity involved in replaying systems executing on multiple processors, some replay systems focus only on uniprocessors [42, 8, 15]. Some replayers designed for uniprocessors can actually replay multi-processor executions, e.g. DejaVu [8], but

they serialize all synchronization and memory operations into a single stream – a global order that effectively simulates an execution in a single processor machine. In our classification we do not consider these systems as supporting multi-processor replay. Despite the involved complexity, most surveyed systems support replay on multiple processor machines [27, 5, 35, 41, 51, 16, 34, 33, 52, 12, 31, 19, 32, 4, 37, 20].

2.7.1.2 Data-race Support

A data race is said to occur whenever two concurrent tasks access the same shared memory location without synchronizing with one another and at least one of them performs a write operation. A subset of data races, the synchronization races, are defined as data races that occur on memory locations used to implement synchronization mechanisms, such as a spin lock. The set of synchronization races is orders of magnitude smaller than the set of data races and, as a result, it is much more efficient to trace the former than the latter. Tough the outcomes of all data races have to be reproduced for a completely faithful deterministic replay, in uniprocessor machines these can be derived from the outcomes of synchronization races. Thus, any replayer that does not tackle the problem of multi-processors is classified as supporting data races.

Data races need only be recorded explicitly when the replay system tackles executions on multi-processors. Our survey shows that most replayers which support multi-processor executions also provide some form of support for data race replay [5, 35, 51, 34, 33, 52, 12, 31, 19, 32, 4, 37, 20]. Some, on the other hand, choose to avoid the overhead of recording data races by placing a constraint on the target system: it must be perfectly synchronized [27, 41, 16]. Nonetheless, these replay systems can faithfully replay an execution up to the point at which the first data race occurs. In addition, they may be coupled with a data-race detector to enable debugging of imperfectly synchronized programs [41].

2.7.1.3 Task Creation Model

A replay system's internal mechanisms may place constraints on how tasks are created in the target system. We consider two models for task creation: (a) a static model in which the number of tasks is fixed and known a priori, and (b) a dynamic model in which tasks can be created and destroyed freely throughout the recorded execution.

We found that most replayers do not force the target system to conform to a static model of task creation, with all user-level replayers supporting dynamic creation of tasks. Since system-level replayers are based on recording the behavior of processors instead of individual user-level processes or threads, they inherently allow for dynamic task creation. They mostly do, however, assume a static number of processors, which is a reasonable limitation. Only Capo

[32] takes measures to enable a variable amount of processors, but only between the record phase and the replay phase. From all the surveyed systems, only Netzer's Transitive Reduction algorithm [35], which was never implemented by its authors, forces the target system to conform to the static task creation model. This is due to its use of vector clocks, which contain a fixed number of positions, one for each task [25, 29]. The algorithm could, however, be subject to extension by using dynamic vector clocks to handle task creation and termination, at the cost of larger space and time overheads [26]. An implementation of Netzer's TR was developed for the system-level, hardware-based replayer FDR [51], but it was modified to use scalar instead of vector clocks [25, 29].

2.7.2 Recording Mechanism

2.7.2.1 Algorithm type

The algorithms used to record memory non-determinism can be classified as either content-based or order-based.

Content-based The most straightforward approach is a pure content-based algorithm: a recorder simply stores the data read by each instruction in a log file and the same data is then fed back in during replay. Such an approach may generate logs of unreasonable size, making it too inefficient for any practical use. Despite this apparent impracticability, some deterministic replay systems have managed to follow a content-based approach and achieve reasonable efficiency by recording only a subset of the data read by instructions [34].

There is a benefit which is unique to this sort of recording algorithm: one is able to replay subsets of or even individual tasks. Since all memory input is recorded, the tasks that generated it do not necessarily have to be running. One may, however, argue that this is actually a disadvantage for debugging purposes, as replaying tasks in isolation can make it harder to analyze the complex interactions they may have had with others during the recording phase [27].

Order-based Instead of tracing the state observed by each instruction, order-based approaches take advantage of the fact that shared memory already contains most of that state. We need only make sure that each instruction sees the shared state at the correct moment during re-execution. In concurrent programs, tasks influence the shared state by writing to shared memory, while their own behavior can be conditioned by other tasks when they read from shared memory. Thus, in an order-based algorithm, the recorder traces the relative order of critical events, such as shared-memory accesses or synchronization operations. During replay, the runtime environment is set up in an initial state equivalent to the one of the recorded execution. Then, the program executes and, whenever a situation arises in which the execution could

deviate from the original, the log is used to nudge it in the right direction. In other words, the tasks are forced to access shared memory in the same order as in the original execution, forcing the data in memory to undergo the same chain of read and write operations, as well as the corresponding sequence of states.

The main advantage of this approach is that most of the data read by instructions is reproduced by the program itself and needs not be recorded. This can also be seen as a shortcoming, since instructions can no longer be executed in isolation. Nonetheless, a total order between all critical events generally takes up lesser trace bandwidth than recording all the data read from memory by executed instructions. A reflection of this fact is that all but one of the surveyed memory non-determinism replay systems use an order-based algorithm [27, 5, 35, 42, 8, 41, 51, 16, 33, 15, 52, 12, 31, 19, 32, 4, 37, 20].

No matter what approach we use, their purest forms are too inefficient for practical purposes. They record at too low an abstraction level, resulting in a lot of trace data. Only by raising the level of abstraction and designing techniques to trace subsets of all events can a deterministic replay system be deployed in production environments.

As a final note, the order-based approach is only available when replaying memory non-determinism. A replayer for input non-determinism cannot guarantee that external sources inject the right input into the target system, at the right time. The only solution is, therefore, to store the contents of those inputs.

2.7.2.2 Traced events

Recording memory non-determinism can be achieved by tracing different kinds of events.

Shared-memory Accesses Unsynchronized shared-memory accesses are behind every instance of memory non-determinism, though only if one of the accesses is a write operation. Systems that directly trace accesses to shared memory generally support both data races and multi-processor executions.

Synchronization Operations Another way of recording memory non-determinism is to trace the order of synchronization operations, such as those that manipulate mutexes or monitors. This type of recording only provides enough information to reproduce synchronization races. As a consequence, no system that records solely the order of synchronization operations is able to simultaneously support multi-processor executions and data races [41, 16].

Task Schedule Reproducing the task schedule of an execution is enough to replay it, given that it occurs in a uni-processor machine [42, 15]. It is also enough for executions in multi-processor machines if the program is perfectly synchronized. Therefore, tracing task schedules is about as powerful a method as tracing synchronization operations.

Chunk Commits Recent hardware replay systems use a chunk-based approach, in which the order of chunk commits is traced [31, 32, 19]. A chunk represents a block of instructions that are executed without conflicting with each other in terms of memory accesses. Thus, the order of memory accesses is derived implicitly from the order of chunk commits.

2.7.2.3 Sharing Identification

Recorders that trace memory accesses often employ techniques to distinguish between accesses to shared and to local memory. If this cannot be achieved, every memory access has to be considered shared by default. There are three ways of detecting shared access events used in the surveyed systems.

High-level Constructs If all accesses to shared-memory are done through well-defined high-level constructs at either language or OS level, the work of the recorder is very simplified [27, 8]. This approach makes the replay system dependent on a particular protocol for accessing shared objects.

Dynamic Most recorders detect accesses to shared-memory dynamically, as the system executes. There are a lot of techniques that enable this approach, from using scalar or vector clocks [41, 16, 8, 35] to spying on cache coherence messages [5, 51, 33, 19] and using hardware page protections [34], among others.

Static One surveyed system [20] uses static analysis on Java programs prior to their record execution. The analysis identifies a conservative set of object fields that may be shared by concurrent threads.

2.7.2.4 Trace Optimization

Many of the recorders we surveyed apply optimization techniques to their trace files in order to reduce their sizes.

A very common optimization for systems that record shared-memory accesses is transitive reduction, which identifies redundant constraints on the ordering of accesses and removes them from the log. Netzer [35] proposed an algorithm to find the optimal set of constraints that are sufficient for a correct replay. The slightly modified version of this algorithm is used in FDR [51]. An improvement over this algorithm, called Regulated Transitive Reduction (RTR) [52], introduces artificial constraints to further reduce the set that needs to be explicitly stored in the trace file. Systems that use transitive reduction optimization include FDR [51], PRES [37] and SMP-ReVirt [12].

Another frequent optimization is to record intervals instead of individual events. As an example, if a recorder logs shared-memory accesses, and multiple successive accesses are done

by the same task, these may be represented as an interval. DejaVu [8], Bacon & Goldstein [5] and LEAP [20] employ this kind of trace file compression.

A third optimization is to represent timestamps as increments, instead of as absolute values. A further improvement is to eliminate increments by taking a certain increment (e.g. $+1$) as the default one and tracing only those which are different. RecPlay [41] and JaRec [16] use this optimization.

Finally, data-based approaches can take advantage of the fact that in the absence of external entities, the target system can regenerate the values read by most instructions without the help of the replay system. BugNet [34] uses checkpoint intervals to identify parts of an execution that meet this criterion and records only the value read by the first load operation of the checkpoint interval to each memory location.

2.7.3 Replay Mechanism

2.7.3.1 Determinism

All memory non-determinism has its roots on data races. No matter whether all of them are recorded or how the recording is accomplished, replayers may provide different levels of replay fidelity. Even though research on record and replay technology has largely had high fidelity replay as an objective, some recent solutions attempt to relax this guarantee in order to reduce recording overhead.

Value Determinism Under value determinism, the replayed execution reads and writes the same values to and from memory, at the same execution points, as the recorded execution [4]. This kind of determinism has been used from the very first deterministic replay systems all the way to very recent work [27, 5, 35, 42, 8, 41, 51, 16, 34, 33, 15, 52, 12, 31, 19, 32, 20]. It provides a high fidelity replay of the original execution. It is not the maximum possible fidelity guarantee however, as the replayed execution is usually allowed to diverge from the original in terms of task local state.

Conditional Determinism Value determinism allows replayers to succeed every time, but may require a lot of trace information to be generated by the recorder, especially for multiprocessor machines. Without deterministic replay, the exact opposite is true: there is virtually no guarantee that the original execution will be reproduced and the record-time overhead vanishes. The authors of PRES [37] decided to explore the space between these two extremes by relaxing the guarantees of value determinism to reduce recording overhead. The original execution is only traced partially, meaning that the replays may not conform with the original and the fault (PRES is aimed at debugging) may not be reproduced. The solution used is to perform multiple replay attempts that conform with the partial trace and to use user-provided

conditions to decide whether each attempt is successful. In short, conditional determinism guarantees that replayed executions conform with the original in terms of the partial trace and user-provided conditions.

Output Determinism. The authors of ODR [4] proposed a replay system that provides output determinism, i.e., the replayed execution's output is equal to that of the original, with outputs defined as any value sent to devices. Output determinism offers weaker fidelity guarantees than value or conditional determinism, since it does not make promises about non-visible program state. As a consequence, only output-visible failures such as assertion violations, crashes, core dumps and corrupted data can be reproduced, while failures that produce no distinctive output, such as a deadlock, cannot. It is argued that, despite its limitations, output determinism is enough for debugging purposes for two reasons: (i) it can reproduce all output-visible failures, and (ii) it provides memory-access values that are consistent with the failure, even if they are distinct from those which originally caused the failure.

2.7.3.2 Probabilistic Replay

Debugging a concurrent program without replay support usually requires executing the system many times until the bug is finally reproduced. Most replay systems, on the other hand, reduce the number of attempts to just one, but at the cost of a possibly high recording overhead. A very recent idea in deterministic replay is to explore the space between these two extremes [37, 4]. In other words, the replay system may need a few attempts (e.g. 5-10) to replay the bug, but provides the benefit of reduced recording overhead by only partially tracing the original execution. Since the recorder provides only a partial trace of the original execution, there must be a mechanism that, during the replay phase, is able to somehow reconstruct an equivalent execution to the original in regards to some fidelity guarantee.

Both PRES [37] and ODR [4] can record partial traces of the original execution at different levels: from recording only synchronization operations to tracing all shared-memory accesses. Then, at the beginning of the replay phase, the space of possible executions that fit the partial trace is intelligently explored until the fault manifests itself. The executions performed during this process are fully traced, which enables 100% successful replays after the fault is reproduced for the first time. The partial trace raises a new issue: how do probabilistic replayers know when they have found the right execution? The answer lies with the relaxed fidelity guarantees discussed in Section 2.7.3.1. PRES and ODR offer only conditional and output determinism, respectively. PRES analyzes the state of the replayed execution when visible events occur to check whether the conditions provided by the user are true, while ODR finds the execution that produces a certain output by using a formula solver.

2.7.4 Software-only Solutions

It is no surprise that software-based deterministic replay systems mostly operate at user-level. The exception to the rule is SMP-ReVirt [12] which is implemented as a virtual machine to record at system-level. Those that record at user-level are implemented either through user libraries [27, 41, 15], OS modifications [42] or binary instrumentation, which is performed either statically [20] or dynamically, using a process-level VM [4, 37] or a high-level language VM [8, 16]. It is also noteworthy that none of the software-only systems use data-based algorithms.

2.7.4.1 Synchronization Race Approaches

Systems that record scheduling decisions or synchronization operations reproduce only synchronization races. This is usually done to avoid the overhead imposed by recording all data races, but comes at the cost of the ability to simultaneously replay multi-processor executions and support imperfectly synchronized programs.

Russinovich & Cogswell proposed a replay system with the purpose of reproducing uniprocessor executions of concurrent programs [42]. The target program is instrumented at compile time to maintain a software instruction counter. A modified Mach OS supplies the replay system with the precise points at which context switches occur. They report overheads around 10-15% during the record phase and slightly higher during the replay phase, which are reasonable for a production run. Being implemented as a set of OS modifications makes the solution highly dependent on that particular OS.

liblog [15] is a library-based replay system that, besides recording input non-determinism and enabling replay of distributed applications, records the thread schedule of each process. It faces the interesting challenge of recording the schedule using only a user-level library. Even though this is an easy task for OS- or VM-based replayers, a user-level library is not aware of context switches, making it impossible to monitor or control them. The solution found was to introduce a user-level cooperative scheduler on top of the OS scheduler. Context switches are performed at `libc` API call points, because liblog records the input non-determinism introduced by these (Section 2.6.1).

Instant Replay [27] assumes that the program manipulates shared objects through coarse-grain synchronization operations that implement a CREW (Concurrent Read Exclusive Write) protocol. Because only these operations are reproduced, Instant Replay does not support data races. To generate a total order over object accesses, the synchronization operations are instrumented to increment a version number, associated with each shared object, equivalent to a scalar Lamport clock. Instant Replay traces the version number upon every read operation and the number of read operations between writes. Since no compression method is used to reduce the trace file, it can become very large. During replay, read operations wait until the version number is

correct and write operations wait until the correct number of reads has been performed. Depending on how coarse-grained the synchronization operations are, the performance overhead of the system may differ greatly — fine-grained synchronization will incur high overhead.

DejaVu [8] (and ***Distributed DejaVu*** [24]) records the logical thread schedule of Java programs, consisting of the ordering of synchronization operations (`monitorenter`, `monitorexit`, `wait`, `notify`, `suspend/resume` and `interrupt`) and shared-memory accesses. However, since shared objects are identified through the use of the Java `volatile` keyword, they are always accessed in a safe way and can be modeled as a synchronization operation. The recording mechanism uses scalar Lamport clocks: a global, and a local for each thread. The process yields a total order, making it impossible to replay multi-processor executions without simulating a uniprocessor. The trace file itself is optimized by recording intervals instead of whole sets of timestamps. Implementation-wise, the system makes modifications to the JVM's synchronization routines. The authors succeed in producing small trace files and report recording overheads below 100%.

RecPlay [41] is another system that traces synchronization operations. However, it uses scalar Lamport clocks, not to create a total order, but a partial order, enabling parallelism during replay of multi-processor executions. Their ROLT (reconstruction of Lamport timestamps) method produces a trace, for each thread, consisting of a sequence of timestamps. The trace is compressed by storing only non-deterministic clock increments instead of absolute clock values. Since only synchronization races are recorded, replays are only guaranteed to be correct up to the first data race. RecPlay attempts to minimize this inconvenience by employing data race detection during the first replay execution, using a method based on vector clocks. Performance-wise, the authors report an average of about 20% slowdown during the record phase, making it quite efficient.

JaRec [16] is Java record/replay system implemented using bytecode instrumentation. Loaded classes are passed to an instrumentor through the JVMPI (JVM Profiler Interface). No modifications to the JVM are needed, making the approach completely portable. JaRec traces all synchronization operations available in Java, including synchronized methods and blocks (monitors), `wait` and `notify` calls, and the `start` and `join` methods of threads. A scalar Lamport clock is associated with each thread and synchronization object to create a partial order between synchronization operations. The partial order enables parallelism during replay and, thus, replaying of multi-processor executions. Nonetheless, imperfectly synchronized programs are not supported past the first data race. The generated trace file is similar to that of RecPlay and is compressed using the same techniques.

2.7.4.2 Data Race Approaches

Since replaying solely synchronization races prevents the replay system from simultaneously supporting multi-processor executions and imperfectly synchronized programs, there has been substantial work done with the goal of reproducing all data races. Doing so is generally a high overhead process, making it difficult for software-only systems to have sufficiently low overhead for practical purposes. This fact led to many hardware-assisted solutions which will be surveyed in Section 2.7.5. For now, we survey only software solutions that tackle the problem.

Netzer introduced an algorithm for tracing the optimal set of ordering constraints necessary to reproduce an execution, named Transitive Reduction (TR) [35]. The key feature of the algorithm is that the trace file can be reduced by removing dependencies between shared-memory accesses that are implied by other dependencies. For example, if `T1:1` $\rightarrow$ `T2:4` and `T1:2` $\rightarrow$ `T2:3` are dependencies detected between tasks `T1` and `T2`, then the first dependence need not be stored. This is due to the fact that `T2` waits at instruction `3` for `T1` to reach instruction `2`, which implies that upon reaching instruction `4`, `T1` will have already executed instruction `1`.

To track dependencies, the algorithm uses vector clocks attached to each task and shared memory location. Every time a task accesses a shared-memory location, both clocks are compared and updated, which has the potential for introducing a lot of overhead. Another disadvantage of the algorithm is that vector clocks must have a slot for each task, forcing the target system to follow a static task creation model. This shortcoming can be overcome by employing dynamic vector clocks to handle task creation and termination [26]. The authors did not implement the algorithm, so no practical results about its performance are presented.

SMP-ReVirt [12] is the only software-based approach that replays at system-level. It is implemented using a virtual machine and employs a unique solution for detecting shared-memory accesses through the use of hardware page protections. More specifically, processors have different privileges for each memory page and, when a processor attempts to access a page to which it holds no access privileges, SMP-ReVirt increases them while lowering those of another processor. The log is made up of the points at which privileges change.

This mechanism has one substantial limitation: the granularity of sharing is limited to the size of a page, leading the system to succumb to false positives when fine-grained sharing is used and as the number of processors increases. As a consequence, runtime overhead and trace file sizes scale very poorly. It is reported that the overhead can go up to 10x on machines with a modest number of processors.

LEAP [20] is a replayer for Java programs that produces a partial order by tracing the threads that access each shared variable. As a result, it can reproduce multi-processor executions. Synchronization operations are also traced and the trace file is optimized using intervals to

represent successive accesses by the same thread. LEAP is the only surveyed system that employs a static technique to identify shared variables, instead of having them manually identified through some high-level construct or identifying them dynamically as the program executes. The technique is inherently conservative, which guarantees reproduction of every data race. The system is implemented by instrumenting the bytecode of the target program to generate a record and a replay version. The authors report results showing LEAP is about 10x faster than global order systems, 5x faster than Instant Replay and about 2x faster than JaRec.

ODR [4] introduced the concept of output determinism, already discussed in Section 2.7.3.1. The authors argue that, for debugging purposes, the fidelity of value determinism is helpful, but unnecessary. By lowering its fidelity guarantees, ODR manages to free itself from the burden of recording the outcome of every data race. The trace of an execution consists of three sets: (1) the *input-trace*, which is the result of input non-determinism recording; (2) the *lock-order*, which is a total ordering of lock operations; and (3) the *path-sample*, a set of tuples (t, c, l) where t is a thread, c is an instruction count and l is the program location of the instruction. The detail of each trace may vary and the missing pieces compose the state of possible re-executions. A depth-first search algorithm explores this space using a formula solver to find executions that generate the same output as the original.

The combination of the lower fidelity guarantees and the offline state exploration stage yields a replayer that supports multi-processor executions and imperfectly synchronized programs with low recording overhead, in exchange for a potentially costly or unsuccessful replay.

PRES [37] is a probabilistic approach to replaying multi-processor executions using a software-only recorder. Like ODR, it overcomes the overhead of recording all data races by performing only a partial trace of the original execution. Thus, in order to reconstruct the partially recorded execution, an intelligent offline replayer must search the space of possible outcomes to non-recorded data races. The resulting replay attempts are always consistent with the partial trace of the original, but the user must provide information that enables detection of a correct replay. We classify this guarantee as conditional determinism, because the replayed execution may differ from the original as long as the final state satisfies the user-provided conditions (see Section 2.7.3.1). The space of possible replays can be very large, but a feedback system is proposed that evaluates each failed attempt and generates additional information to guide the next one. This mechanism results in many bugs being successfully replayed after a very modest number of replay attempts.

The authors also explore the space of possible methods for sketching (or partially tracing) the original execution. Starting from a baseline recorder that traced only input, signals and thread scheduling, they experiment with different recorders that incrementally store more information: global order of synchronization operations, system calls, functions, basic blocks and shared-memory operations. As the amount of traced events increases so does the overhead of

recording, the speed of replaying and the faithfulness of the replay. It is reported that PRES, using synchronization or system call global order tracing, significantly lowers the recording overhead of previous approaches. These results come at little cost, as most bugs were still reproduced in under 10 replay attempts. Furthermore, PRES scales well as the number of processors increases.

2.7.5 Hardware-assisted Solutions

The software-only solutions to memory non-determinism replay struggle a lot with the overhead involved in recording data races. As a result, some systems sacrifice flexibility by replaying only uniprocessor executions or assuming data race freedom, while others relax fidelity guarantees and increase replay speed by deriving the outcomes of data races offline. Mainly as an answer to the limitations of software approaches, many researchers have worked on deterministic replay systems that take advantage of hardware support [5, 51, 34, 33, 52, 31, 19, 32]. Such support has the notable side effect of enabling replay of whole systems, unlike the user abstraction level of most software solutions. Nonetheless, a few hardware-assisted systems do record at the level of user libraries.

2.7.5.1 Point-to-point Approaches

Replay systems that track dependencies at the level of individual memory accesses are said to use a point-to-point approach. A timestamp is associated with each memory block and updated on every memory access.

Bacon & Goldstein were the first to propose replaying executions by spying on the cache coherence protocol of directory-based multi-processors using hardware modifications [5]. They piggyback a hardware instruction counter on coherence messages to identify sharing. A subset of the messages is logged to generate a partial order of memory accesses. The replayer has little time overhead, but can generate substantial logs.

FDR [51] can replay the last moments of the execution of a whole target system running on a directory-based multi-processor. It augments cache blocks to contain scalar clocks and modifies the cache coherence protocol to carry and update them. By spying on the protocol's messages, FDR is able to derive the dependencies between memory accesses. It improves upon Bacon & Goldstein's approach substantially by implementing a modified version of Netzer's TR algorithm, in hardware, to compress the trace of memory dependencies. Their version uses scalar clocks instead of vector clocks, which reduces the overhead in exchange for a slightly larger trace size. The authors report that on a 4-processor server with commercial workloads, and given less than 7% of physical memory, FDR can record the last second of execution with less than 2% slowdown.

RTR [52] is an extension of FDR in which Netzer's TR algorithm is improved, resulting in the Regulated Transitive Reduction algorithm. Moreover, FDR's assumption of Sequential Consistency (SC) is relaxed to Total Order Store (TSO). RTR improves on TR by creating artificial dependencies that allow for further trace reduction. TSO is supported by having a hardware component that detects violations of SC and stores the loaded value instead of the usual ordering constraint. The overhead imposed during recording is as negligible as FDR's, but no evaluation of the benefits of RTR compression over Netzer's TR was made.

2.7.5.2 Chunk-based Approaches

A chunk represents a block of instructions that are executed without conflicting with each other in terms of memory accesses. Enforcing the original order of chunk commits is sufficient to replay an execution. Chunk-based approaches can benefit from transitive reduction techniques just like point-to-point approaches.

BugNet [34] supports deterministic replay of user code and shared libraries. The operation of the system revolves around checkpoint intervals, which start with the creation of a new checkpoint consisting only of register state. The value returned by memory load operations that first access a certain memory location in a particular interval is logged, while the values of following loads in that interval are derived by the program itself. A dictionary of common values is used to compress the trace. Given these mechanics, checkpoint intervals represent a set of committed instructions. Each interval has a maximum size and can be prematurely terminated by interrupts and context switches. To aid in debugging, BugNet also uses FDR's point-to-point approach to record shared memory dependencies, but this trace is unnecessary for replay. Due to its data-based recording algorithm and checkpointing mechanism, BugNet can replay individual tasks and start the replay at the beginning of arbitrary checkpoint intervals.

Strata [33] proposes using a logging primitive called stratum. The system maintains an instruction counter for each processor in the machine and, whenever a conflicting memory access is to be performed, a vector with the current counter values for all processors is traced. This is analogous to committing a chunk that started when the previous stratum was recorded, but chunks are named strata regions. Stratums are only recorded if the first memory access in the conflict occurred in the previous strata region. This fact enables a transitive reduction algorithm more efficient than Netzer's TR, because a single stratum can capture multiple dependencies. The trace is reduced even further by not recording WAR (write after read) dependencies, because an offline analysis stage is able to derive the total order between memory accesses without them. Another advantage of Strata is that, unlike point-to-point approaches, the replayer is applicable in both snoop- and directory-based systems.

DeLorean [31] forces processors to execute instructions in chunks, which are invisible to soft-

ware. When a chunk finishes executing, it asks a central module, the Arbiter, whether it can commit. Each chunk is associated with a signature based on Bloom Filters that is used by the Arbiter to immediately make the decision by comparing it with the signatures of already committed chunks. The system can record in three modes: (1) *Order&Size* mode, in which both the size of chunks and their order is non-deterministic; (2) *OrderOnly* mode, in which chunking is deterministic; and (3) *PicoLog* mode, in which everything is deterministic. *PicoLog* mode requires no recording whatsoever, because the Arbiter forces a predefined chunk schedule (e.g. processors round-robin) and size during both the original and replay executions. In *OrderOnly* mode, DeLorean simply traces the order of chunk commits. Finally, in *Order&Size* mode, a log of chunk sizes is also maintained, because chunks can be truncated due to somewhat rare events. The purpose of these three modes of operation is to explore the trade-off between overhead and log size — the latter decreases while the former increases as we move from *Order&Size* to *OrderOnly* and then *PicoLog*.

Capo [32] is a software-hardware hybrid approach that operates at user level (including shared libraries). The key abstraction of the system is the notion of *Replay Sphere* that allows for the separation of duties between software and hardware modules, and enables multiple jobs running in parallel (recording, replaying and standard execution). Each sphere is a group of threads that are recorded and replayed as a whole. Threads belonging to the same process must be part of the same sphere, but the latter may include threads from multiple processes. Tracking threads instead of processors provides more flexibility.

Memory non-determinism is handled by the hardware components of the system, given the overhead that it imposes when done in software. Capo places little constraints on the way the interleaving of memory accesses is recorded, which allows for integration with any of the hardware replayers discussed here. A prototype of Capo was built which used the DeLorean replay mechanism. Despite the additional abstractions taking a toll on both trace sizes (15% and 38% for engineering and system application, respectively) and recording overhead (21% and 41%) when compared with the original DeLorean, they are still modest. Concurrently recording two applications increased the overhead by 6% and 40% for the same classes of applications.

ReRun [19] records how long a thread executes without conflicting with another. The system passively creates atomic episodes analogous to chunks. Lamport clocks are used to establish and trace the interleaving of episodes. Their size is also recorded, as an episode must be terminated when it conflicts with another in terms of memory accesses. The detection of conflicts itself is done by piggybacking on the cache coherence protocol. Enforcing the size and interleaving of episodes is enough to replay the original execution. The main advantage of ReRun is enabling scalable trace sizes on par with other hardware recorders, while requiring only a fraction of the hardware state. While FDR requires augmentations to all cache blocks, ReRun only needs a very small amount of state per processor.

2.8 Distributed Replay

We say a deterministic replay system is distributed if it is capable of cooperating with other instances of itself when replaying distributed programs. There are three major cases to consider: (1) the closed world case, in which all tasks involved in a distributed system are operating under the replayer's supervision, (2) the open world case, in which only one task is supervised by the replayer, and (3) the mixed world case, in which some tasks are supervised and others are not.

While the open world case can and has to be handled by recording network input to the tasks, note that such a replayer does not fit our definition, because it does not cooperate with others. Instead, the tasks are replayed individually by simulating the environment. Nonetheless, this mechanism has been the norm for deterministically replaying distributed systems.

In closed world situations, much space overhead can be avoided by having multiple replayer instances coordinate and regenerate network messages, instead of simulating them. ReVirt [11] proposes this optimization, but does not implement it. Distributed DejaVu [24] handles closed world cases by extending DejaVu's notion of critical event to encompass relevant network-related Java API calls and their paper is very explicit on how to handle stream-based communication, datagram-based communication and connections. liblog [15] uses Lamport clocks to replay communicating peers consistently.

Mixed environments can be handled as closed world cases for cooperating peers and open world cases for the rest. However, we must either know which peers are cooperative a priori, or have a discovery protocol in place that can find them without interfering with the communication protocols used by the distributed system.

2.9 Summary

Support for input or memory non-determinism is the most distinguishing criterion of deterministic replay, because the systems on both sides use very distinct techniques. Input non-determinism can be replayed efficiently with software-only solutions. On the other hand, memory non-determinism is only fully handled in an efficient way by hardware-assisted approaches. Software-based approaches struggle with recording data races and many avoid them altogether, recording solely synchronization races. Thus, they are unable to simultaneously support multiprocessor executions and imperfectly synchronized programs. However, recent probabilistic approaches and supported on static analysis have shown potential for enabling efficient recording while supporting multiprocessors and imperfectly synchronized programs, at the cost of higher replay overhead and lower fidelity guarantees.

3 Ditto

In this chapter we describe the algorithms and data structures that make up Ditto from a mostly design and theoretical point of view, while implementation details are dealt with in Chapter 4. Section 3.1 starts by giving an overview of the different techniques that are used in Ditto. Then, Section 3.2 lists the events that must be monitored and Section 3.3 explains the base algorithms used to record and replay executions. Subsequent sections deal with improvements and optimizations built on top of the base algorithms to enhance Ditto's performance.

3.1 Overview

Ditto is a deterministic replay system capable of reproducing non-deterministic executions of programs executed by the Java Virtual Machine, which combines multiple state-of-art and original techniques.

1. Ditto takes advantage of the semantic differences between load and store memory operations in order to reduce the trace file and enable a high degree of concurrency during the replayed executions. Instant Replay [27] is the only deterministic replayer that uses a similar approach, though it is not targeted at Java applications. More details on how Ditto handles each type of event are given in Section 3.3;

2. Ditto serializes memory accesses at the granularity of individual instance fields, the finest possible granularity, never achieved by previous Java replayers. This means that memory accesses are only explicitly ordered by the replayer if their target is the same instance, static or array field. Section 3.4 provides more details;

3. Ditto uses TLO (Thread Local Objects) static analysis to identify class fields that are guaranteed to never be simultaneously accessed by more than one thread. The use of this type of static analysis is a technique pioneered by LEAP [20]. The information about thread locality allows us to ignore memory accesses to thread local fields, significantly reducing the cost of recording and replaying. Ditto also employs some simple escape analysis to identify arrays which are not shared between threads. This array analysis is performed at compile-time and used because TLO does not provide data on array fields. Details on the static analysis and array escape analysis are provided in Sections 3.6 and 3.7, respectively;

4. Ditto reduces its trace file by employing a new transitive reduction algorithm inspired by Netzer's TR [35]. It also avoids tracing order constraints which can be derived from program order. Both techniques are used to prune redundant order constraints from the trace file and are described in detail in Section 3.5;

5. Ditto operates inside the JVM itself, an approach only previously taken by DejaVu [8], which allows for fine-level monitoring and controlling of the application. We built Ditto on top of JikesRVM (Jikes Research Virtual Machine) [1], of which we make a quick overview in Section 4.1. The hooks used by Ditto to monitor and control an execution inside JikesRVM are described in section 4.2;

6. Ditto employs a number of trace file optimizations which significantly reduce its size. These include storing clocks as increments and using metadata to allow for values of dynamic size, as detailed in Section 3.8.

7. Ditto takes advantage of compiler optimizations to reduce the amount of tracing overhead. Details are given in Section 4.2.

3.2 Events of Interest

In order to reproduce a non-deterministic execution in a multi-processor, the outcomes of all data races must be recorded and replayed. Since data races occur between pairs of shared memory accesses, it follows that a recorder must be able to identify the order in which each such access is performed in relation to the others, so that the same order may be enforced during other executions. The JVM's memory model limits the set of memory access instructions which can read or manipulate shared memory to three groups:

1. Accesses to static fields through the bytecode instructions `getstatic` and `putstatic`;

2. Accesses to object fields through the bytecode instructions `getfield` and `putfield`;

3. Accesses to array fields through a certain pair of instructions depending on the type of the array. For instance, if the array holds references to objects, the application uses the instructions `aaload` and `aastore` to read and write fields, respectively. Similarly, `baload` and `bastore` are used for boolean and byte arrays, `caload` and `castore` for char array, `daload` and `dastore` for double arrays, `faload` and `fastore` for float arrays, `iaload` and `iastore` for integer arrays, `laload` and `lastore` for long integer arrays, and `saload` and `sastore` for short integer arrays.

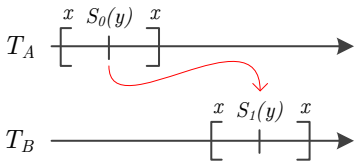


Figure 3.1: Example of a incorrectly recorded execution vulnerable to replay-time deadlock.

In addition to reproducing the order in which shared memory accesses occur, it is mandatory to reproduce the order in which synchronization operations are performed. Though these events have no effect on shared memory, an incorrect ordering can cause the replayer to deadlock. Figure 3.1 illustrates a recorded execution in which not tracing the order of monitor acquisitions can lead to a deadlock during replay. If T_B acquires x's monitor before T_A does, it will proceed to wait for T_A to execute the store operation $S_0(y)$. However, T_A cannot execute that operation because it needs to acquire x to do so and x belongs to T_B – a deadlock. It is, thus, imperative that the order between synchronization operations of any kind be traced and reproduced. In the JVM, synchronization is supported through synchronized methods, synchronized blocks and synchronization methods such as `wait` and `notify`. These are built on top of a single structure: the monitor. Since Ditto operates inside the JVM, it has privileged access to the monitor mechanism which systems based on offline bytecode instrumentation can not benefit from; it is able to trace monitor acquisitions that occur in the context of a `wait` method call, for example. As such, Ditto only needs to track the order between monitor acquisitions to guarantee a correct replay.

Summarizing, Ditto needs to track static field accesses, object field accesses, array field accesses and monitor acquisitions.

3.3 Base Record and Replay Algorithms

The recording and replaying algorithms of Ditto rely on logical clocks (or Lamport clocks) [25]. A logical clock is a mechanism designed to capture chronological and causal relationships, usually consisting of a monotonically increasing software counter. Logical clocks are associated with threads, objects and object fields to identify the order between events of interest. For each event, an order constraint is generated and inserted in the trace file. The constraint can later be used by the replayer to order the event after past events on which its outcome depends.

3.3.1 Recording

In the previous section we asserted that there are two groups of events which our recorder must order, namely the shared memory access operations and the synchronization operations. There need not be, however, order constraints between events belonging to different groups. Indeed load and store operations are ordered in relation to one another, but not in relation to monitor

acquisitions, and vice-versa. In essence, we create two separate streams of order constraints – one orders memory accesses while the other orders monitor acquisitions.

Recording shared memory accesses Taking advantage of the semantic differences between load and store memory accesses is the main design driver on which Ditto's memory access recording mechanism was built upon.

To record the order in which shared memory accesses are performed, Ditto requires state to be associated with threads and fields. Threads are augmented with one logical clock, which we refer to as simply the thread's clock. Fields are extended with one logical clock, the field's store clock, incremented whenever a store operation is performed, and a counter, whose value is the field's load count and represents the number of loads performed on the field since the last store. When a thread performs a monitored shared memory access on a field, the operation is reflected in the values of the thread's clock, the field's load count and possibly in its store clock. The load or store operation and the manipulation of thread and field state must be done atomically in order to guarantee the correct order is traced. Ditto does this by acquiring a monitor associated with the field before the operation and releasing it after the operation has been completed and recorder state has been modified to reflect it. The monitor cannot be part of the application's scope, as its usage would interfere with the application and could lead to deadlocks. We now provide a more detailed explanation of the algorithms that handle each load and store memory access intercepted during execution recording.

When a thread T_i performs a load operation on a field f, it starts by acquiring f's assigned monitor. Then, it traces an order constraint consisting of f's store clock. Such a constraint implies that the load is performed after f's store clock has taken its current value, i.e., the current load operation is to be ordered after the store operation that last modified the value of f. After tracing the constraint, f's load count is incremented by one unit and, if T_i's clock is lower than f's store clock, the former is set to the value of the latter. Following that, the actual load operation is performed and the monitor associated with f is released. Pseudo-code for this process is listed in Algorithm 3.1.

When a thread T_i performs a store operation on a field f, it again starts by acquiring the monitor associated with the field. Then, an order constraint consisting of f's store clock and load count is traced. The constraint implies that the ongoing store operation occurred after f's store clock and load count had been set to their current values, i.e., it is ordered after the previous store operation of f and all load operations performed on the field since then. Afterwards, the values of T_i's clock and f's store clock are updated to the current maximum of the two incremented by one, and the load count of f is reset to zero. Finally, the actual store operation is performed and f's monitor is released. Algorithm 3.2 lists pseudo-code for this process.

Recording synchronization In Section 3.2 we reasoned that reproducing the order in which synchronization operations are performed is mandatory for achieving a correct replay of arbi-

Algorithm 3.1 Recording load memory access operations

```
Parameters: f is the field whose value is being loaded
  method BEFORELOAD(f)
    MONITORENTER(f)
    t ← GETCURRENTTHREAD()
    TRACE(f.storeClock)
    f.loadCount ← f.loadCount + 1
    if f.storeClock > t.clock then
      t.clock ← f.storeClock
    end if
  end method
  method AFTERLOAD(f)
    MONITOREXIT(f)
  end method
```

Algorithm 3.2 Recording store memory access operations

```
Parameters: f is the field where the value is being stored
  method BEFORESTORE(f)
    MONITORENTER(f)
    t ← GETCURRENTTHREAD()
    TRACE(f.storeClock, f.loadCount)
    newClock ← MAX(t.clock, f.storeClock) + 1
    f.storeClock ← newClock
    f.loadCount ← 0
    t.clock ← newClock
  end method
  method AFTERSTORE(f)
    MONITOREXIT(f)
  end method
```

trary executions. Ditto records this order in a similar way it does the order between memory accesses. Even so, there are a few important differences. While shared memory accesses are performed on fields, monitor acquisitions are performed on objects. Furthermore, there is only one kind of monitor acquisition. Thus, recording synchronization requires no state for fields, but it does for objects: one logical clock, which we refer to as the object's synchronization clock, incremented whenever the object's monitor is acquired. Because the order of synchronization is traced independently of the order between shared memory accesses, the thread clock we used before cannot be used for this new purpose. As such, another logical clock is added to the state of each thread, to which we refer as the thread's synchronization clock. A final distinction is that, since a monitor acquisition already provides us with a critical section, there is no need to create our own to assure the atomicity of the monitor acquisition and modification of the thread and object state.

When a thread T_i acquires the monitor of an object o, it starts by tracing an order constraint

comprising the current value of o's synchronization clock. The constraint implies that the monitor acquisition occurred after the synchronization clock had reached its current value, i.e., that it is ordered after the previous acquisition of o's monitor. Then, before executing the application logic protected by the monitor, the synchronization clocks of both o and T_i are updated to the current maximum of the two incremented by one. This process is listed as pseudo-code in Algorithm 3.3.

Algorithm 3.3 Recording monitor acquisition operations

Parameters: o is the object whose monitor is being acquired
method AFTERMONITORENTER(o)
$t \leftarrow$ GETCURRENTTHREAD()
TRACE($o.syncClock$)
$newClock \leftarrow$ MAX($t.syncClock, o.syncClock$) $+ 1$
$o.syncClock \leftarrow newClock$
$t.syncClock \leftarrow newClock$
end method

3.3.2 Consistent Thread Identification

Ditto's traces are composed of individual data streams for each thread. Thus, it is mandatory that we map record-time threads to their replay-time counterparts. Neither the JVM nor the Java API provide us with a thread identification mechanism that remains consistent between executions. Java thread identifiers are attributed sequentially and can be reused once a thread dies. Since threads can race to start other child threads, an identifier with these properties does not meet our requirements.

To achieve the desired effect, we introduce a new identifier, the replay identifier, which is attributed to each thread by Ditto during the thread's start-up process. The identifier is attributed inside a critical section, which is traced by the mechanism that handles Java synchronized blocks. When the replayer reproduces the order of synchronization operations, it will force the identifiers attributed by Ditto to be assigned to the same threads to which they had belonged during the recorded execution. The consistent identifiers allow the replayer to associate trace streams to their respective threads. This is a recursive mechanism, since a trace stream must be associated with the parent thread so that the replayer is able to correctly order the creation of its child threads. The base case is the main thread of the application that is always assigned the identifier zero.

3.3.3 Replaying

The replayer uses the order constraints traced by the recorder to reproduce an execution. The trace is organized as a set of order constraint sequences, each one corresponding to one runtime

thread. The replayer uses consistent replay identifiers to locate the correct constraint sequence for each thread. As a thread runs, it needs to perform events of interest to make progress. The replayer is responsible for using the constraints to guarantee that each event of interest is performed after the events on which its outcome depends. The traces of threads are read sequentially and they contain no metadata that allows the replayer to verify that a certain constraint corresponds to the event currently being executed. As such, the user must guarantee that the program executes the same sequence of events of interest as it did during the record phase. The user is only responsible for providing a non-modified version of the program, as ordering the events is the replayer's responsability. In section 4.7 we talk about ways in which Ditto allows for the target application to be modified to contain debug statements without modifying the events of interest intercepted during its execution.

Replaying shared memory accesses As hinted above, the replayer delays load operations until the values they read during recording are available, while store operations are additionally delayed until all load operations that read the current value of the field at record-time are performed. This is an approach that allows for maximum concurrency during the replay of an execution, since each memory access waits solely for the events that directly affect its outcome.

When a thread T_i performs a load operation on field f, it starts by reading a load order constraint in the thread's trace, from which the target store clock is extracted. T_i then waits until f's store clock matches the target store clock read from the constraint. At this point, the thread is safe to execute the actual load operation. After doing so, T_i increments f's load count and, if the field's store clock is higher than the thread's clock, the latter is updated to its value. Finally, T_i notifies any threads waiting for the field's state to change, allowing them to recheck the conditions required to advance. The pseudo-code for this process is listed in Algorithm 3.4.

Algorithm 3.4 Replaying load memory access operations

```
Parameters: f is the field whose value is being loaded
  method BEFORELOAD(f)
    t ← GETCURRENTTHREAD()
    targetStoreClock ← GETNEXTLOADCONSTRAINT(t)
    while f.storeClock < targetStoreClock do
      WAIT(f)
    end while
  end method
  method AFTERLOAD(f)
    t ← GETCURRENTTHREAD()
    if f.storeClock > t.clock then
      t.clock ← f.storeClock
    end if
    f.loadCount ← f.loadCount + 1
    NOTIFYALL(f)
  end method
```

When a thread T_i performs a store operation on field f, it reads a store order constraint from its trace. It then takes a target store clock and a target load count from the constraint. While f's store clock is lower than the target store clock or f's load count is lower than the target load count, T_i waits. After being notified and making sure the conditions required to proceed are met, T_i performs the store operation. Afterwards, it updates its clock and f's store clock to the current maximum of the two incremented by one and resets f's load count to zero. Lastly, threads waiting on f are notified of the state changes. Algorithm 3.5 lists pseudo-code for this process.

Algorithm 3.5 Replaying store memory access operations

```
Parameters: f is the field where the value is being stored
  method BEFORESTORE(f)
    t ← GETCURRENTTHREAD()
    targetStoreClock, targetLoadCount ← GETNEXTSTORECONSTRAINT(t)
    while f.storeClock < targetStoreClock or f.loadCount < targetLoadCount do
      WAIT(f)
    end while
  end method
  method AFTERSTORE(f)
    t ← GETCURRENTTHREAD()
    newClock ← MAX(t.clock, f.storeClock) + 1
    t.clock ← newClock
    f.storeClock ← newClock
    f.loadCount ← 0
    NOTIFYALL(f)
  end method
```

Replaying synchronization Replaying monitor acquisitions is quite similar to replaying shared memory accesses. When a thread T_i attempts to acquire the monitor of object o, it gets a synchronization order constraint from its trace. The target synchronization clock is taken from that constraint. While o's synchronization clock is lower than the target synchronization clock, T_i waits. Once notified and validated the conditions for proceeding, T_i acquires o's monitor. After that, both T_i's and o's synchronization clocks are updated to the maximum of the two incremented by one and threads waiting on o are notified of the state changes. The process is presented as pseudo-code in Algorithm 3.6.

3.4 Recording Granularity

As far as we have been able to ascertain, Ditto offers the most fine-grained serialization of events of any previous software-based deterministic replayer. We shall use the term recording granularity to refer to the smallest application entity which is considered to be independent of others of its type by the recorder. A global order-based recorder, for example, groups all pro-

Algorithm 3.6 Replaying monitor acquisition operations

```
Parameters: o is the object whose monitor is being acquired
  method BEFOREMONITORENTER(o)
    t ← GETCURRENTTHREAD()
    targetSyncClock ← GETNEXTSYNCCONSTRAINT(t)
    while o.syncClock < targetSyncClock do
      WAIT(o)
    end while
  end method
  method AFTERMONITORENTER(o)
    t ← GETCURRENTTHREAD()
    newClock ← MAX(t.syncClock, o.syncClock) + 1
    t.syncClock ← newClock
    o.syncClock ← newClock
    NOTIFYALL(o)
  end method
```

gram entities together and conservatively considers them to access the same shared resource. In the JVM's context, this means the recorder is incapable of distinguishing between different instances, different fields or even different classes. As a result, all traced events are serialized with one another, removing all concurrent behavior from the replayed executions and introducing record-time overhead due to a global bottleneck.

Some recorders generate partial orders of the events they trace. Nonetheless, all previous systems do so at sub-optimal levels either because of their recording algorithm or due to their granularity being too coarse. LEAP [20] is an example of this latter case: the recorder can only distinguish between memory accesses to different fields; it is unable to identify which instance the field belongs to, treating all fields as if they were static. Thus, all accesses to a field of a class are serialized, even if they are performed on different instances of that class. JaRec, on the other hand, traces at the granularity of individual objects, as it is able to distinguish between different instances of classes, the smallest entity that operations on monitors can manipulate. JaRec's problem lies on its recording algorithm, which creates artificial orderings between events that not only excessively limit concurrency during replay, but can also produce traces that always result in deadlocks.

As explained in Section 3.2, Ditto traces two types of events: shared memory accesses and monitor acquisitions. The former are performed at the level of fields of individual instances, while the latter are performed at instance-level. Ditto's recording granularity is optimal in both cases. This yields a benefit in terms of record-time overhead for applications with fine-grained thread interactions, but it is especially beneficial in terms of trace file size and replay execution time. The downside is that Ditto requires state to be associated with each instance field, adding to the application's memory requirements. To deal with situations in which this

memory consumption could become a problem, Ditto is capable of operating with a instance-level recording granularity, thus consuming only one unit of state per object.

Handling arrays An interesting and seldom mentioned issue is the recording of array field accesses, which is mandatory given that array references can be shared by different threads just like object references. LEAP treats all array field accesses as a single entity, which means all such operations are serialized and no concurrency is allowed between them. Ditto, on the other hand, treats array fields as it does object fields, but with an important twist. Since it is not uncommon for arrays to have many fields, allocating one unit of state for each could lead to unreasonable memory consumption. To avoid this situation, we set a maximum for the number of state units allocated for a single array and map the indexes used in array field accesses to one of the available states. As an example, if the maximum is set to 10 states and an array has a length of 80, the fields are grouped in 10 sets of 8 fields and each set is treated by the recorder as a single entity. This may not be an optimal solution in terms of recording granularity, but it goes towards a compromise with the memory requirements of Ditto. The maximum number of field states for arrays is a user-provided argument. We did not evaluate Ditto's behavior when this value is modified, though it would be an interesting experiment.

3.5 Pruning Redundant Order Constraints

The base algorithm described in Section 3.3 traces one order constraint per memory access, comprised of one value for load operations and two values for store operations. Though the algorithm is correct, the trace file it creates is generally unreasonably large, introducing high disk bandwidth requirements and overhead. The issue is mostly related to scale: memory accesses make up a very significant fraction of the instructions executed by a typical application. Fortunately, many order constraints are redundant, i.e., the order of operations they enforce is already indirectly enforced by other constraints or program order. Redundant constraints can be safely pruned from the trace file without compromising correctness.

Ditto prunes constraints implied by program order, or implied by previously traced constraints. The latter are removed using a technique based on Netzer's transitive reduction, which finds the optimal subset of order constraints that enable correct reproduction of executions [35]. RTR (Regulated Transitive Reduction) further extends Netzer's TR by introducing artificial constraints which allow for the removal of multiple real constraints [52]. Ditto does not directly employ either Netzer's TR or RTR's algorithm, for reasons related to performance degradation, and the need for keeping state that limits application flexibility, such as Netzer's usage of vector clocks, requiring the number of threads to be known a priori. We do, however, use these algorithms as inspiration for our own partial transitive reduction algorithm.

The two mechanisms used by Ditto to prune order constraints are elaborated upon in Sec-

tions 3.5.1 and 3.5.2. Section 3.5.3 then describes how we represent pruned constraints in the trace file and how they are handled during recording and replaying. Finally, we provide a more unified and data structure oriented view of the constraint pruning algorithm of Ditto in Section 3.5.4. In some instances we may talk about the transitive reduction mechanisms only as they apply to the tracing of memory access operations, though Ditto applies them in the synchronization trace as well.

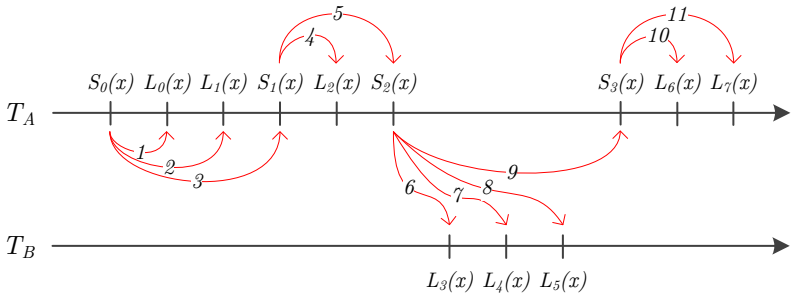
3.5.1 Program Order Pruning

Using program order to prune order constraints proved to be the most effective method of reducing the size of the trace file and, by association, the recording and replaying overheads. One can easily visualize its potential benefits by means of the example in Figure 3.2. Two threads, T_A and T_B, perform load and store operations on a shared object field, x. Figure 3.2(a) illustrates the constraints traced by the base recording algorithm: one for each memory access operation. Notice that many of them order events performed in a sequential manner by the same thread. Constraint 1, for instance, forces the replayer to order the load operation $L_0(x)$ after the store $S_0(x)$, an unnecessary constraint given that the former comes after the latter is T_A's program order and, as a result, can never be executed before it. Using the same reasoning, we can further prune constraints 2, 4, 10 and 11.

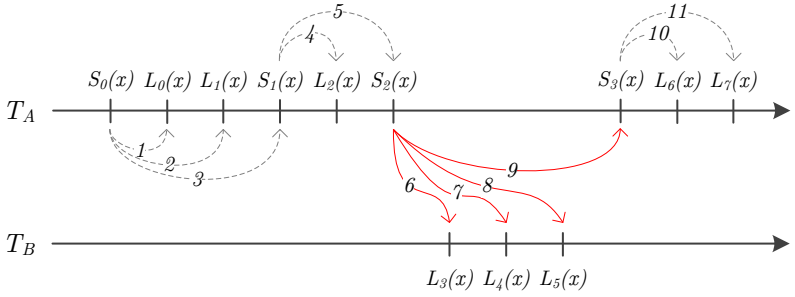
For constraints that order store operations the rules are slightly different, since they depend not only on the last store operation, but also on the load operations performed in-between (for the sake of simplicity the dependency on load operations is not represented in the diagrams of Figure 3.2). These constraints can only be pruned if the last store operation and all load operations in-between were performed by the current thread. As such, constraints 3 and 5 can be safely removed, while constraint 9 cannot, due to the multiple loads performed by T_B since the last store. Figure 3.2(b) illustrates our example trace after pruning program order-implied constraints. Notice how only inter-thread order constraints remain.

3.5.2 Transitive Reduction

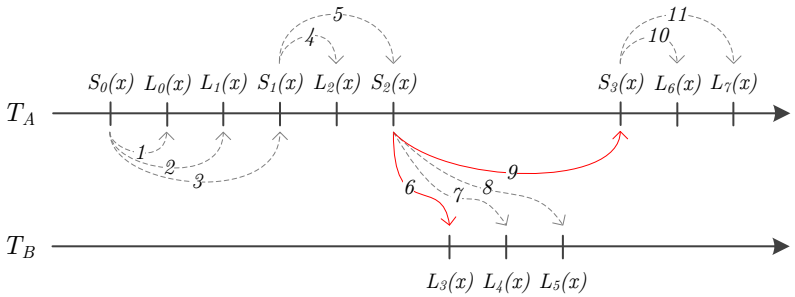
A second pruning mechanism with a high capability to reduce trace file size is transitive reduction, taking advantage of previously traced constraints and program order. Again, we use the example in Figure 3.2 to better visualize the issue and the solution. After removing constraints implied by program order, four constraints still remain, three of which are exactly the same: three load operations ($L_3(x)$, $L_4(x)$ and $L_5(x)$) ordered after the same store operation ($S_2(x)$). Given that the load operations are executed according to program order, notice how forcing the first load to be performed after $S_2(x)$ is enough to guarantee that the two following loads are also subsequent to $S_2(x)$. As such, constraints 7 and 8 are redundant and can be removed,



(a) Order constraints traced by base recording algorithm.



(b) Order constraints traced after pruning those implied by program order.



(c) Order constraints traced after further pruning those implied by previous constraints.

Figure 3.2: Example application of the pruning algorithm for redundant order constraints.

resulting in the final trace file of Figure 3.2(c).

3.5.3 Free Runs

Removing the order constraints associated with certain memory access operations leaves gaps that the base replay algorithm is not equipped to deal with. To handle these gaps, we introduce the concept of free runs, which represent a sequence of one or more memory operations that can be performed freely, without having to synchronize with other threads. When performing a free run of size n, the replayer essentially allows n memory operations to occur without concerning itself with the progress of other threads. Free runs are included in the trace file where the order constraints that generated them would be. Going back to our example, the trace stream of T_B in Figure 3.2(c) would be encoded as a load order constraint followed by a

free run of size 2.

3.5.4 Order Constraint Pruning Algorithm

Though the word prune seems to suggest that redundant constraints are removed from the trace post-execution, this is not the case. Redundant constraints are identified at record-time and are never included in the trace file.

Order constraints implied by program order To enable pruning of program order-implied constraints, we introduce two new variables to the state associated with each application-level field: the replay identifier of the last thread to perform a store operation on the field (*last_thread*) and a flag that is activated whenever a thread which did not perform the last store on the field loads its value (*loaded*).

When a thread T_i loads the value of a field f, three different outcomes are possible: (i) if T_i was the last thread to store a value in f, T_i's current free run is incremented; (ii) if f has not yet been initialized, T_i's current free run is incremented and F's *loaded* flag is set to true; (iii) otherwise, a new load order constraint is traced and f's *loaded* flag is turned on. Similarly, if T_i stores a value in a field f, three outcomes are possible: (i) if T_i was the last thread to store a value in f and that value has never been loaded by other threads (i.e., the *loaded* flag is off), T_i's current free run is incremented; (ii) if f has never been initialized or loaded, T_i's current free run is incremented; (iii) otherwise, a new store order constraint is traced and f's *loaded* flag is turned off.

To perform this optimization on the synchronization trace, we add to each object's state a field with the replay identifier of the last thread to enter the object's monitor. When a thread T_i acquires the monitor of object o, two outcomes are possible: (i) if T_i was the last thread to enter o's monitor or the monitor has never been entered, T_i's current free run is incremented; (ii) otherwise, a new synchronization order constraint is traced.

Partial Transitive Reduction Pruning constraints implied by other constraints is a more complex matter. Besides the already introduced *last_thread* variable for each application field, we add a table to each thread's state. Ditto uses this table to track the most recent inter-thread order constraint between the thread and each other thread it has ever interacted with through shared memory. More specifically, whenever a thread T_i accesses a field f whose *last_thread* is T_j (with $T_i \neq T_j$), f's store clock is inserted in the interaction table at index T_j. In essence, this allows Ditto to declare that any order constraint whose source is the thread T_j with a clock lower than the one in the interaction table is redundant, implied by a previous constraint.

Figure 3.3 illustrates an example execution where constraints are removed using the described algorithm. Notice how constraint 3 would have (correctly) ordered $S_0(z) \rightarrow L_0(z)$ and

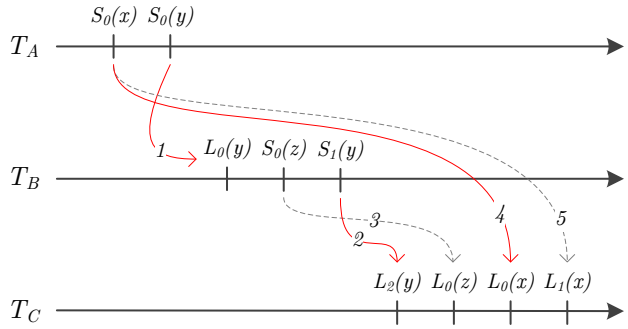


Figure 3.3: Example of partial pruning of order constraints implied by other constraints.

how constraint 2 enforces the order $S_1(y) \rightarrow L_2(y)$. Program order, on the other hand, enforces the orders $S_0(z) \rightarrow S_1(y)$ and $L_2(y) \rightarrow L_0(z)$. As such, following constraint 2 is enough to order $L_0(z)$ after $S_0(z)$, making constraint 3 redundant. Ditto, upon tracing constraint 2, would have registered the interaction with T_B in T_C's interaction table, enabling T_C to realize that constraint 3 is redundant when executing $L_0(z)$. The same mechanism allows Ditto to avoid tracing constraint 5 as well.

The reader may have noticed by now that the set of order constraints traced in the example of Figure 3.3 is not optimal. Indeed, constraint 4 is redundant, as the combination of constraints 1 and 2 would indirectly enforce the order $S_0(x) \rightarrow L_0(x)$. For Ditto to be able to achieve this conclusion, the interaction tables of T_B and T_C would have to be merged when tracing constraint 2. The merge operation proved to be too detrimental to efficiency, as it would have to be performed each time a constraint is traced. Furthermore, the benefit is limited to a maximum of one pruned constraint. In our example, we can see that despite tracing the redundant constraint 4, the similar constraint 5 is pruned, since T_C registered the interaction with T_A upon tracing constraint 4. Given this limited benefit and the high overhead imposed by merging interaction tables, we decided to perform only partial transitive reduction: Ditto is only aware of thread interactions that span a maximum of one traced order constraint. A specialized data structure might be able to enable fast merging of tables and make a full transitive reduction worthwhile, but we have not created such a structure yet.

3.6 Thread Local Objects Static Analysis

The JVM's memory model provides some guarantees with regards to the locality of variables, namely that method-local variables cannot be shared. Accesses to such variables are easily identifiable, as they are performed by a specific family of bytecodes, allowing Ditto to avoid tracing them and hence reduce overhead. In spite of this, there remain a lot of accesses to static and instance fields which are not involved in inter-thread interactions, but about which there is no locality information. As a result, we are forced to conservatively consider them to

be a potential source of non-determinism and trace their outcome. The authors of LEAP have recently employed TLO (Thread Local Objects) static analysis to reduce the set of memory accesses traced during executions of Java programs [20]. We follow suit and use the same kind of analysis in Ditto. In the following paragraphs we explain how TLO analysis works, what its outputs are and how Ditto takes advantage of them to improve performance.

Process and Outputs The output of TLO analysis is a classification of each class field as being either thread-local or thread-shared [18]. Thread-local fields can never be accessed by more than one thread, while thread-shared fields can. This is generally an undecidable problem, which is why the analysis identifies a superset of thread-shared fields. All shared fields are identified as such, but there may be false positives, which will negatively impact performance, but never compromise the correctness of a replay.

TLO requires a set of thread classes T as input. It goes through each thread $t \in T$ independently and performs three classification steps. In the first classification step, fields of t accessed by external methods are classified as thread-shared, while all others are classified as thread-local. Furthermore, the parameters of the methods of t are also classified as thread-shared if the method is invoked externally. For the next classification step, TLO relies on another type of static analysis, IFA (Information Flow Analysis), which approximately computes an answer to whether the value of a memory location x has been derived from the value in another location y. IFA creates an information flow graph and an information flow summary for each method, specifying which of the values manipulated by the method are publicly visible. For each method in t, TLO queries IFA's summary to find where thread-shared values flow into, propagating the thread-shared classification to the destination. This process repeats until the classifications converge. The third and final step involves propagating classifications of fields and parameters of each of t's methods to other variables through call sites.

At this point, we can query TLO to find out whether a certain variable is thread-local or thread-shared inside of a method's context. TLO finds all sources of the value held by the variable and considers it thread-shared if and only if at least one source is thread-shared.

TLO in Ditto We use TLO analysis as it is implemented in the Soot bytecode optimization framework [1] [50]. A stand-alone application performs the static analysis and generates a report file that lists the signatures of all static and instance fields classified as thread-shared. Final and immutable fields after initialization are not included in the report, even if TLO conservatively identifies them as shared. The report file is then fed as optional input to Ditto, which uses the information to avoid intercepting accesses to thread-local fields. Ditto can still function in absence of this report file for two reasons: there is a possibility that the TLO implementation may be unsound in some situations; and the extremely high cost of performing the analysis for even modestly complex applications may make it impossible to generate the report file.

[1] `http://www.sable.mcgill.ca/soot/`

3.7 Array Escape Analysis

TLO static analysis provides very useful information about the locality of class fields, allowing us to avoid the unnecessary overhead involved in monitoring accesses to thread local fields. Unfortunately, TLO does not offer information on array fields, which are harder to track statically. Without taking further measures, we would be required to conservatively monitor all array field accesses. This is a worst-case scenario and there is a lot of potential for improvement.

Ditto uses some very simple compile-time escape analysis on array references to avoid monitoring accesses to fields of arrays declared in a method whose reference never escapes that same method. Aliases for an array used within the method are backtracked to their source. If this source is not a new array instruction, the array is conservatively considered to be thread escaped. This analysis is very simplistic, but it can still avoid some useless overhead. Nonetheless, there is a lot of unexplored potential for this kind of analysis on array references to reduce the overhead associated with tracing array field accesses.

3.8 Trace File

Until now we have merely scratched the surface of the issue which is trace file structure, having asserted that Ditto's trace file is composed of a stream of order constraints per thread. We wish to further discuss this topic, as it affects the performance of Ditto in many ways, from the size of the trace to the way memory and buffers are managed.

Organized by thread Having the trace organized by thread is advantageous for various reasons. The first, and most significant, is that it is easy to intercept the creation and termination of threads. Intercepting these events is crucial for the management of trace buffers, as they must be created when a thread starts and dumped to disk once it terminates. Intercepting the lifetime of other program entities is not as easy. Let us consider the example of LEAP [20], which creates traces organized by field. The lifetime of a field coincides with that of the execution itself, as a field can be accessed at any time. As such, a trace buffer for each field has to be kept across the whole execution, introducing a constant and potentially high memory requirement. In contrast, thread trace buffers are only kept while the thread is alive. An even more complicated situation would be to extend LEAP's approach and organize the trace by instance. In this situation, the lifetime of the buffer is that of the instance, which is, for most of them, significantly shorter than the lifetime of the execution. Intercepting the creation and collection of the instances is, however, a much more complex endeavor than doing the same for threads. The JVM can provide optional support for the `finalize` method of `java.lang.Object` which would be an easy way to intercept object collection, but some kinds of garbage collectors are not necessarily aware of the objects that are no longer alive.

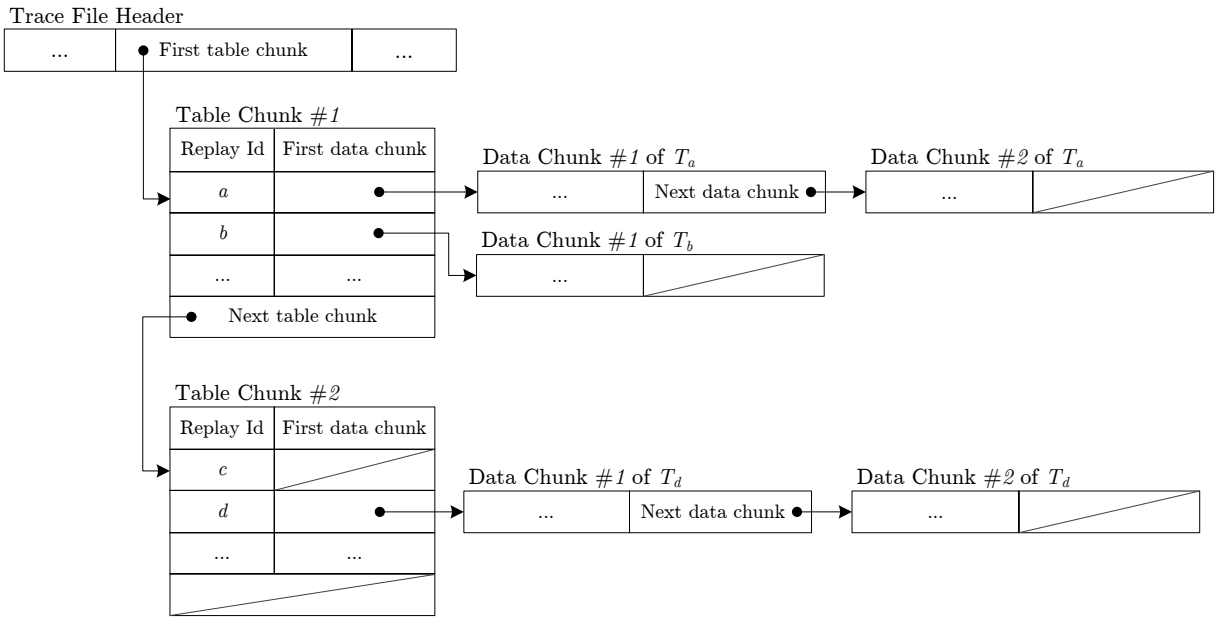


Figure 3.4: Trace file format.

Having a buffer per instance would also be problematic due to memory requirements – either the buffers are small and have to be written to disk very regularly, or we risk introducing huge memory overhead. This brings us to the second advantage of traces organized by thread: the number of simultaneously executing threads is not only much lower, in general, than the number of fields or instances in the execution, but also limited, allowing us to place an upper limit on the amount of memory that can be spent with trace buffers.

3.8.1 Trace File Format

It is common knowledge that sequential I/O is more efficient than random access I/O. It would, thus, be preferable to have the stream for each thread be in its own continuous region in the trace file. Since we use buffers to temporarily hold trace data in memory and dump them once they fill up or the corresponding thread terminates, this optimal scenario is impossible to achieve if we use a single trace file. Hence, we explored the option of using a different file for the stream of each thread, in hopes that such an approach would allow Ditto to take advantage of sequential I/O and improve its performance. The ensuing experimental results proved us wrong, however, as even though each thread was performing sequential I/O, Ditto as a whole was still behaving randomly, with the additional disadvantage of having to constantly switch between the various files.

The best format we developed is illustrated in Figure 3.4 and uses the trace file as a table, indexed by the thread replay identifiers. Indexing the table retrieves the order constraint stream

corresponding to the thread whose identifier was used as a key. The stream itself is stored as a linked list of data chunks which contain the actual trace data. In addition to these chunks, there are also table chunks that map replay thread identifiers to the first data chunk of the corresponding thread. Furthermore, there is another pointer in the file's header that identifies the location of the first table chunk. Table chunks have a limited size and can contain a pointer to other table chunks. Following the table chunk chain we can locate all thread streams. The data chunks that comprise those streams are connected, again, by pointers.

The largest issue with this trace file format is that whenever a memory buffer becomes full and is dumped to disk, a random seek must be performed so that a pointer to its location can be placed in the previous chunk of its stream. Nonetheless, the overhead involved in doing this is significantly lower than the one introduced by the usage of multiple trace files. Sequential I/O could still be performed during replay by rewriting the trace file post-recording. Since we did not experiment with this option, the benefits involved cannot be quantified.

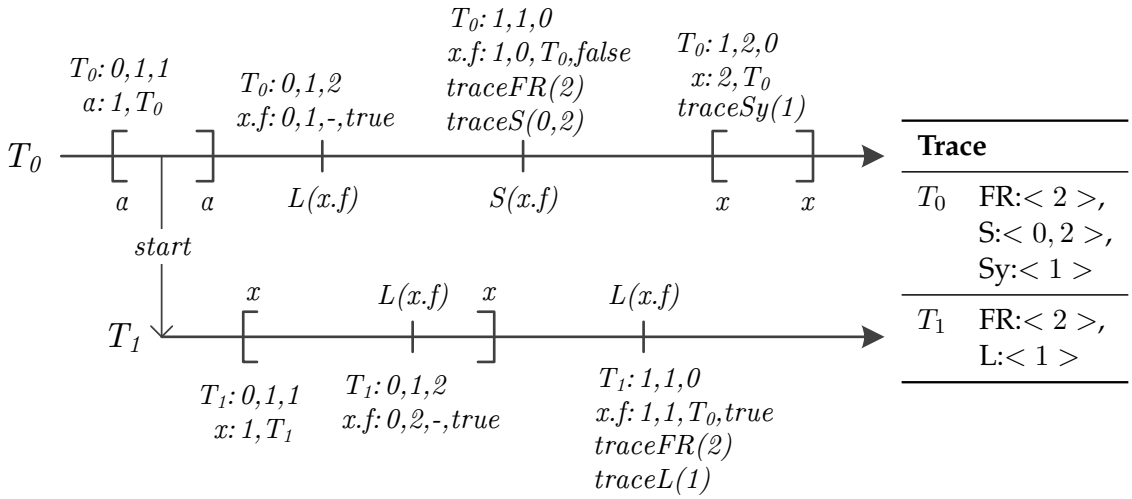
3.8.2 Logical Clock Value Optimization

One disadvantage of using a logical clock-based algorithm is that the clocks increase with every monitored operation (or store operation in Ditto's case) during the execution. As a result, clocks may grow to very high values in long-running applications. In regards to the trace file, this translates into a necessity to use upwards of 8 bytes to store each clock value in order to enable the recording of long executions.

A simple but effective optimization is to store each clock value as an increment in relation to the one that precedes it in the stream, instead of as an absolute value. For the most part, threads will make slow but regular progress, moving its clock forward in small increments. This property combined with the proposed optimization reduces the space used to store the great majority of clock values to 1 or 2 bytes. We delve further into the implementation details of the trace file in Section 4.4.

3.9 Concluding Remarks

Along this chapter we developed the design for Ditto, a deterministic replayer targeted at concurrent programs executing on the JVM and capable of handling the non-determinism of multiprocessor executions. Ditto achieves these goals by using a novel pair of base recording and replaying algorithms, enhanced by multiple optimizations. The recording and replaying algorithms take advantage of the semantic differences between load and store memory accesses, while additionally serializing them at the finest possible granularity to maximize replay-time concurrency and reduce trace data. Even so, they still generate an unreasonable amount of



Legend

$T_i : a, b, c$	Thread T_i with $(store\_clock, sync\_clock, current\_free\_run) = (a, b, c)$
$x : a, b$	Object x with $(sync\_clock, last\_thread) = (a, b)$
$f : a, b, c, d$	Field f with $(store\_clock, load\_count, last\_thread, loaded\_by\_others) = (a, b, c, d)$
$traceFR(a)$	Trace free run of length a
$traceS(a, b)$	Trace store constraint with $(target\_store\_clock, target\_load\_count) = (a, b)$
$traceL(a)$	Trace load constraint with $(target\_store\_clock) = (a)$
$traceSy(a)$	Trace synchronization constraint with $(target\_sync\_clock) = (a)$

Figure 3.5: Recording of a simple execution and resulting trace.

order constraints – one per event. To reduce their number, Ditto uses two techniques: (i) a combination of thread local objects (TLO) static analysis, array escape analysis and compiler optimizations to identify and avoid the tracing of thread-local memory locations; and (ii) a novel take on transitive reduction, performed on-the-fly, which aims at a compromise between the limitations and performance degradation imposed by full transitive reduction and the advantages thereof. We further presented a trace file format designed with I/O delay and memory management in mind. Moreover, a trace file optimization was introduced which reduces the size of logical clock-based traces.

Figure 3.5 illustrates the recording of a short execution by using the fully optimized recording algorithm. Notice (a) the synchronization around thread creation, protected by an object α; (b) the creation of free runs when no inter-thread interactions occur; (c) the distinct semantics of monitoring loads and stores; (d) the separate order for memory accesses and synchronization; (e) the different states corresponding to threads, objects and fields; and (f) the four distinct order constraint types in the trace.

4 Implementation Details

In this chapter we attempt to give the reader an idea about the more interesting and important aspects of Ditto's implementation on top of the Jikes RVM. Section 4.1 gives a brief overview of the RVM's history, thread management and compiler subsystems. Section 4.2 describes a set of modifications to the RVM that support Ditto, focusing on interception of events of interest and state management. Sections 4.3, 4.4 and 4.5 outline aspects relating to the wait/notify mechanism used to control thread progress, the trace file, and memory management, respectively. Finally, Section 4.7 explains how Ditto supports modifications to the application post-recording, while maintaining replay capability.

4.1 The Jikes Research Virtual Machine

The Jikes RVM (Research Virtual Machine) is a high performance implementation of the JVM born under the name Jalapeño at the IBM T.J. Watson Laboratories in 1997 [3], and later renamed and donated to the open-source community[1] in 2001 [2]. Jikes RVM is written almost entirely in a slightly enhanced Java, which provides "magic" methods that the compiler replaces with code implementing low-level semantics, such as pointer arithmetic operations. A novel technique is used to close the Java-in-Java meta-circularity of Jikes RVM. A second JVM is used to bootstrap Jikes, which then compiles itself and writes a boot image to disk. A small C loader application can later be used to load the boot image into memory, after which point Jikes RVM can run on its own. Writing the RVM in Java was partially intended as a way to provide ease of development for researchers, as a main design goal was to create a research platform where novel VM ideas could be explored, tested and evaluated [1]. The project has certainly succeeded in this regard, as evidenced by the more than 200 papers and 40 dissertations that are based on the RVM as of 2012.

Given the properties of Jikes RVM and its success as a research platform, it presented itself to us as the best JVM on which to implement Ditto. In Sections 4.1.1 and 4.1.2 we describe the two subsystems of Jikes most relevant to our purposes – thread management and the compilers.

[1] `http://jikesrvm.org/`

4.1.1 Thread Management

For most of Jikes RVM's life, Java threads were multiplexed on virtual processors implemented as operating system threads. Thread switching was done by preempting threads at specified yield points. The RVM has, however, been recently modified to map each Java thread to a native thread, in an effort to improve compatibility with JNI code, improve performance and simplify the scheduling infrastructure. This change is very relevant to our implementation, since it implies that all scheduling decisions are offloaded to the OS and cannot be traced or controlled from inside the RVM. Java monitors are implemented with recourse to OS locking primitives, as a consequence of using native threads.

4.1.2 Compilers

Jikes RVM does not interpret bytecode; all methods are compiled to machine code on-demand. The VM has two different compilers and, when using an adaptive configuration (enabled by default), manages both to find a good balance between compilation time and execution time reduction. Methods are first compiled by the baseline compiler, which is fast, but produces slow machine code that mimics the stack machine behavior of the JVM very closely. As the application is run, a sampling mechanism identifies frequently executed methods. These are recompiled by the optimizing compiler, which is much slower and complex, but generates machine code of much higher quality and efficiency.

Both compilers generate maps that the garbage collector (GC) uses to identify object reference locations. This was not a problem for the optimizing compiler, as it is built to be easily extensible, but we were unable to insert the somewhat complex instrumentation needed by Ditto in the baseline compiler due to GC map related issues. This is unfortunate, as Jikes RVM is able to replay compilation choices of the adaptive system, hence eliminating the only other problem for a deterministic replayer – a non-deterministic compiler. Given our inability to tame the baseline compiler, Ditto does not use the adaptive system of Jikes RVM and all experiments were performed using solely the optimizing compiler.

The optimizing compiler The goal of the optimizing compiler is to generate the best possible machine code for a certain budget. It must offer significant performance improvements, while preserving correct execution semantics.

Bytecode is not directly translated to machine code. Instead, three intermediate representations (IR) are used to perform different optimizations. Bytecode is first converted to the High Level IR (HIR), which is architecture independent. HIR resembles the bytecode instruction set, but uses a register transfer language instead of the bytecode stack. The resulting code is then passed through a set of optimizations. We consider this point in the compilation process to be

optimal to instrument shared memory accesses, as accesses to instance, static and array fields are still easily identifiable using instruction opcodes, but we already benefit from a large set of optimizations which can reduce the amount of such accesses. Expansion of runtime support like monitor acquisition calls is also performed at this moment. Afterwards, HIR is converted to Low Level IR (LIR), another architecture independent IR resembling the instruction set of a RISC machine. A number of optimizations are applied and the code is then converted to Machine Level IR (MIR), which is architecture specific, with a one-to-one mapping to the target ISA (Instruction Set Architecture). In a final phase, MIR is converted to executable code. The whole compilation process is driven by an optimization plan which can be easily extended by adding new compilation phases. We use this mechanism by inserting an instrumentation phase after the HIR optimizations.

4.2 Hooks, Instrumentation & State

This section explains which hooks and instrumentation are placed by Ditto in Jikes RVM to intercept relevant events. Additionally, it describes where field and thread state is kept and how it is maintained.

4.2.1 Intercepting Events of Interest

VM vs. Application Code A drawback of having a JVM implemented in Java when intercepting events is that the VM uses the same mechanisms to execute as the application. Hence, when intercepting events, such as a thread start-up or a monitor acquisition, we must ascertain whether the event originated in VM or application code. The Jikes RVM does not distinguish between system and user code, though there have been discussions in the community to introduce such a mechanism. Thus, Ditto uses compile-time and runtime tests to decide when an event should be traced. Ditto traces all events that are not generated in ignored packages, i.e., packages explicitly identified as not belonging to the application. These always include the VM's own packages and those of the Java API, but the user is free to specify other packages to be ignored. If TLO static analysis is used (Section 3.6), Ditto further ignores accesses to fields identified as thread-local. Though not implemented, it would be trivial to include the option of overriding the default ignored packages. This could be beneficial to locate bugs thought to be triggered inside the Java API classes, for example.

Thread lifetime Ditto must intercept the creation and termination of threads. Three hooks are introduced in the `start` method of the `RVMThread` class to intercept thread creation: (i) *thread before start*, triggered before the native thread is launched; (ii) *thread before start after id*, triggered after the thread has been assigned a replay identifier; and (iii) *thread after start*, triggered after the native thread has been launched. The first and last hooks are used to enter and exit a critical

section that keeps replay identifiers unique, while the second is used to perform thread state initialization that requires the replay identifier to be known. Thread termination is intercepted by a hook in the `RVMThread.terminate` method, right before the Java runtime is notified of the thread's death. Ditto uses this hook at record-time to complete the trace of the thread by dumping its trace memory buffers to disk.

Synchronization: Monitors Monitor acquisitions can occur in three different contexts. The first are acquisitions performed directly by the application through a synchronized method or block, implemented by having the compiler insert calls to a method inside the VM that performs the lock operation. When such an acquisition is to be traced, Ditto simply replaces the target of that call at compile-time, pointing it to a wrapper method responsible for tracing the event. The second are acquisitions performed by the VM during the execution of synchronization methods, such as `wait` and `suspend`, which Ditto traces using hooks in those same methods. To avoid doing runtime tests, Ditto has the compiler produce code that activates a flag right before each call to a synchronization method that is to be traced. The third and last are acquisitions performed by native code, which Ditto intercepts through a hook in the VM method that implements the corresponding JNI call. Since the VM does not compile native code, we were unable to avoid performing a runtime test for each JNI monitor acquisition that inspects the stack to decide whether or not to trace the event.

Shared Memory Accesses During method compilation, accesses to shared memory are wrapped in two calls, one before and one after, to methods that trace the operation. This instrumentation is performed after HIR optimizations have been executed on the method, allowing Ditto to take advantage of optimizations that remove object, array or static field accesses. Such optimizations include common sub-expression elimination, object/array replacement with scalar variables using escape analysis, among others.

4.2.2 Thread, Object and Field State

The thread state required by Ditto is kept in the `RVMThread` objects and initialized when the corresponding thread is created, using the thread lifetime hooks. Keeping object and field state is a more challenging issue. We use a section of the object header reserved by Jikes RVM for extensions to store a reference to a state object, which includes the instance's state and that of its fields. The reference on its own introduces a memory overhead of one word per instance. The object scanning mechanism of the GC was modified so that this new reference in the header is checked and the state marked as a live object. This approach allows us to keep state for as long as the corresponding object is alive, but no longer than that. State creation and initialization is performed on-demand, in order to avoid unnecessary memory consumption.

4.2.3 Handling Deadlocks

Ditto requires the trace file to be complete once the execution is over in order to replay it. When the execution ends up in a deadlock state, the JVM will never exit and, as a result, the trace file may not be finished, as memory buffers are not dumped to disk. This problem is solved by adding a signal handler to JikesRVM which intercepts `SIGUSR1` signals and instructs the replay system to finish the trace file. The user is responsible for sending the signal to the JVM before killing its process if a deadlock is thought to have been reached.

4.3 Wait and Notify Mechanism

During execution replay, threads are constantly testing whether their next event of interest can be performed. When the conditions for advancement do not hold true, the thread is forced to wait. Hence, the mechanism through which they wait and are later notified of state changes is a very significant implementation detail. Algorithms 3.4, 3.5 and 3.6 were presented in Section 3.3 as the ones used to handle each one of the three types of events of interest: loads, stores and monitor acquisitions. In these, threads wait on the field or object whose current state is keeping them from proceeding and are notified every time the same state is modified. Once notified, threads have to re-evaluate the conditions they require to advance and, if these remain unmet, go back to waiting for further state changes. This is the simplest approach, but notifying all waiting threads every time a state is modified leads to a bottleneck when they compete to reacquire the field's (or object's) associated monitor. In practice, Ditto uses a much more refined solution.

The replay-time states of fields and objects are augmented with a table, which we refer to as the wait table. It is indexed using keys of three types: (i) load keys, used by load operations to wait for a specific store clock; (ii) store keys, used by store operations to wait for a specific combination of store clock and load count; and (iii) synchronization keys, used by monitor acquisitions to wait for a specific synchronization clock. Let us use an example to demonstrate the mechanism. A thread T_i attempts to perform a load operation on a field f, but finds that f's store clock is lower than its target store clock, i.e., the value to be loaded is yet to be stored in the field. T_i then creates a load key using the target store clock tsc and adds a new entry to the wait table using the key as both index and value. T_i then invokes `wait` on the key. By having the key classes implement custom `hashcode` and `equals` methods whose return values are based on the contents of the key, other threads are able to create their own keys and index the wait table to find threads waiting for a very specific state. Going back to our example, a thread T_j might come along and perform a store operation on f which updates its store clock to tsc. T_j creates a load key with tsc as the store clock and a store key with tsc as the store clock and 0 as the load count. It indexes the wait table using both keys and finds that there is a value in

Metadata bits	Value type
00 01	Positive clock increments and load counts
10	Free runs
11	Negative clock increments

Table 4.1: Distribution of metadata bits per value type.

it for the load key. This implies that at least one thread is waiting for the store clock to become tsc in order to perform a load operation and, indeed, T_i is that thread. T_j invokes `notifyAll` on the key retrieved from the table, allowing T_i to verify that f's store clock is now at the right value and proceed with the load.

An additional technique used to reduce the amount of overhead imposed by repeated calls to `wait` and `notifyAll` is to have threads perform a processor yield the first time the requirements for advancement test negative. Only after regaining the processor and again negatively evaluating the requirements for advancement do threads actually perform a `wait` invocation. Given that memory accesses are performed quite regularly, there is a high chance that the conditions for advancement will be met until the thread regains the processor after having yielded it. This mechanism could probably still be improved upon by using more complex polling techniques usually employed by I/O devices.

4.4 Trace File

4.4.1 Metadata

A critical issue that affects the trace file size is the way different kinds of values are encoded. As described in Section 3.8, the trace is organized as a set of order constraint streams, one for each thread. The streams themselves are easily identified using the table chunks and intra-file pointers. Structuring the streams is another challenge. Given the way Ditto records executions (Section 3.3) and the logical clock value optimization (Section 3.8.2), we need ways to encode the following types of values: (i) clock increment values; (ii) free run values; and (iii) load count values. Furthermore, as the clock value optimization does not provide us with an upper limit on how large a clock increment can be, the number of bytes used to store each value is flexible, requiring the introduction of a mechanism to encode that information as well.

Encoding the three different kinds of values is achieved by using the two most significant bits of each value as identification metadata. Table 4.1 shows how these metadata bits are used. Since two bits can encode four states but we only require three, the two states corresponding

to a most significant bit of zero are grouped together to represent positive clock increments and load counts. The choice to take space from the negative increments to use as free runs is not arbitrary, as we expect most clock increments to be positive due to the inherent monotonic increase of logical clocks.

Representing the size of values is a different issue. For one, at least two bits are needed to encode the possible sizes for values that we consider – 1, 2, 4 or 8 bytes. Storing this information as additional metadata for each value would total four metadata bits, leading to a crippling reduction of the value range that each entry can represent. Furthermore, it is usual for consecutive values to have equal size, leading to a lot of redundant information if the size is declared for each individual entry. Taking these observations into consideration, we introduce meta-values to the streams which encode the size of values that follow them and how many they are. These meta-values take up two bytes, but our experiments show that their number is insignificant in comparison to the total number of values stored, allowing for very significant trace file size reductions.

4.4.2 Writer Thread

Having reasoned in favor of a single-file trace file in Section 3.8, we set off to optimize I/O operations and minimize the amount of time threads spend waiting for them to complete. To achieve this goal, a daemon thread was added to Ditto, whose sole purpose is to write trace buffers to disk. Additionally, application threads are given two trace memory buffers. When one becomes full, it is given to the writer thread, while the application thread is free to continue executing and filling the second buffer. In most cases, the former buffer is written to disk faster than the latter is filled; a situation in which application threads do not have to wait for I/O at all.

4.5 Memory Management

A small memory footprint was a significant design driver for our implementation of Ditto. In this section we discuss the practical aspects of memory management; the theoretical memory requirements have been progressively stated throughout Chapter 3. Our highest priority was to have Ditto's memory consumption be constant in regards to the execution time, i.e., memory usage should only increase with new threads or objects. We achieve this by recycling the objects used by Ditto's internal mechanisms, avoiding the creation of instances unless the number of threads or objects has grown. Not only does this reduce the memory footprint of Ditto, but it also reduces overhead associated with object creation and garbage collection processes.

Another key issue regarding memory is the management of trace buffers. Ditto creates

trace streams for each thread, which means buffers are associated with threads. The first benefit of this is that there is a low upper limit on how much memory can be spent with trace buffers, since the maximum number of simultaneously running threads is small. A second benefit is that it is easy to place hooks at key moments of a thread's lifetime, allowing us to easily initialize thread state and dump the trace buffer when a thread terminates. This is not as easy to do when buffers are associated with class fields, as they are in LEAP [20], or if we were to associate trace streams with each instance field. The size of trace buffers can also be subject to optimization. On the one hand, we want them to be large enough that it takes the application more time to fill one than it takes for the writer thread to dump the previous one to disk. On the other hand, buffers size should be minimized due to the fact that the writer thread mechanism uses two buffers per thread. The size of each buffer can be specified through a command line argument.

4.6 Consistent Thread Identifiers

The consistent thread identifiers assigned by Ditto to application threads are attributed in a sequential manner. For applications that spawn a high amount of threads the values of identifiers may grow large. As such, we store identifiers using 4 to 8 bytes. For Ditto, this is not very problematic, as thread identifiers are only stored once in the trace file, inside one of the chunk tables. For recorders whose trace is composed of thread identifiers, such as DejaVu or LEAP, the size of identifiers is more important. Though our implementation does not do so, it would be possible to recycle replay identifiers in the same way the Java API recycles identifiers of dead threads. This optimization would place an upper limit on the possible value of an identifier, namely the maximum number of threads that the JVM can run concurrently. The identifiers could then be stored using significantly less space.

4.7 Modifying the Original Application

Debugging methodologies like cyclic debugging are based on performing multiple iterations of the faulty program in hopes of making observations that would allow the programmer to reduce the set of possible causes for the fault at hand. Many times, the process of information gathering relies on the introduction of new statements into the program which allow the programmer to take a glimpse of program state which was not observable before. At first glance, this way of debugging seems to be incompatible with Ditto's requirement that the stream of events of interest not change between recording and replaying, though it is not.

The metadata provided by the JVM environment about program structure enables us to avoid monitoring events of interest generated in certain parts of the application. There are

many ways in which a user may modify the target application and instruct Ditto to ignore events of interest that originate from those modifications. Events generated within methods can be ignored either by specifying method signatures as a command line parameter or by adding an annotation to the methods. Since method invocations are not monitored, this technique allows for arbitrary execution of code without changing the stream of events handled by Ditto. These two techniques can also be used to ignore events whose target is a specific class field, instead of events generated during the execution of specific methods. Extending the scope of command line parameters and annotations from method and field level to class or package level is also a possibility. All events associated with fields and generated by methods of the specified class or classes of the specified package are ignored by Ditto.

5 Evaluation

This chapter describes the evaluation process to which Ditto was subjected and reports on its results.

5.1 Evaluation Methodology

We evaluate Ditto by assessing its ability to correctly replay recorded executions and by measuring its performance in terms of record overhead, replay overhead and trace file size. Though comparing performance results with those of previous approaches is very beneficial to understand how well Ditto performs, doing so based solely on their corresponding publications would be nearly impossible. To bypass this problem, we implemented three previous published solutions to replaying Java applications inside JikesRVM, using the same runtime hooks and other facilities used by Ditto itself. These are: (a) DejaVu [8], a global-order replayer; (b) JaRec [16], a partial-order, Lamport clock-based replayer; and (c) LEAP [20], a partial-order, access vector-based replayer. DejaVu and JaRec were originally designed to replay synchronization races in uni-processor or data-race-free settings. Thus, we extend their approaches to trace every shared-memory access. We followed the respective publications as closely as possible and, in some aspects, improvements were even introduced. Indeed, DejaVu and JaRec are allowed to take advantage of the TLO static analysis just like Ditto and LEAP. Moreover, our implementation of LEAP actually follows its publication better than the publicly available version, which neglects to regularly dump the trace to disk, compress simultaneous accesses to a field by the same thread, or even wrapping memory accesses and their corresponding trace operation in a critical section.

We start, in Section 5.2, by assessing the correctness of the replayed executions using both a highly non-deterministic microbenchmark and a number of applications from the IBM Concurrency Testing Repository [1] which, between themselves, exhibit various concurrent bug patterns [13, 14]. This is followed, in Section 5.3, by a thorough comparison between Ditto's runtime performance characteristics and those of the other implemented approaches. The results are gathered by performing a microbenchmark and running select applications from the Java Grande and DaCapo benchmarks. Even though the DaCapo benchmark consists of real-world appli-

[1] `https://qp.research.ibm.com/concurrency_testing`

cations, we planned to further evaluate the overhead imposed by Ditto on actual programs publicly available on the Internet. Unfortunately, the combined limitations of JikesRVM, the GNU Classpath, and our own system, made it impossible to run most of these on top of Ditto.

Experiments were conducted on a 8-core 3.40Ghz Intel i7 machine with 12GB of primary memory and running 64-bit Linux 3.2.0. Results for variable number of processors were obtained using the same machine by limiting the JikesRVM process to a subset of cores.

5.2 Replay Correctness

Even though the correctness of replayed executions is of the utmost importance, deciding whether an execution is a correct replay of another depends on how faithfully we want to reproduce it. As such, in section 5.2.1, we start by defining what replay correctness means in the context of our work. Only then do we assess whether Ditto can faithfully replay executions by attempting to reproduce (1) a highly erratic, non-deterministic microbenchmark (Section 5.2.2); and (2) the bugs exhibited by a collection of applications from the IBM Concurrency Testing Repository [13] (Section 5.2.3).

5.2.1 Defining Replay Correctness

With the exception of some hardware-based solutions [51, 31], most deterministic replayers do not attempt to reproduce executions to the highest possible level of fidelity. In fact, probabilistic approaches, by their very definition, have to relax their fidelity guarantees. For instance, a replay performed by ODR [4] is only guaranteed to generate the same output as the original, while one performed by PRES [37] is considered faithful if the recorded fault reoccurs, even if the execution deviates from the original in other ways. Non-probabilistic approaches, on the other hand, tend to allow replayed executions to deviate from the originals only during those periods in which tasks do not modify shared program state. This means that, during replay, the shared state is forced through the transitions it experienced during recording, while the collection of task local states is allowed to go through different transitions.

Ditto was developed with the latter fidelity guarantee in mind. As a result, a replay execution is considered correct if and only if the shared program state goes through the same transitions as it did during recording.

5.2.2 Microbenchmark

The main design driver for our microbenchmark is to produce a highly erratic and non-deterministic output, so that we can confirm the correctness of replay with a high degree of

Application	Bug Pattern
Account	Wrong Lock or No Lock
Airline Tickets	Non-atomic Operation
Booking	Non-atomic Operation
Bounded Buffer	`notify` instead of `notifyAll`, Deadlock
Bubble Sort	Non-atomic Operation, Orphaned Thread
Linked List	Non-atomic Operation
Liveness	Dormancy, Lost `notify`
Loader	`sleep` Interleaving
Lottery	Non-atomic Operation, Wrong Lock or No Lock
Manager	Non-atomic Operation
Merge Sort	Non-atomic Operation
Pingpong	Wrong Lock or No Lock
Piper	Condition for `wait`, Deadlock
Producer Consumer	Orphaned Thread
Shop	`sleep` Interleaving, Double-checked Locking

Table 5.1: Summary of evaluated applications from the IBM Concurrency Testing Repository

assurance. We accomplish this by having threads randomly increment multiple shared counters without any kind of synchronization and using the final counter values as the output. Various parameters can be specified, such as the number of threads, number of iterations and number of shared counters. The microbenchmark's source code is available online[2].

After a few iterations, the final counter values are completely unpredictable due to the non-atomic nature of the increments. Naively re-executing the benchmark in hopes of getting the same output will prove unsuccessful virtually every time. On the contrary, by using Ditto, one is able to reproduce the final counter values every single time, even when stressing the system by providing high parameter values for number of threads and iterations.

5.2.3 IBM Concurrency Testing Repository

The IBM Concurrency Testing Repository contains a number of small applications that exhibit various concurrent bug patterns while performing some practical task. The next few paragraphs describe the functionality and bugs of a representative subset of applications in the benchmark suite we used to evaluate Ditto. Table 5.1 summarizes the whole suite by listing every evaluated application along with the concurrent bug pattern they exhibit, which are formally defined in [14].

Account *Functionality.* Simulates a bank; each thread manages accounts by depositing, withdrawing and transferring money. *Bug.* Methods are synchronized, but when transferring from

[2] `https://github.com/gdeOo/ditto`

account A to account B, only A's monitor is acquired. An example of the Wrong Lock bug pattern.

Airline Tickets *Functionality.* Threads act as agents that sell airline tickets. Like in the real world, the amount of tickets distributed between agents is larger than the total number of actual seats. Thus, before selling a ticket, the agent must make sure that there are still seats available by communicating with a central point. *Bug.* Due to wrong atomicity assumptions, the last ticket can end up being sold to multiple clients by different agents. An example of the Non-atomic Operation pattern.

Bounded Buffer *Functionality.* A buffer is shared between multiple consumer and producer threads, which may have to wait in a queue if the buffer is empty or full, respectively. *Bug.* Consumers and producers share the same wait queue, but `notify` is used to wake them. If, for example, a producer fills the buffer's last spot and notifies another producer by chance, the application deadlocks. An example of the `notify` instead of `notifyAll` bug pattern.

Liveness *Functionality.* A server deals with a limited number of concurrent client requests. Additional clients wait in a queue and are notified once server slots are available. *Bug.* The server may finish dealing with all its clients and issue a `notify` before a to-be-waiting client actually calls `wait`, but after it has decided to do so. An example of the Lost `notify` and Dormancy bug patterns.

Loader *Functionality.* Performs bubble sort on a list of integers, with each iteration being performed by a different thread. *Bug.* The initialization of child threads issues a `sleep` which is believed to always produce the expected interleaving: the parent completes its task before the child starts its own. An example of the `sleep` Interleaving bug pattern.

Piper *Functionality.* Passengers, represented by threads, try to enter a plane. If the plane is full, they wait in a queue until a `notifyAll` awakens them. *Bug.* The `wait` call is wrapped in an `if` statement instead of a `while` statement. As such, if there are more passengers in the queue than there are seats, some of the awaken passengers won't have an available seat, but they won't go back to the queue either. An example of the Condition for `wait` bug pattern.

Producer Consumer *Functionality.* A master threads assigns work to slave threads through a bounded buffer. *Bug.* A local slave bug can make the queue inconsistent and crash the master, leaving all the other slaves waiting forever and deadlocking the system. An example of the Orphaned Thread bug pattern.

Results

Ditto is capable of correctly reproducing each and every of the bugs exhibited by the applications in Table 5.1. Although these do not constitute the whole benchmark suite, they do amount

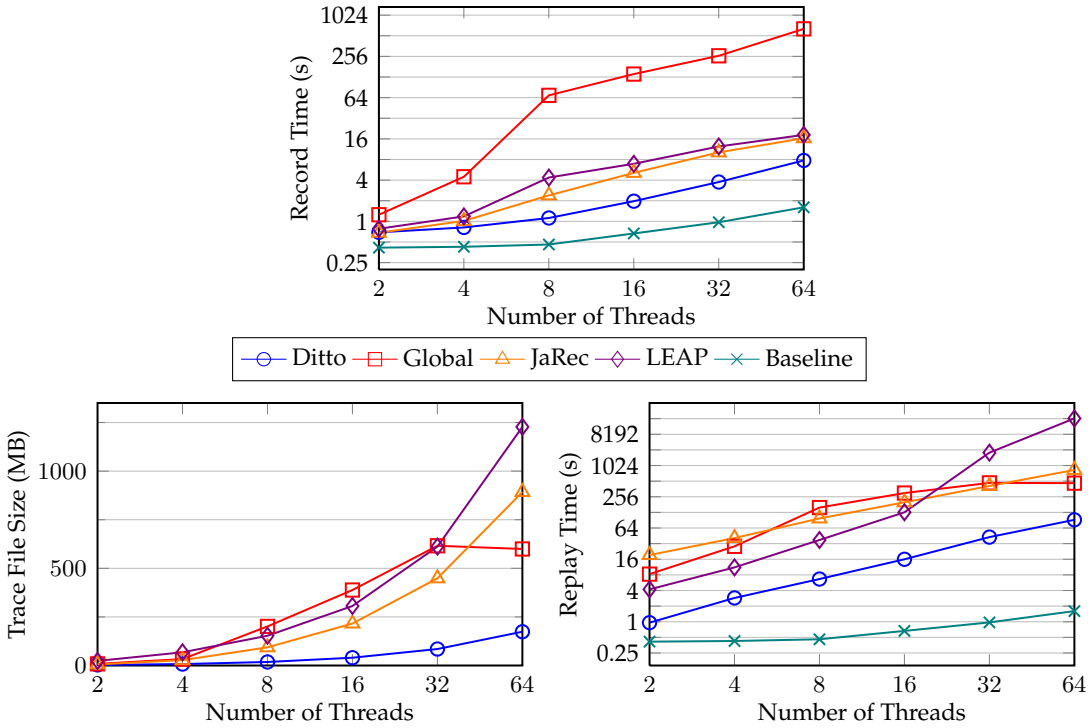


Figure 5.1: Microbenchmark's performance results for Ditto and previous replayers as a function of the number of threads.

to the subset of the suite which does not rely on input for bugs to manifest and is compatible with JikesRVM, the vast majority. Since any program state transitions triggered by specific input, such as the results of a random number generator, are outside the scope of Ditto, we can safely assert that it successfully replayed every bug in the IBM Concurrency Testing Repository which is triggered by memory non-determinism.

5.3 Performance Results

Having asserted Ditto's capability to correctly replay many kinds of concurrent bug patterns, we set off to evaluate its performance by measuring record overhead, trace file size and replay overhead. To put the resulting experimental results in perspective, we use the same performance indicators to evaluate three previous approaches that represent the state-of-the-art deterministic replay techniques for Java programs – DejaVu, JaRec and LEAP, as detailed in Section 5.1. To start with, we perform a flexible microbenchmark that provides us with a very clear picture of how Ditto performs in relation to the other replayers across multiple axes. The ensuing results are presented in Section 5.3.1. In addition, we measure recording overhead and trace file size for executions of benchmark applications from the Java Grande and DaCapo suites, the results of which are reported in Sections 5.3.2 and 5.3.3, respectively.

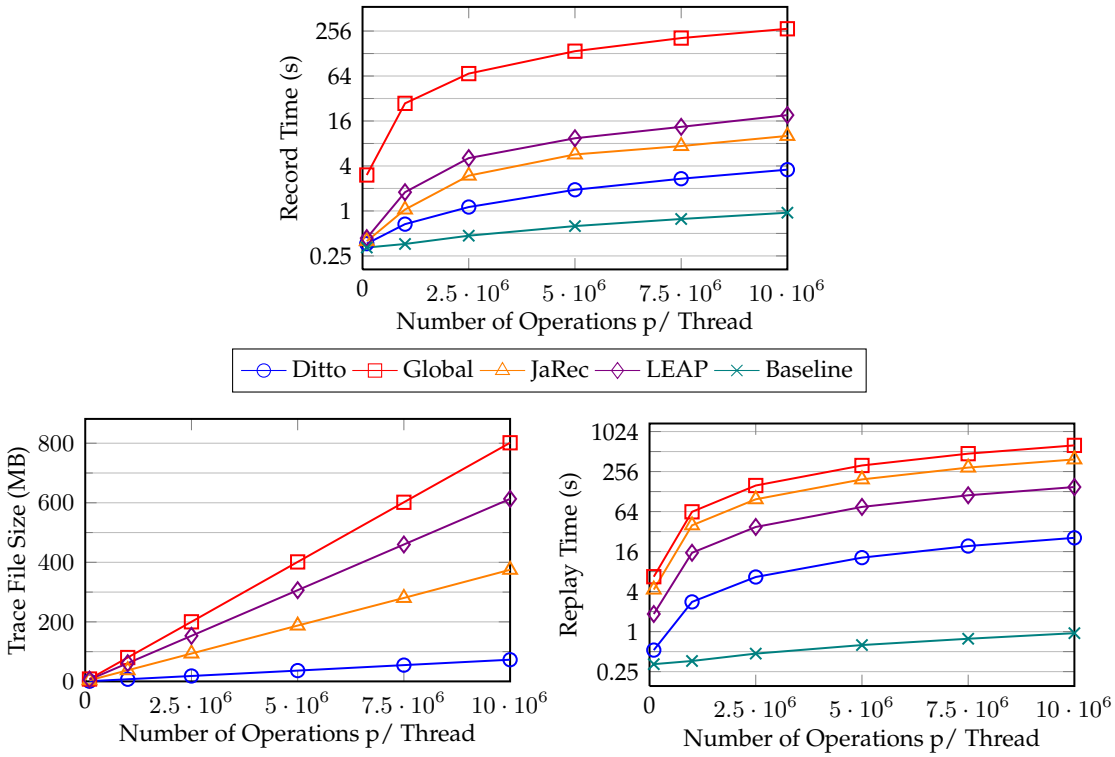


Figure 5.2: Microbenchmark's performance results for Ditto and previous replayers as a function of the number of memory access operations performed by each thread.

5.3.1 Microbenchmark

For this evaluation step we return to the microbenchmark used to assess replay correctness in Section 5.2.2. The benchmark achieves very high non-determinism by having threads concurrently access random fields of random counter objects in a non-synchronized manner. This time, however, we use five input parameters to modify the benchmark's properties in a controlled manner, namely (i) number of threads; (ii) number of memory access operations performed by each thread; (iii) load:store ratio; (iv) number of fields per shared object; and (v) number of shared objects. This allows us to analyze how different application properties are reflected in the performance of each replayer.

In the next few sections we report on how each replayer performs along these axes using a collection of plots depicting their record execution time, trace file size and replay execution time. Note that execution time plots use logarithmic scales due to order-of-magnitude-sized differences between replayers. Finally, we evaluate how performance scales with the number of processors by limiting the VM's process to a subset of available CPUs.

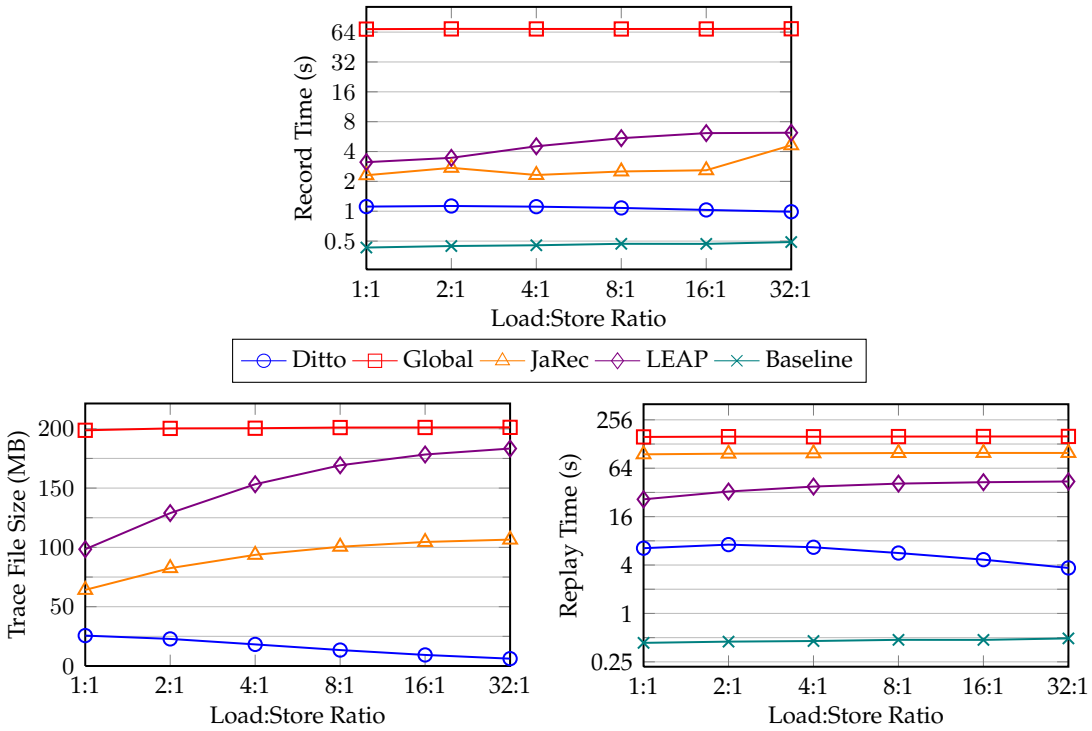


Figure 5.3: Microbenchmark's performance results for Ditto and previous replayers as a function of the load:store ratio.

5.3.1.1 Effect of the Number of Threads

The performance results for each recorder in function of the number of threads are depicted in Figure 5.1. Record execution times grow linearly with the number of threads, but Ditto outperforms its competitors in terms of absolute values by at least one order of magnitude. A similar trend is observed in replay execution time measurements, though Ditto's advantage increases to two orders of magnitude and LEAP behaves in an exponential manner. As for trace file sizes, Ditto stays below 200Mb while no other replayer comes under 500Mb, with the maximum being achieved by LEAP at around 1.5Gb.

5.3.1.2 Effect of the Number of Memory Access Operations

All algorithms increase the record overhead, replay overhead and trace file size linearly with the number of operations performed by each thread. We believe this result can be attributed to two factors: (i) none of the algorithms keeps state whose complexity increases over time, and (ii) our conscious effort during implementation to keep memory usage constant by avoiding unnecessary garbage collections and object creations. Though all replayers react in the same way, Ditto is superior in terms of absolute values, having better efficiency in the record phase by at least an order of magnitude, in the replay phase by two orders of magnitude, and a trace file size substantially smaller than the competing approaches. Figure 5.2 presents the

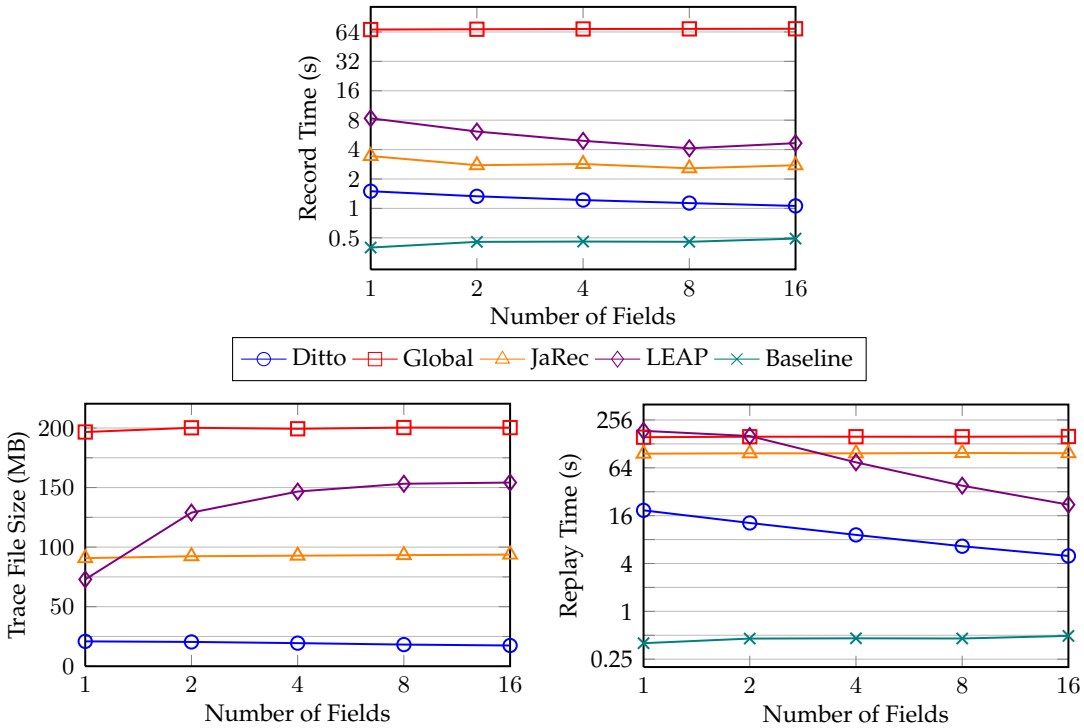


Figure 5.4: Microbenchmark's performance results for Ditto and previous replayers as a function of the number of fields of shared objects.

measurements supporting these observations.

5.3.1.3 Effect of the Load:Store Ratio

Ditto is the only evaluated replayer that claims to take advantage of the different semantics of load and store memory access operations. As such, we expect it to be the only replayer to positively react in the presence of a higher load:store ratio. The experimental results shown in Figure 5.3 are consistent with these expectations. As the ratio increases, Ditto's record execution time, trace file size and replay execution time consistently decrease, with the most significant improvement being in terms of trace file size. The remaining replayers react to a higher ratio either in a negative or neutral way. Furthermore, Ditto remains superior in terms of absolute values for execution times by multiple orders of magnitude.

5.3.1.4 Effect of the Number of Fields of Shared Objects

The amount of fields of shared objects is relevant for replayers that trace at such granularity. Of the four evaluated replayers, only LEAP and Ditto have this property. Thus, we expect both of these to improve their performance as the number of fields increases. Figure 5.4 shows the measurements of our experiments, which are consistent with these expectations. In terms of execution times, both LEAP and Ditto take advantage of a higher number of fields, though,

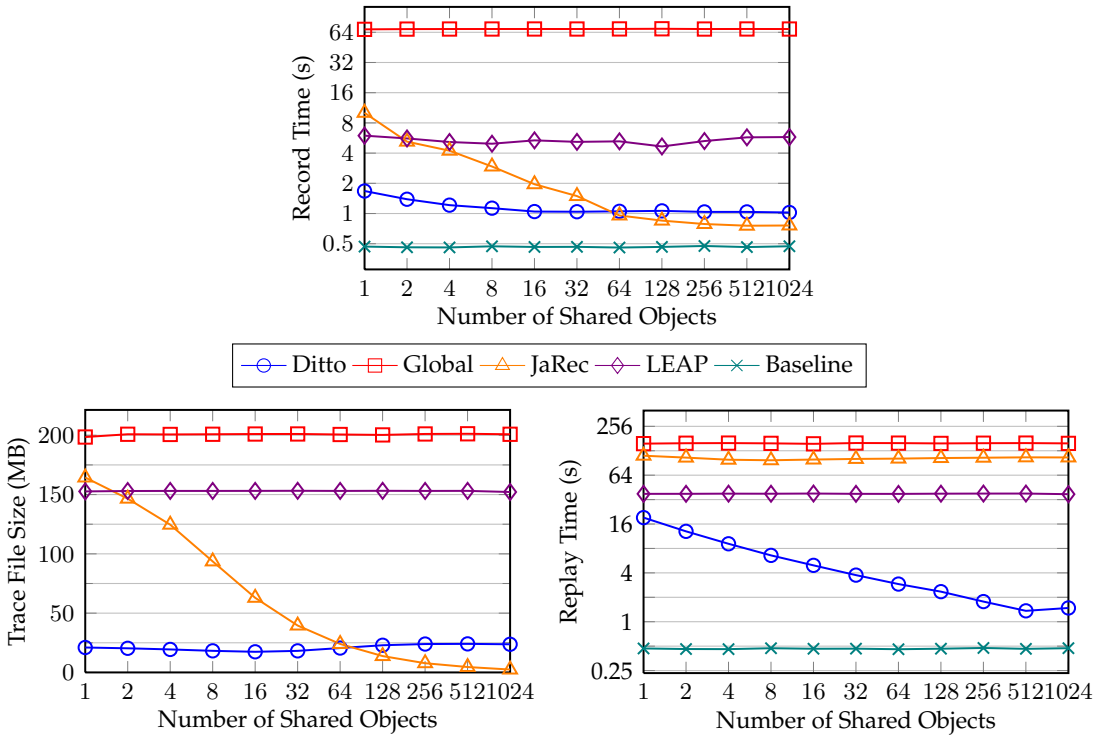


Figure 5.5: Microbenchmark's performance results for Ditto and previous replayers as a function of the number of shared objects.

surprisingly, LEAP increases its trace file size, a behavior we believe is a direct result of the access vector approach to tracing. Ditto's absolute execution times remain the lowest by over one order of magnitude in both recording and replaying phases.

5.3.1.5 Effect of the Number of Shared Objects

Figure 5.5 shows performance results as a function of the number of shared objects. This is the only application property along whose axis Ditto is overtaken by one of the competing replayers. Though both Ditto and JaRec take advantage of the number of shared objects to lower record overhead, JaRec manages to acquire the position for most efficient recorder past the 64 object mark. The exact same result is observed in the trace file size measurements. In terms of replay execution time, however, JaRec returns to the usual high overheads while Ditto positively reflects the increase in number of objects. As expected, the global order and LEAP replayers are not affected by the change in the amount of objects, as none differentiates between different instances.

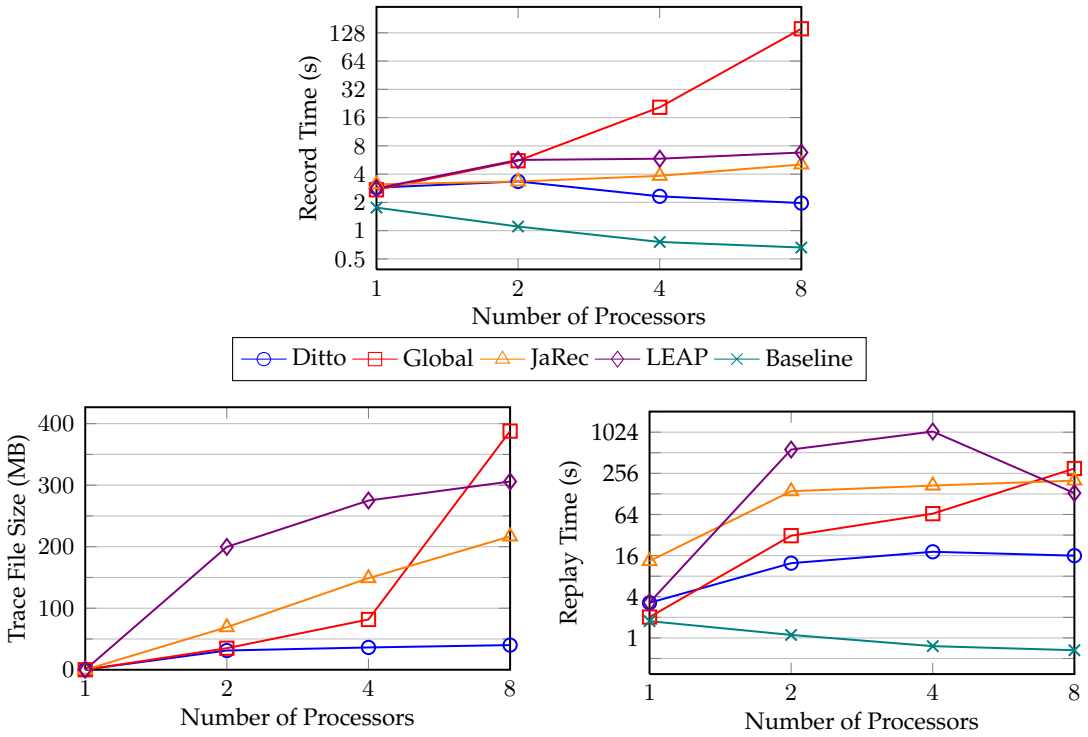


Figure 5.6: Microbenchmark's performance results for Ditto and previous replayers as a function of the number of processors.

5.3.1.6 Effect of the Number of Processors

By limiting the JikesRVM process to a subset of processors in our 8-core test machine, we produced the performance results depicted in Figure 5.6, which show how each replayer deals with a growing number of processors and, as a consequence, parallelism. Ditto is the only algorithm that lowers its record execution time as the number of processors increases, promising increased scalability to future deployments and applications in production environments. Additionally, though the traces it generates grow very slightly in size as we increase the amount of processors, they are smaller than those of other recorders. The replay execution time also increases slightly with the number of processors, but Ditto is still three orders of magnitude more efficient than the second best replayer at the 8 processor mark.

5.3.1.7 Trace File Compression

Trace files generated by Ditto can benefit greatly from compression algorithms to reduce their size. Table 5.2 summarizes the result of using gzip[3] to compress the trace files created during the microbenchmark experiments. The compression rates are quite high, averaging 41.8%.

[3] `http://www.gzip.org/`

Trace file compression rate			
Average	Std. Dev.	Minimum	Maximum
41.8%	7.0%	30.1%	64.1%

Table 5.2: Trace file compression rates across the microbenchmark experiments.

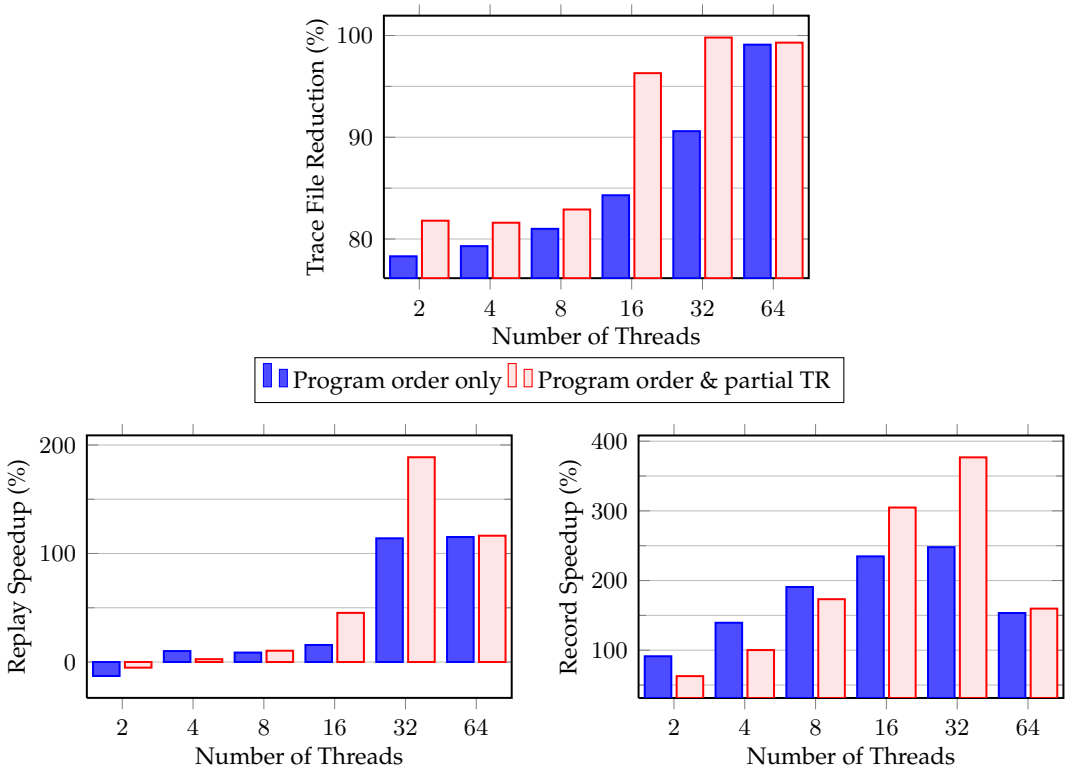


Figure 5.7: Effects of Ditto's pruning algorithm on perfomance.

5.3.1.8 Effects of the pruning algorithm

To demonstrate the potential benefits of Ditto's pruning algorithm, we changed the benchmark's random access pattern to a more sequential pattern in which each thread is responsible for a portion of counters that overlaps with the portions of two other threads. We then measure the trace file reduction, record-time speedup and replay-time speedup of Ditto when using program order pruning only and program order pruning plus partial transitive reduction. Figure 5.7 presents the experimental results thereof. Program order pruning alone is enough to reduce trace file size by 78.3 to 99.1%. The combination of program order pruning and partial transitive reduction reduces the trace file size by 81.6 to 99.8%. With reductions of this magnitude, instead of seeing increased execution times, we actually observe significant drops in overhead, due to the avoided tracing efforts. The observed drop in speedup between 32 and 64 threads seems to be caused by the combination of the memory access pattern of the benchmark and the fact that the number of shared objects is also 64, as the recording time and trace file size are

MolDyn from Java Grande						
	# of Threads	Recorder				
		None	Ditto	Global	JaRec	LEAP
Record Execution Time (s)	2	0.740	17.108	59.718	21.518	>28.248*
	4	0.447	11.402	168.904	13.051	>28.389*
	8	0.404	11.840	>733.649*	16.106	>56.381*
Record Overhead	2	N/A	2212%	7970%	2808%	>3817%*
	4	N/A	2451%	37686%	2820%	>6351%*
	8	N/A	2831%	>181596%*	3887%	>13956%*
Trace File Size	2	N/A	42 KB	80 MB	8 MB	>2 GB*
	4	N/A	98 KB	514 MB	118 MB	>2 GB*
	8	N/A	239 KB	>2 GB*	188 MB	>2 GB*

* Current implementation cannot deal with trace files over 2 GB.

Table 5.3: Record-time performance results for the MolDyn benchmark of the Java Grande suite.

both larger in the 32 thread measurement.

If the application uses a random shared memory access pattern, the benefits are not as visible. Sometimes the additional overhead caused by the maintenance of extra state required by the pruning algorithm is actually not worth the small trace file reductions. As an example, when recording the random access microbenchmark with 8 shared objects and 64 threads, the pruning algorithm incurs 15.2% extra overhead and only achieves a reduction in of 6.4% in trace file size. However, simply increasing the shared objects to 64 changes the picture to 2.7% additional overhead and a 24.0% file size reduction. We believe this risk of extra overhead for little benefit is well worth the potential benefits shown in Figure 5.7

5.3.2 Java Grande Benchmark

The Java Grande benchmark suite[4] contains so-called "Grande" applications – ones with large requirements in either memory, bandwidth or processing power [46]. The multi-threaded version of the suite, designed for parallel execution on shared memory multi-processors, is composed of multi-threaded versions of a subset of the benchmarks in the sequential version. In practice, there are three large scale concurrent applications, all of which are highly computation intensive: (a) MolDyn, a molecular dynamics simulation; (b) MonteCarlo, a monte carlo simulation; and (c) RayTracer, a 3D ray tracer. The purpose of using the Java Grande benchmarks is measuring the record overhead and size of trace files produced by Ditto in comparison with previous recording algorithms.

MolDyn The benchmark performs a simulation of the interactions between a number of particles. The simulation is computationally intense when calculating the forces acting on each

[4] `http://www.epcc.ed.ac.uk/research/java-grande`

particle, which involves an outer loop over all particles and an inner loop over a subset of them. The iterations of the outer loop are performed by different threads.

Table 5.3 reports on the performance results obtained by Ditto and the other recorders (Global, JaRec and LEAP) when recording executions of MolDyn. Ditto manages to obtain the best results in terms of overhead, though it remains still too excessive for production environments in such computationally intensive scenarios. Ditto is also the recorder that creates the smaller trace file, which is almost insignificant in view of the execution time, indicating that most monitored memory accesses were actually not involved in inter-thread interactions. We consider that this result highlights an opportunity for future work: a better may-happen-in-parallel static analysis may be able to further reduce the subset of instructions that are monitored and, as a result, improve recording overhead. Another interesting result is the enormous difference between the insignificant trace file sizes produced by Ditto and the ones produced by the Global and LEAP approaches. Due to a current limitation of our recorder implementations, the trace file size is limited to 2 GB, as Java integers are used as position pointers by some classes in the `java.nio` package. which is always surpassed by LEAP independently of the number of threads, keeping us from materializing its record execution time. Nonetheless, we present the lower bound on that time – the moment the trace reaches 2 GB.

MonteCarlo The benchmark performs a financial simulation, using multiple monte carlo iterations to price products. The monte carlo iterations can be attributed to different threads and run concurrently. The record-time performance results of the various recorders for executions of MonteCarlo are listed in Table 5.4. The record overheads for this benchmark are a lot lower than for MolDyn, but remain above 100%, with Ditto again taking the lead. The Global and LEAP recorders generate large traces, while Ditto and JaRec do not. The result indicates that there are few inter-thread interactions, again suggesting the usefulness of a better static analysis algorithm.

RayTracer This benchmark runs a 3D ray tracer algorithm over a scene. Parallelism is, again, obtained by splitting the outermost loop iterations over multiple threads. Table 5.5 shows the performance results obtained by each recorder, which are very similar to those obtained for the MolDyn benchmark, though with higher overall record overheads. Once again, Ditto generates immensely small trace files, even though its recording overhead is greater than 4000%. We witness that most monitored memory accesses are not involved in inter-thread interactions.

5.3.3 DaCapo Benchmark

The DaCapo benchmark suite[5] consists of a set of open source, real world applications with non-trivial memory loads [6]. Many are concurrent, exhibiting different levels of inter-thread

[5] `http://dacapobench.org`

MonteCarlo from Java Grande						
	# of Threads	Recorder				
		None	Ditto	Global	JaRec	LEAP
Record Execution Time (s)	2	1.325	4.352	11.646	4.619	9.176
	4	0.783	3.002	23.872	3.072	13.031
	8	0.559	2.741	445.383	2.852	57.510
Record Overhead	2	N/A	228%	779%	249%	593%
	4	N/A	283%	2949%	292%	1564%
	8	N/A	390%	79575%	410%	10188%
Trace File Size	2	N/A	127 KB	56 MB	0.12 KB	97 MB
	4	N/A	208 KB	139 MB	0.21 KB	144 MB
	8	N/A	248 KB	1273 MB	0.39 KB	336 MB

Table 5.4: Record-time performance results for the MonteCarlo benchmark of the Java Grande suite.

RayTracer from Java Grande						
	# of Threads	Recorder				
		None	Ditto	Global	JaRec	LEAP
Record Execution Time (s)	2	0.923	49.817	118.342	46.837	>31.007*
	4	0.544	24.899	327.427	25.123	>31,880*
	8	0.443	21.391	>730.406*	23.466	>42.958*
Record Overhead	2	N/A	5297%	12721%	4974%	>3359%*
	4	N/A	4477%	60089%	4518%	>5860%*
	8	N/A	4729%	>164877%*	5197%	>9697%*
Trace File Size	2	N/A	0.76 KB	160 MB	111 KB	>2 GB*
	4	N/A	2.07 KB	940 MB	28 MB	>2 GB*
	8	N/A	4.72 KB	>2 GB*	21 MB	>2 GB*

* Current implementation cannot deal with trace files over 2 GB.

Table 5.5: Record-time performance results for the RayTracer benchmark of the Java Grande suite.

interaction granularity. We evaluate the record-time performance of Ditto and the other recorders using three applications from the 9.12-bach version of the suite. Table 5.6 summarizes the results.

Lusearch The benchmark uses lucene[6] to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. It is multithreaded and driven by one client thread per hardware thread, requiring little interaction between threads. For this benchmark the static analysis algorithm seems to succeed in identifying the right memory accesses involved in thread interactions, allowing all recorders to achieve very low record overheads. In this low interaction application, Ditto's higher complexity makes it the least efficient recorder, albeit with smaller trace files and low absolute overhead. The fastest recorders are

[6] `http://lucene.apache.org`

		Recorder			
		Ditto	Global	JaRec	LEAP
lusearch	Record Overhead	4.56%	1.89%	2.26%	0.69%
	Trace File Size	3 KB	288 KB	3 KB	564 KB
xalan	Record Overhead	5.23%	4.52%	2.71%	2.73%
	Trace File Size	6 KB	475 KB	0.2 KB	485 KB
avrora	Record Overhead	378%	2771%	372%	–*
	Trace File Size	22 MB	565 MB	23 MB	>2 GB*

* Current implementation cannot deal with trace files over 2 GB.

Table 5.6: Record-time performance results for the lusearch, xalan and avrora benchmark applications of the DaCapo suite.

Global and LEAP, though their trace files remain the largest.

Xalan An application that transforms XML documents into HTML. Multithreaded and explicitly driven by the number of hardware threads available, each thread taking an element from a work queue. Interactions between threads are limited to little more than access to the work queue. Our static analysis algorithm identifies shared fields quite well, allowing all recorders to, once again, achieve low overheads. The results are overall very similar to those obtained for the *lusearch* benchmark.

Avrora Simulates a number of programs run on a grid of AVR microcontrollers. It is driven by a single external thread, but internally multithreaded with each simulated element using a thread (i.e. each node in a grid of simulated nodes is threaded). Avrora demonstrates a high volume of fine granularity interactions between simulator threads. These fine interactions lead to record overheads which resemble those obtained when recording the Java Grande applications, with Ditto and JaRec leading in terms of efficiency. This time, however, Ditto generates significant trace data, indicating that the monitored memory accesses were indeed used by threads to interact with each other.

5.3.4 Discussion

From the experiments with the microbenchmark and the Contest benchmark suite described in Section 5.2 we can safely say that Ditto's record and replay algorithms are capable of reproducing non-deterministic behaviors and many associated concurrent bug patterns. The most interesting results, however, arise from the comparative performance evaluation of Ditto and previous solutions, described in Section 5.3.

Firstly, the experiments with the flexible microbenchmark allow us to easily visualize and compare the most relevant runtime characteristics of the recording and replaying algorithms employed by each system. Overall, the results show that Ditto is superior in terms of record-

ing overhead, trace file size and replaying overhead across multiple dimensions of application properties and behaviors. Though Ditto has better record-time overhead, the most significant improvements over previous techniques are observed in the trace file size and replaying overhead. Ditto is the most well-rounded solution, as the other recorder/replayers seem to neglect either trace file bandwidth or replay phase efficiency in favor of lower recording overhead.

The second interesting result was obtained from the evaluation results of Ditto when recording executions of the Java Grande benchmark suite's applications. Though Ditto incurs extremely high overheads, which are still lower than those imposed by the other recorders, it generated trace files are of negligible size. This indicates that the vast majority of monitored operations, responsible for the high overhead, are not involved in actual inter-thread interactions. By itself, the reduction of the trace file is an improvement over the other recorders, but we believe the most important thing to take from these results is an opportunity for future work: the offline static analysis should be improved on to, somehow, correctly categorize the memory accesses not involved in thread interactions as accessing thread local state. We believe that the next step in deterministic replay research should be focused on effectively reducing the amount of monitored operations, instead of new recording algorithms.

A third result we want to underline was observed when measuring performance of recording executions of the DaCapo applications. In the two benchmarks that exhibited very coarse thread interaction granularity, Ditto was the least efficient recorder, even though its absolute overhead was quite modest still. It is our belief that this result is a consequence of Ditto's higher complexity in comparison with the other recorders. We argue, however, that all other solutions have limitations that make them a worse option than Ditto, even when their recording overhead is lower. To start with, the Global and LEAP recorders generate much more trace data than Ditto and JaRec. Secondly, JaRec's thread-focused recording can delay events for a long time during replaying, a behavior which introduces false dependencies between threads and can easily lead to deadlocks. Finally, the replay phase of Ditto is the one that allows for the most concurrency and is, thus, the most faithful to the original execution.

A final and very important result is that neither Ditto nor the other evaluated replay systems are capable of providing sufficiently low overheads for every concurrent application. From our experiments, it is clear that only coarse thread interactions can be efficiently reproduced. For applications with finer inter-thread interactions, we place our hopes in future work on static analysis of parallelism.

6 Conclusions

The rise of multi-processor machines in the past decade has brought on the ubiquity of concurrent paradigms of software development. Reasoning about concurrency does not come easy to programmers, making programs developed under these paradigms especially prone to faults arising from unanticipated interactions between parallel tasks. These bugs are easy to hatch, but hard to find, as most have quite specific and rare pre-conditions. Conventional debugging methodologies are not prepared to handle non-deterministic faults. As such, there is a need for debugging tools capable of reproducing these faults within a reasonable amount of time, so that developers can dedicate their time to fixing them. The value proposition of deterministic replayers is just that: enabling debugging methodologies that rely on fault-determinism to be used in the context of non-deterministic concurrent programs. Though executions can be reproduced efficiently if the program is perfectly synchronized or if it occurs on a single processor machines, the same cannot be said for execution on multi-processors. This is the open problem which we addressed in this thesis.

We developed Ditto, a deterministic replay system for the JVM, capable of correctly replaying executions of imperfectly synchronized applications on multi-processor machines. It is based on a pair of novel execution recording and replaying algorithms that combine state-of-the-art and original techniques to outperform previous replayers aimed at Java programs. These techniques include (a) managing the semantic differences between load and store memory accesses to reduce trace data and increase replay concurrency; (b) serializing memory accesses at the finest possible granularity, distinguishing between distinct static, instance and array fields; (c) using a modified version of transitive reduction to prune the trace file on-the-fly; and (d) taking advantage of TLO static analysis, escape analysis and compiler optimizations to reduce the amount of monitored memory accesses. Moreover, we introduce a set of trace file optimizations that greatly lower its size.

Ditto was successfully evaluated to ascertain its capability to reproduce different concurrent bug patterns and highly non-deterministic executions. Its performance was compared with that of a global-order based replayer similar to DejaVu [8], JaRec [16] and LEAP [20]. Ditto outperformed all of them in terms of record-time overhead, trace file size and replay-time overhead across multiple application properties, namely number of threads, number of processors, number of memory access operations, load to store ratio, number of shared objects and number of fields per shared object. Our experiments clearly show that Ditto is the most

well-rounded system, performing well in all three indicators, while others neglect trace file size and/or replay overhead. Even so, Ditto's record-time overhead is still too high for production environments when targeting applications with fine-grained inter-thread interactions.

6.1 Future Work

It is our belief that future work should focus on creating better static analysis modules to identify static, instance and array field accesses that are involved in inter-thread interactions. This is evidenced by the negligible amount of trace data generated by Ditto when recording executions of the Java Grande benchmark applications, even though the overhead was unreasonably high. Such a result implies that very few inter-thread interactions actually occurred, but that the TLO static analysis was overly conservative.

It would also be beneficial to identify thread-locality not at field level, but at individual memory access level; a field may be involved in inter-thread interactions in some accesses, but not in others. This may actually be possible using Soot's implementation of TLO, but we did not tackle this potential improvement.

Bibliography

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, January 2000.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, January 2005.

[3] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. *SIGPLAN Not.*, 34(10):314–324, October 1999.

[4] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 193–206. ACM, 2009.

[5] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, pages 194–206. ACM, 1991.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41:169–190, October 2006.

[7] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14:80–107, February 1996.

[8] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59. ACM, 1998.

[9] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. In *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[10] Carl Dionne, Marc Feeley, and Jocelyn Desbiens. A taxonomy of distributed debuggers based on execution replay. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996.

[11] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36:211–224, December 2002.

[12] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130. ACM, 2008.

[13] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):267–279, March 2007.

[14] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2003.

[15] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 27–27. USENIX Association, 2006.

[16] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 34:523–547, May 2004.

[17] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208. USENIX Association, 2008.

[18] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 353–364. IEEE Computer Society, 2007.

[19] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276. IEEE Computer Society, 2008.

[20] Jeff Huang, Peng Liu, and Charles Zhang. 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, pages 207–216. ACM, 2010.

[21] Joel Huselius. Debugging parallel systems: A state of the art report. Technical report, 2002.

[22] João M. Silva and Luís Veiga. Reprodução probabilística de execuções na jvm em multiprocessadores. In *INFORUM 2012*.

[23] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1. USENIX Association, 2005.

[24] Ravi Konuru. Deterministic replay of distributed java applications. In *In Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium*, pages 219–228, 2000.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[26] Tobias Landes. Dynamic vector clocks for consistent ordering of events in dynamic distributed applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 31–37. CSREA Press, 2006.

[27] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36:471–482, April 1987.

[28] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42:329–339, March 2008.

[29] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier B.V., 1989.

[30] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21:593–622, December 1989.

[31] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300. IEEE Computer Society, 2008.

[32] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 73–84. ACM, 2009.

[33] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 229–240. ACM, 2006.

[34] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 284–295. IEEE Computer Society, 2005.

[35] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, PADD '93, pages 1–11. ACM, 1993.

[36] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 97–108. ACM, 2009.

[37] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192. ACM, 2009.

[38] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18. USENIX Association, 1995.

[39] Gilles Pokam, Cristiano Pereira, Klaus Danne, Lynda Yang, Sam King, and Josep Torrellas. Hardware and software approaches for deterministic multi-processor replay of concurrent programs. *Intel®Technology Journal*, 13, 2009.

[40] M. Ronsse, K. De Bosschere, and J. Chassin de Kergommeaux. Execution replay and debugging. *eprint arXiv:cs/0011006*, November 2000.

[41] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17:133–152, May 1999.

[42] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 258–266. ACM, 1996.

[43] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 69–76. ACM, 2005.

[44] José Simão, Tiago Garrochinho, and Luís Veiga. A checkpointing-enabled and resource-aware java virtual machine for efficient and robust e-science applications in grid environments. *Concurrency and Computation: Practice and Experience*, 24(13):1421–1442, 2012.

[45] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, FTCS '96, pages 250–. IEEE Computer Society, 1996.

[46] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 8–8. ACM, 2001.

[47] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *SIGARCH Comput. Archit. News*, 30:123–134, May 2002.

[48] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 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, pages 3–3. USENIX Association, 2004.

[49] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 158–167. ACM, 2000.

[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 13–. IBM Press, 1999.

[51] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135. ACM, 2003.

[52] Min Xu, Mark D. Hill, and Rastislav Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 49–60. ACM, 2006.

[53] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, and Vmware Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.