Output-Deterministic Replay for Multicore Debugging

Gautam Altekar
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-108
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-108.html

August 3, 2009

ODR: Output-Deterministic Replay for Multicore Debugging – v. 1842

Gautam Altekar
UC Berkeley
galtekar@cs.berkeley.edu

Ion Stoica
UC Berkeley
istoica@cs.berkeley.edu

ABSTRACT

Reproducing bugs is hard. Deterministic replay systems aim to address this problem, by providing a high-fidelity replica of an original program execution that can be repeatedly executed to zero-in on bugs. Unfortunately, existing replay systems for multiprocessor programs fall short. These systems either incur high overheads, rely on non-standard multiprocessor hardware, or fail to reliably reproduce executions. Their primary stumbling block is data races – a source of non-determinism that must be captured if executions are to be faithfully reproduced.

In this paper, we present ODR–a software-only replay system that reproduces bugs and provides low-overhead multiprocessor recording. The key observation behind ODR is that, for debugging purposes, a replay system does *not* need to generate a high-fidelity replica of the original execution. Instead, it suffices to produce *any* execution that exhibits the same outputs as the original. Guided by this observation, ODR relaxes its fidelity guarantees to avoid the problem of reproducing data-races all-together. The result is a system that replays real multiprocessor applications, such as Apache, MySQL, and the Java Virtual Machine, and provides low record-mode overheads.

1. INTRODUCTION

Computer software often fails. These failures, due to software errors, manifest in the form of crashes, corrupt data, or service interruption. To understand and ultimately prevent failures, developers employ cyclic debugging – they re-execute the program several times in an effort to zero-in on the root cause. Non-deterministic failures, however, are immune to this debugging technique. That's because they may not occur in a re-execution of the program.

Non-deterministic failures can be reproduced using *deterministic replay* (or *record-replay*) technology. Deterministic replay works by first capturing data from non-deterministic sources, such as the keyboard and network, and then substituting the same data in subsequent re-executions of the program. Many replay systems have been built over the years, and the resulting experience indicates that replay is valuable in finding and reasoning about failures [6].

The ideal record-replay system has three key properties. First, it produces a high-fidelity replica of the original program execution, thereby enabling cyclic debugging of non-deterministic failures. Second, it incurs low recording overhead, which in turn enables in-production operation and ensures minimal execution perturbation. Third, it supports multi-threaded (parallel) software running on commodity multi-core processors. However, despite much research, the ideal replay system still remains out of reach.

A chief obstacle to building the ideal system is data-races. These sources of non-determinism are prevalent in modern software. Some are errors, but many are intentional. In either case, the ideal-replay system must reproduce them if it is to provide high-fidelity replay. Some replay systems reproduce races by recording their outcomes, but they incur high recording overheads [2, 4]. Other systems achieve low record overhead, but rely on non-standard hardware [14]. Still others assume data-race freedom, but fail to reliably reproduce failures [17].

In this paper, we present ODR–a software-only replay system that reliably reproduces failures and provides low-overhead multiprocessor recording. The key observation behind ODR is that a high-fidelity replay execution, though sufficient, is *not* necessary for replay-debugging. Instead, it suffices to produce *any* execution that exhibits the same output, even if that execution differs from the original. This observation permits ODR to relax its fidelity guarantees and, in so doing, enables it to all-together avoid the problem of reproducing and hence recording data-race outcomes.

The key problem ODR must address is that of reproducing a failed execution without recording the outcomes of data-races. This is challenging because the occurrence of a failure depends in part on the outcomes of races. To address this challenge, rather than record data-race outcomes, ODR infers the data-race outcomes of an output-deterministic run. Once inferred, ODR substitutes these

values in subsequent program executions. The result is output-deterministic replay.

To infer data-race outcomes, ODR uses a technique called *Deterministic-Run Inference*, or DRI for short. DRI leverages the original output to search the space of executions for one that exhibits the same outputs as the original. An exhaustive search of the execution space is intractable. But carefully selected clues recorded during the original execution and memory-consistency relaxations allow ODR to home-in on an output-deterministic execution in polynomial time for several real-world programs.

To evaluate ODR, we used several sophisticated multi-threaded applications, including Apache and the Splash 2 suite. Like most replay systems, ODR is not without its limitations (see Section 11). While during the recording phase our Linux/x86 implementation slows down the native execution by only 1.6x on average, inference times can be impractically long for many programs. However, we show that the decision of which information to record can be used effectively to trade between the recording and the inference times. For example, while recording all branches slows down the original execution from 1.6x to 4.5x on average, in some cases, this can decrease the inference times by orders of magnitude. Despite its limitations, we believe ODR represents a promising approach to address the difficult problem of reproducing bugs and enable cyclic debugging for in-production parallel applications running on modern multi-core processors.

2. THE PROBLEM

ODR addresses the *output-failure replay problem*. In a nutshell, the problem is to ensure that all failures visible in the output of some original program run are also visible in the replay runs of the same program. Examples of output-failures include assertion violations, crashes, core dumps, and corrupted data. Solving the output-failure replay problem is important because a vast majority of software errors result in output-visible failures. Hence reproduction of these failures would enable cyclic debugging of most software errors.

In contrast with the problem addressed by traditional replay systems, the output-failure replay problem is narrower in scope. Specifically, it is narrower than the *execution replay problem*, which concerns the reproduction of all original-execution properties and not just those of output-failures. It is even narrower than the *failure replay problem*, which concerns the reproduction of all failures, output-visible or not. The latter includes timing related failures such as unexpected delays between two outputs.

Any system that addresses the output-failure replay problem should replay output-failures. In addition, to be practical, the system must meet the following requirements.

Support multiple processors or cores. Multiple cores are a reality in modern commodity machines. A practical replay system should allow applications to take full advantage of those cores.

int status = ALIVE, int *reaped = NULL

Master (Thread 1; CPU 1)	Worker (Thread 2; CPU 2)
1 r0 = status	1 r1 = input
2 if (r0 == DEAD)	2 if (r1 == DIE or END)
3 *reaped++	3 status = DEAD

Figure 1: Benign races can prevent even non-concurrency bugs from being reproduced, as shown in this example adapted from the Apache web-server. The master thread periodically polls the worker's status, without acquiring any locks, to determine if it should be reaped. It crashes only if it finds that the worker is DEAD.

Replay all programs. A practical tool should be able to replay arbitrary program binaries, including those with data races. Bugs may not be reproduced if the outcomes of these races are not reproduced.

Support efficient and scalable recording. Production operation is possible only if the system has low record overhead. Moreover, this overhead must remain low as the number of processor cores increases.

Require only commodity hardware. A software-only replay method can work in a variety of computing environments. Such wide-applicability is possible only if the system does not introduce additional hardware complexity or require unconventional hardware.

3. BACKGROUND: VALUE DETERMINISM

The classic approach to the output-failure replay problem is *value-determinism*. Value-determinism stipulates that a replay run reads and writes the same values to and from memory, at the same execution points, as the original run. Figure 2(b) shows an example of a value-deterministic run of the code in Figure 1. The execution is value-deterministic because it reads the value DEAD from variable status at execution point 1.1 and writes the value DEAD at 2.3, just like the original run.

Value-determinism is not perfect: it it does not guarantee causal ordering of instructions. For instance, in Figure 2(b), the master thread's read of status returns DEAD even though it happens before the worker thread writes DEAD to it. Despite this imperfection, value-determinism has proven effective in debugging [2] for two reasons. First, it ensures that program output, and hence most operator-visible failures such as assertion failures, crashes, core dumps, and file corruption, are reproduced. Second, within each thread, it provides memory-access values consistent with the failure, hence helping developers to trace the chain of causality from the failure to its root cause.

The key challenge of building a value-deterministic replay system is in reproducing *data-races values*. Data-races are often benign and intentionally introduced to improve performance. Sometimes they are inadvertent and result in software failures. No matter whether data-races are

(a) Original	(b) Value-deterministic	(c) Output-deterministic	(d) Non-deterministic
2.1 `r1 = DIE`	2.1 `r1 = DIE`	2.1 `r1 = END`	2.1 `r1 = DIE`
2.2 `if (DIE...)`	2.2 `if (DIE...)`	2.2 `if (END...)`	2.2 `if (DIE...)`
2.3 `status = DEAD`	1.1 `r0 = DEAD`	1.1 `r0 = DEAD`	1.1 `r0 = ALIVE`
1.1 `r0 = DEAD`	2.3 `status = DEAD`	2.3 `status = DEAD`	2.3 `status = DEAD`
1.2 `if (DEAD...)`	1.2 `if (DEAD...)`	1.2 `if (DEAD...)`	1.2 `if (ALIVE...)`
1.3 `*reaped++`	1.3 `*reaped++`	1.3 `*reaped++`	
`Segmentation fault`	`Segmentation fault`	`Segmentation fault`	*no output*

Figure 2: The totally-ordered execution trace and output of (a) the original run and (b-d) various replay runs of the code in Figure 1. Each replay trace show-cases a different determinism guarantee

benign or not reproducing their values is critical. Data-race non-determinism causes replay execution to diverge from the original, hence preventing down-stream errors, concurrency-related or otherwise, from being reproduced. Figure 2(d) shows how a benign data-race can mask a null-pointer dereference bug in the code in Figure 1. There, the master thread does not dereference the null-pointer `reaped` during replay because it reads `status` before the worker writes it. Consequently, the execution does not crash like the original.

Several value-deterministic systems address the data-race divergence problem, but they fall short of our requirements. For instance, content-based systems record and replay the values of shared-memory accesses and, in the process, those of racing accesses [2]. They can be implemented entirely in software and can replay arbitrary programs, but incur high record-mode overheads (e.g., 5x slowdown [2]). Order-based replay systems record and replay the ordering of shared-memory accesses. They provide low record-overhead at the software-level, but only for programs with limited false-sharing [4] or no data-races [17]. Finally, hardware-assisted systems can replay data-races at very low record-mode costs, but require non-commodity hardware [9, 14].

4. OVERVIEW

In this section, we present an overview of our approach to the output-failure replay problem. In Section 4.1, we present *output determinism*, the concept underlying our approach. Then we introduce key definitions in Section 4.2, followed by the central building block of our approach, *Deterministic-Run Inference*, in Section 4.3. In Section 4.4, we discuss the design space and trade-offs of our approach and finally, in Section 4.5, we identify the design points with which we perform our evaluation.

4.1 Output Determinism

To address the output-failure replay problem we use *output-determinism*. Output-determinism dictates that the replay run outputs the same values as the original run. We define *output* as program values sent to devices such as the screen, network, or disk. Figure 2(c) gives an example of an output-deterministic run of the code in Figure 1. The execution is output-deterministic because it outputs the string `Segmentation fault` to the screen just like the original run.

Output-determinism is weaker than value-determinism. Specifically, output-determinism does not guarantee that the replay run will read and write the same values as the original. For example, the output-deterministic trace in Figure 2(c) reads `END` for the input while the original trace, shown in Figure 2(a), reads `DIE`. Also output-determinism does not guarantee that the replay run takes the same path as the original execution.

Despite its imperfections, we argue that output-determinism is effective for debugging purposes, for two reasons. First, output-determinism ensures that output-visible failures, such as assertion failures, crashes, core dumps, and file corruption, are reproduced. For example, the output-deterministic run in Figure 2(c) produces a crash just like the original run. Second, it provides memory-access values that, although may differ from the original values, are nonetheless consistent with the failure. For example, we can tell that the master segfaults because the read of `status` returns `DEAD`, and that in turn was caused by the worker writing `DEAD` to the same variable.

The chief benefit of output-determinism over value-determinism is that it does not require the values of data races to be the same as the original values. In fact, by shifting the focus of determinism to outputs rather than values, output-determinism enables us to circumvent the need to record and replay data-races all-together. Without the need to reproduce data-race values, we are freed from the tradeoffs that encumber traditional replay systems. The result, as we detail in the following sections, is `ODR`–an Output-Deterministic Replay system that meets all the requirements given in Section 2.

4.2 Definitions

In this section, we define key terms used in the remainder of the paper.

A *program* is a set of instruction-sequences, one for each processor. Each sequence consists of four key instruction types. A *read/write* instruction reads/writes a byte from/to a memory location. The *input* instruction accepts a byte-value arriving from an input device (e.g., the keyboard) into a register. The *output* instruction prints a byte-value to an output device (e.g., screen) from a register. A *conditional branch* jumps to a specified program location iff its register parameter is non-zero. To simplify our presentation, we purposely omit other instruction types (such as arithmetic operations and indirect jumps). We also assume that traditionally hardware-implemented features are implemented programatically using these simple instruction types. For example, interrupts may be modeled as an input and a conditional

branch to a handler, done at every instruction.

A *run* or *execution* of a program is a sequence of program states, where each state is a mapping from memory and register locations to values. The first state of a run maps all locations to 0. Each subsequent state is derived by executing instructions, chosen in program order from each processor's instruction sequence, one at a time in some total order. The content of these subsequent states are a function of previous states, with two exceptions: the values returned by memory read instructions are some function of previous states and the underlying machine's memory consistency model, and the values returned by input instructions are arbitrary (e.g., user-provided). Finally, the last state of a run is that immediately following the last available instruction from the sequence of either processor.

A run's *determinant* is a triple that uniquely identifes the run. This triple consists of a schedule-trace, input-trace, and a read-trace. A program *schedule-trace* is a sequence of thread identifiers that specifies the ordering in which instructions from different processors are interleaved (i.e., a total-ordering). An *input-trace* is a finite sequence of bytes consumed by input instructions. A *read-trace* is a finite sequence of bytes returned by read instructions. For example, Figure 2(b) shows a run of schedule-trace $(2, 2, 1, 2, 1, 1)$ with input-trace (DIE) and read-trace $(\text{DEAD}, 0)$, where $r0$ reads DEAD, and `*reaped` reads 0. Figure 2(c) shows another run with the same schedule-trace with the same read-trace, but consuming a different input-trace, (END).

A run and its determinant are equivalent in the sense that either can be derived from the other. Given a determinant, one can generate a run (a sequence of states), by (1) executing instructions in the interleaving specified by the scheduling order, (2) substituting the value at the i-th input-trace position for the i-th input instruction's return value, and (3) substituting the value at the j-th read-trace position for the j-th read instruction's return value. The reverse transformation is straightforward.

We say that a run is *M-consistent* iff its read-trace (a component of the run's determinant) is in the set of all possible read-traces for the run's schedule-trace, input-trace, and memory consistency model M. For example, the run in Figure 2(a) is strict-consistency conformant [**?**] because the values returned by its reads are that of the most-recent write for the given schedule-trace and input-trace. Weaker consistency models may have multiple valid read-trace, and hence consistent runs, for a given schedule and input-trace. To simplify our presentation, we assume that the original run is strict-consistent. We omit a run's memory model qualifier when consistency is clear from context.

4.3 Deterministic-Run Inference

The central challenge in building `ODR` is that of reproducing the original output without knowing the entire read-trace, i.e., without knowing the outcomes of data races. To address this challenge, `ODR` employs *Deterministic-Run Inference* (DRI) – a method that returns the determinant of an output-deterministic run. Figure 3 shows how `ODR` uses DRI. `ODR` records information about the original run and then gives it to DRI in the form of a *query*. Minimally, the query contains the original output. DRI then returns a determinant that `ODR` uses, in the future, to quickly regenerate an output-deterministic run.

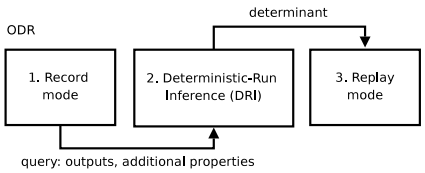

Figure 3: `ODR` uses Deterministic-Run Inference (DRI) to compute the determinant of an output-deterministic run. The determinant, which includes the values of data-races, is then used to drive future program runs.

In its simplest form, DRI searches the space of all strict-consistent runs and returns the determinant of the first output-deterministic run it finds. Conceptually, it works iteratively in two steps. In the first step, DRI selects a run from the space of all runs. In the second step, DRI compares the output of the chosen run to that of the original. If they match, then the search terminates; DRI has found an output-deterministic run. If they do not match, then DRI repeats the first step and selects another run. At some point DRI terminates, since the space of struct-consistent runs contains at least one output-deterministic run – the original run. Figure 4(a) gives possible first and last iterations of DRI with respect to the original run in Figure 2(a). Here, we assume that the order in which DRI explores the run space is arbitrary. Note that the run space may contain multiple output-deterministic runs, in which case DRI will select the first one it finds. In the example, the selected run is different from the original run, as $r1$ reads value `END`, instead of `DIE`.

An exhaustive search of program runs is intractable for all but the simplest of programs. To make the search tractable, DRI employs two techniques. The first is to *direct the search* along runs that share the same properties as the original run. Figure 4(b) shows an example illustrating this idea. In this case, DRI considers only those runs with the same schedule, input, and read trace as the original run. The benefit of directed search is that it enables DRI to prune vast portions of the search space. In Figure 4(b), for example, knowledge of the original run's input and schedule trace allows DRI to converge on an output-deterministic run after exploring just one run. The effectiveness of directed search depends on the information recorded during the original run; more information enables greater pruning of the search space. This illustrates the trade-off between the recording overhead and the DRI overhead.

The second technique is to *relax the memory-consistency* of all runs in the run space. The benefit of relaxing the memory-model is that it enables DRI to search fewer runs. In general, a weaker consistency model permits more runs

(a) Exhaustive search		(b) Query-directed search
1st iteration	*last iteration*	*1st & last iteration*
`2.1 r1 = REQ`	`2.1 r1 = END`	`2.1 r1 = DIE`
`2.2 if (REQ...)`	`2.2 if (END...)`	`2.2 if (DIE...)`
`1.1 r0 = ALIVE`	`2.3 status = DEAD`	`2.3 status = DEAD`
`1.2 if (ALIVE...)`	`1.1 r0 = DEAD`	`1.1 r0 = DEAD`
	`1.2 if (DEAD...)`	`1.2 if (DEAD...)`
	`1.3 *reaped++`	`1.3 *reaped++`
No output	`Segmentation fault`	`Segmentation fault`

Figure 4: Possible first and last iterations of DRI using exhaustive search (a) of all strict-consistent runs, and (b) of strict-consistent runs with the original input and schedule trace, both for the program in Figure 2(a). DRI converges in an exponential number of iterations in case (a), and, in this example, in just one iteration in case (b).

(a) Strict consistency	(b) Null consistency
`2.1 r1 = DIE`	`2.1 r1 = REQ`
`2.2 if (DIE...)`	`2.2 if (REQ...)`
`2.3 status = DEAD`	`1.1 r0 = DEAD`
`1.1 r0 = DEAD`	`1.2 if (DEAD...)`
`1.2 if (DEAD...)`	`1.3 *reaped++`
`1.3 *reaped++`	
`Segmentation fault`	`Segmentation fault`

Figure 5: Possible last iterations of DRI on the space of runs for the strongest and weakest consistency models we consider. The null consistency model, the weakest, enables DRI to ignore scheduling order of all accesses, and hence converge faster that strict consistency.

matching the original's output (than a stronger model), which ultimately enables DRI to find such a run faster.

To illustrate the benefit, Figure 4.3 shows two output-deterministic runs for the *strict* and the hypothetical *null* consistency memory models. Strict consistency, the strongest consistency model we consider, guarantees that reads will return the value of the most-recent write in schedule order. *Null* consistency, the weakest consistency model we consider, makes no guarantees on the value a read may return – it may be completely arbitrary. For example, in Figure 4.3(c), thread 1 reads `DEAD` for `status` even though thread 2 never wrote `DEAD` to it.

To find a strict-consistent output-deterministic run, DRI may have to search all possible schedules in the worst case. But under null-consistency, DRI needs to search only along one arbitrary selected schedule. After all, there must exist a null-consistent run that reads the same values as the original for any given schedule.

While relaxing the consistency memory model reduces the number of searches, this benefit does not come for free. A relaxed memory model makes it harder for the developer to track the cause of a bug, especially across multiple threads.

4.4 Design Space

The challenge in designing DRI is to determine just how much to direct the search and relax memory consistency. In conjunction with our basic approach of search, these questions open the door to an uncharted inference design space. In this section, we describe this space and identify our targets within it.

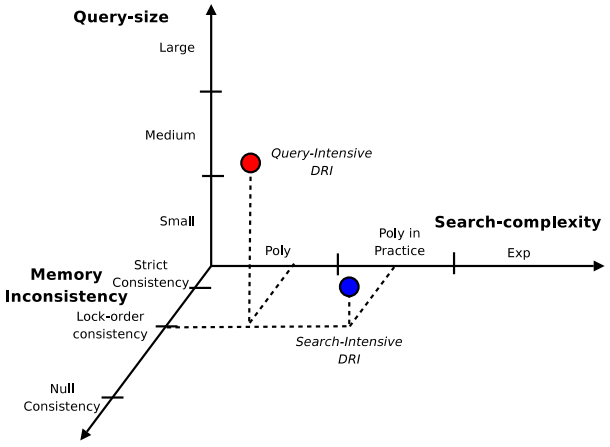

Figure 6: The design space for DRI. Exhaustive search is intractable, but providing more information in the query or relaxing consistency can lower it dramatically.

Figure 6 shows the three-dimensional design space for DRI. The first dimension in this space is *search-complexity*. It measures how long it takes DRI to find an output - deterministic run. The second design dimension, *query-size*, measures the amount of original run information used to direct the search. The third dimension, *memory-inconsistency*, captures the degree to which the memory consistency of the inferred run has been relaxed.

The most desirable search-complexity is polynomial search complexity. It can be achieved by making substantial sacrifices in other dimensions, e.g., recording more information during the original run, or using a weaker consistency model as described in Section 4.3. The least desirable search-complexity is exponential search complexity. An exhaustive search can accomplish this, but at the cost of developer patience. The benefit is extremely low-overhead recording and ease-of-debugging due to a small query-size and low value-inconsistency, respectively.

A query containing just the outputs of the original run has the smallest query-size. All DRI queries must contain the original output. A query containing the original input, schedule, and read trace in addition to output has the greatest query-size. The latter results in polynomial search-complexity (more specifically, constant time search-complexity), but carries a penalty of large record-

mode slowdown. For instance, capturing a read-trace will result in at least a 5x penalty on modern commodity machines [2].

As discussed above, lowering consistency requirements results in substantial search-space reductions, but makes it harder to track causality across threads during debugging.

4.5 Design Targets

In this paper, we evaluate two point in DRI design space: Search-Intensive DRI (SI-DRI) and Query-Intensive DRI (QI-DRI).

SI-DRI strives for a practical compromise among the extremities of the DRI design space. In particular, SI-DRI targets polynomial search-complexity for several applications, though not all – a goal we refer to as "poly in practice". Our results in Section 10 indicate that polynomial complexity holds for several real-world applications.

SI-DRI also targets a query-size that, while greater than just those of the outputs, is still low enough to be useful for at least periodic production use. In particular, in addition to outputs, the query includes the inputs, the lock order, and a sample of the original execution path. The path sample includes every lock instruction, as well as the instructions associated with every input and output in the original execution.

Finally, SI-DRI relaxes the memory consistency model from strict consistency to lock-order consistency – a consistency model that produces strict-consistent runs for data-race free programs.

The second design point we evaluate, Query-Intensive DRI, is almost identical to Search-Intensive DRI. The key difference is that QI-DRI requires that the query contain a path-sample for every branch in the original run–considerably more information than required by SI-DRI. Requiring full path information in the query inflates recording overhead, but results in polynomial search-complexity.

5. Search-Intensive DRI

In this section, we present Search-Intensive DRI (SI-DRI), one inference method with which we evaluate `ODR`. We begin with an overview of what SI-DRI does. Then we present a bare-bones version of its algorithm (called core SI-DRI). Finally, we apply directed search and consistency relaxation to this core algorithm to yield SI-DRI, the finished product.

5.1 Overview

SI-DRI accepts a query and produces an output-deterministic run, like any variant of DRI.

Specifically, in addition to the output-trace, a SI-DRI query must contain three other pieces of information from the original run:

- *input-trace*, a finite sequence of bytes consumed by the input instructions of the original run (see Section 4.2).

(a) LOR-consistent run 1	(b) LOR-consistent run 2
`2.1 r1 = DIE`	`2.1 r1 = DIE`
`2.2 if (DIE...)`	`2.2 if (DIE...)`
`1.1 r0 = ALIVE`	`1.1 r0 = DEAD`
`2.3 status = DEAD`	`2.3 status = DEAD`
`1.2 if (ALIVE...)`	`1.2 if (DEAD...)`
	`1.3 *reaped++`
	`Segmentation fault`

Figure 7: The set of all lock-order (LOR) consistent runs for the schedule (2,2,1,2,1,1) and input DIE. Since the runs' schedule is lock-order conformant with the original lock order (see Figure 1(a)), at least one is guaranteed to be output-deterministic.

- *lock-order*, a total ordering of lock instructions in the original run. Each item in the sequence is a (t, c)-pair where t is the thread index and c is the lock instruction count. For example, $(1, 4)$ denotes a lock that was executed as the 4th instruction of thread 1. The lock-order sequence induces a partial ordering on the program instructions. For instance, sequence $((1, 4), (2, 2), (1, 10))$ captures the fact that the 4th instruction of thread 1 is executed before the 2nd instruction of thread 2, which in turn is executed before the 10th instruction on thread 1.
- *path-sample*, a sampling of instructions in the original run. Each instruction in the path-sample is represented by a (t, c, l)-tuple, where t is the thread index, c is the instruction's count, and l is the program location of that instruction. The path-sample includes one such tuple for each input, output, and lock instruction in the original program path.

SI-DRI produces an output-deterministic run conforming to the hypothetical *lock-order consistency (LOR)* memory model. Intuitively, a read in this model may return either the value of the most-recently-scheduled write to the same memory location or the value of any racing write. We say that two accesses race if they both access the same location and neither access happens before the other according to the lock-order of the run. Figure 5.1 shows two LOR-consistent runs. Since the example has no locks, the read and write of `status` are trivially unordered by the lock-order and therefore race. Under lock-order consistency, a read may return the value of a racing write even if that write is scheduled after the read, as with the read of `status` in Figure 5.1(b); $r0$ reads DEAD *before* this value is written at instruction 2.3. Section 5.4 details how SI-DRI leverages this relaxed memory model.

5.2 Core Algorithm

The core SI-DRI search algorithm, depicted in Figure 9, performs an exhaustive depth-first search on the space of strict-consistent program runs. Since the space may have infinite depth (due to loops), SI-DRI bounds the search by n, the length of the original execution, which we assume is in the original output. Each iteration has four steps.

In the first step, *schedule-select*, SI-DRI selects an n-length schedule-trace from the space of all n-length schedule-

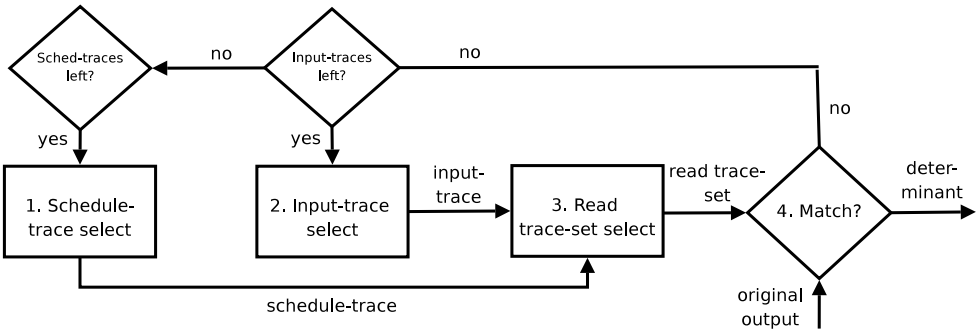

Figure 8: The core Search-Intensive DRI algorithm exhaustively searches the space of bounded-length strict-consistent runs for one that outputs the same values as the original run. It makes use of only the original outputs in the query and consequently has exponential search-complexity.

traces. In the second step, *input-select*, SI-DRI selects an n-length input-trace from the space of all n-length input-traces. In the third step, *read trace-set selection*, SI-DRI produces the set of all valid read-traces for the previously selected schedule-trace, input-trace, and target memory model. In the final step, *output matching*, SI-DRI conceptually runs the program for every read-trace in the read-trace set produced in the previous stage to see if any produce the original output.

If the run produces the same output as the original run, then the search terminates, and SI-DRI returns the determinant of that run. If the run does not produce the same output, then SI-DRI continues the search. For the next search iteration, SI-DRI will choose a different input-trace, if there are any unexplored in the input-trace space. Otherwise, SI-DRI will choose a different schedule-trace, if there are any unexplored in the schedule-trace space. The search will terminate before all schedules are exhausted because there exists some iteration in which SI-DRI selects the original run's schedule, input, and read trace.

5.3 Query-Directed Search

SI-DRI leverages query information to reduce the space of schedule-traces, input-traces, and read-trace sets, using a technique we term query-guidance. SI-DRI uses three different types of query-guidance methods, each corresponding to a selection stage, as depicted in Figure 9.

The first guidance-method is *lock-order guidance*, used in the schedule-selection stage. Lock-order guidance uses the lock-order given in the query to generate a schedule of length n that conforms to it. By considering only those schedule-traces that conform to the lock-order, lock-order guidance avoids searching all schedules.

The second guidance-method is *input-trace guidance*, used in the input-selection stage. Input guidance simply reproduces the original input sequence found in the query. It does not have much work to do because all of the input is given in the query. Input guidances provides an exponential reduction of the input-space.

The third guidance-method is *read trace-set guidance*, used in the read trace-set selection stage. Read trace-set guidance chooses runs that are more likely to be output deterministic. To do this, it leverages the schedule-trace and input-trace chosen in previous stages, and the path-sample provided in the query. This stage is the most complicated and is discussed in detail in Section 7.2.

5.4 Relaxed Consistency Reduction

The key observation behind SI-DRI's consistency relaxation is that, to find an output-deterministic run under LOR-consistency, we need only consider LOR-consistent runs along one schedule. The only restriction on the selected schedule is that it must be lock-order conformant, a restriction which is satisfied by SI-DRI's lock-order guidance. Figure 5.1 shows all LOR-consistent runs for an arbitrarily chosen lock-order conformant schedule. At least one is output-deterministic.

It suffices to consider any lock-order conformant schedule S' because for any such schedule, there exists an input and LOR-consistent read-trace that produces the original output. This holds because given the original schedule S, input, and read-trace, which is by definition a LOR-consistent run, we can construct another LOR-consistent read-trace that when used in conjunction with schedule S' and original input, produces the original output. We omit a formal proof of this, but the intuition is that, one can rearrange the read-values of concurrent-reads in the read-trace of S to yield the read-trace of S'. For example, the read-trace of Figure 2(a) and Figure 5.1(b) are both (DEAD, 0)–a trivial rearrangement.

We describe how to automatically generate the set of LOR-consistent read-traces in Section 7.2.2.

6. Query-Intensive DRI

In addition to SI-DRI, we evaluate another DRI variant called Query-Intensive DRI (QI-DRI). The only difference between SI-DRI and QI-DRI is that the latter requires a query path-sample with many samples. Specifically, it requires a sample point for the instruction following every branch in the original run. While this additional information increases the recording overhead, it leads to a significant reduction of the inference time. This is to be expected since adding more points to the path-sample

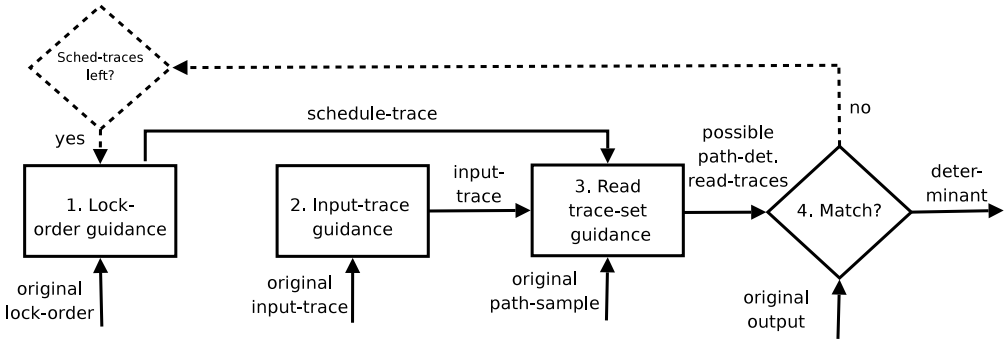

Figure 9: The Search-Intensive DRI algorithm with (dashed and solid) just query-guidance methods applied and (just solid) with both query-guidance and consistency relaxation applied. Consistency relaxation eliminates the need to search multiple schedules.

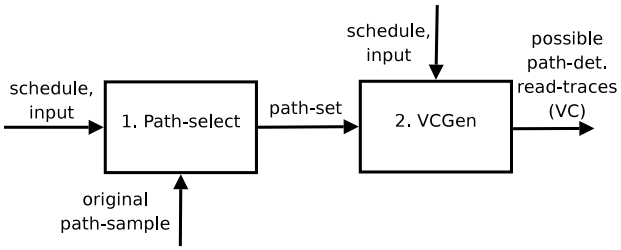

Figure 10: Read trace-set guidance up close. The goal is to consider only those read-traces that conform to the selected schedule-trace, input-trace, and a small set of potentially output-inducing paths.

enables path-select (Section 7.2.1) to identify a candidate path in just one invocation.

7. STAGES IN DETAIL

In this section, we detail the operation of the read-trace select and output-matching stages, used by both SI-DRI and QI-DRI. But first, we provide some definitions of terms used through the section.

7.1 Definitions

The *path* of a multi-threaded program is the sequence of paths of all threads, in ascending thread index order. The path of a thread is the sequence of instructions executed by the thread. The value l of the i-th number in the sequence denotes that the i-th instruction executed by the thread was the instruction at location l in the program. For example, path $((1,2),(1,3))$ denotes the fact that there are two threads, where thread 1 first executes instruction at location 1 and then the one at location 2, while thread 1 executes first the instruction at location 1, and then the one at location 3.

7.2 Read-Trace Set Guidance

The idea behind read trace-set guidance is to reduce the search to *path-deterministic read-traces* only, i.e., read-traces that, in addition to conforming to the selected schedule and input, force the program to take the original path. Searching this set is sufficient to yield an output-deterministic run because the original run is trivially part of this set.

The key challenge of the read trace-set guidance is that the original path is not known in its entirety; the SI-DRI query contains only a sample of the original path. To address this challenge we conservatively broaden the read trace-set search space. Specifically, we search the set of *possible path-deterministic trace-sets*–a superset of path-deterministic runs. While searching this superset is not as efficient as searching the set of path-deterministic runs, it is more efficient than searching the set of all paths.

Figure 10 shows the two stages of read trace-set guidance. The first stage, *path-select*, computes a path-set in which at least one member is the original path. It performs this computation using the original input, lock-order, and path-sample. We describe how path-select performs the computation in Section 7.2.1. The second stage, *VCGen*, computes a logical formula that represents the set of possible path-deterministic, memory model conformant read-traces based on the previously selected path-set, schedule, and input. In practice, the time complexity of *VCGen* is linear in the number of paths.

7.2.1 Path select

Path-selection generates a set of paths that consists only of paths that are *path-template conformant*. We say that a path is path-template conformant if it can be generated from some run of the program along the (1) input-trace, (2) lock-order sequence, and (3) path-sample–collectively called the *path-template*.

The set of path-template conformant paths is a valid path-set, since the original path is a member. Indeed, the original path can be obtained in some run of the program along the original input and lock-ordering. The set of path-template conformant paths is relatively small, so the resulting path-set will be also small. Specifically, the size of the set is exponential only in the number of data-races, as opposed to the set of all paths. In the special case of data-race free execution, for example, the set has only one path – the original path. Without data-races, inputs and lock-order completely determine the program path.

To obtain path-template conformant paths, the path-select stage repeatedly invokes PATH-SELECT–the path-selection algorithm in Figure 1. PATH-SELECT explores a super-set of all path-template conformant paths and returns the

Algorithm 1 PATH-SELECT(T, P). Returns a path-template conformant path (i.e., a path that may be that of the original run).

Require: Path template $T = (I, L, S)$, where I is a sequence of input bytes, L is a partial ordering of instructions, and S is a path sample, all from the original run
Require: Path P – initially empty
Ensure: Path C is path-template conformant
  P', conformant = IS-CONFORMANT-RUN(I, L, S, P)
  if conformant **then**
    return P'
  else
    for all unvisited $(t, c) \in$ RTB-ORACLE(I, L, S, P') **do**
      $C =$ PATH-SELECT$(T,$ FLIP-BRANCH$(P', (t, c)))$
      if $C \neq$ NIL **then**
        return C
    return NIL

first path-template conformant path it finds that already has not been returned in a previous invocation. If all conformant paths have been returned then the algorithm simply returns NIL. Our results in Section 10 show that, in practice, PATH-SELECT often identifies the original path after just one invocation.

To identify a path-template conformant path, PATH-SELECT runs the program on the original run's input, lock-order, and candidate path, and checks it for path-template conformance. The running and checking is done by the IS-CONFORMANT-RUN subroutine, which returns template-conformant path and true if the run happens to pass the conformance check. If a run does not pass the conformance check, for example because it did not match the path-sample, then it immediately terminates the run and returns false. As our results in Section 10 show, the latter is likely to happen if the run did not follow the original path.

If a run does not pass the conformance check, PATH-SELECT backtracks to the most-recent *race-tainted branch* (RTB), in depth-first search style, and forces the run down the previously unexplored branch (done by FLIP-BRANCH). The motivation for backtracking to an RTB is simple: the idea is that a branch influenced by a race may evaluate either way, depending on the outcome of the race influencing it. Race-tainted indirect branches (e.g., due to pointers and array-accesses) are trickier to handle efficiently because at the extreme they may branch to an arbitrary memory location. In this paper, we ignore indirect branches. However, they are rare in practice – we have never encountered them in our experiments.

To identify RTBs, PATH-SELECT invokes an RTB-ORACLE. The oracle returns the set of static branches that may have been affected by race-outcomes in some run of the program along the given program input, lock-order, and path. In PATH-SELECT, this branch super-set is denoted by a set of (t, c)-tuples, where t and c are the thread index and instruction count, respectively, of an RTB. Briefly, the RTB-ORACLE works in two stages to identify RTBs. In the first stage, it invokes a race-oracle to identify a set of racing accesses along the given input, lock-order, and path. In the second stage, it performs a conservative taint-flow analysis of all runs along the given input, lock-order, and path. We provide a more detailed treatment of the RTB-ORACLE in [1],

7.2.2 Verification Condition Generation

The goal of VCGen is to produce a verification condition (VC) [**?**], a logical formula that, for our purposes, represents the set of all read-traces that (1) are memory model M consistent with respect to the selected schedule and input trace and (2) make the program take one of the paths given in the path-set. Symbolic variables of this logical formula represent read-values of instructions executed along each path. Constraints within the formula limit the values that each symbolic read-value may take on. The set of all read-traces represented by this formula, then, is the set of all constraint-satisfying assignments for symbolic variables.

VCGen generates a VC by symbolically executing [3] the program along each path in the path-set given to VCGen. For each path, the symbolic execution produces a set of constraints that collectively describe the set of candidate read-traces for that path. The VC, then, is simply the disjunction of the constraints produced for each path in the path-set. Further details of VC generation and symbolic execution, both well-understood, may be found here [?,3].

The key challenge in VCGen is that of generating a VC under the LOR-consistency model. Under LOR-consistency, a racing read may return one of many values. Hence, VCGen can generate complete constraints only if it knows (1) which reads may race and (2) what values those racing reads may return. To address this challenge, VCGen relies on a race oracle. We describe the oracle in Section 8, but here it suffices to know that it will return the set of all possible read-values for instructions that may race in some execution along a given schedule, input, and path. Upon encountering a racing read, VCGen constrains the corresponding symbolic read-value to the read-values reported by the oracle.

7.3 Output Matching

To determine if a selected schedule, input, and read-trace set is output-deterministic, we could just run the program on the schedule and input trace for each read-trace in the set, and then check the outputs. But this would be slow. Instead, the output matching stage leverages a constraint solver to conceptually perform this running and checking of outputs.

More specifically, output matching works in two steps. In the first step, we augment the verification condition (VC), a set of constraints, generated in the previous phase (see Section 7.2.2) with constraints that further limit the set of read-traces to those that produce the original output.

In the second step, we dispatch the augmented VC to an SMT solver [3]–an intelligent constraint solver that in many cases can find a satisfying assignment to a given logical formula in polynomial time. If there is a satisfying

solution, then the SMT solver reports one possible assignment of read values (i.e., produces a read-trace). The read-trace, in conjunction with the selected schedule and input trace, form an output-deterministic run.

8. RACE ORACLE

SI-DRI and QI-DRI invoke a race-oracle in the process of generating a verification condition (Section 7.2.2). The goal of the race-oracle is to compute $Races(E)$ – the set of memory access instructions that race in some execution in E, where E is the set of executions that conform to a given input, lock-order, and path-set. In the following paragraphs, we describe how ODR computes $Races(E)$ using a race-detector.

8.1 Requirements

Our race-detector must meet several requirements, listed below in decreasing order of their importance.

1. *Soundness.* The detector must report all races in $Races(E)$. If it does not, then PATH-SELECT (Section 7.2.1) will miss backtracking points, in turn rendering path-selection unable to find a valid path-set. Moreover, missed races will break race-value relaxation; since it will not know what data should be represented symbolically, it may not be able to find an output-deterministic schedule.
2. *Precision.* We say that a race-detector is precise if it reports only those races $Races(E)$. Imprecision has two consequences. The first is performance – PATH-SELECT may have to explore more paths before converging. The second is a loss of value-consistency – SI-DRI's consistency relaxation will be applied even to accesses that do not race.
3. *Compatibility.* The race-detector must work on all Linux/x86 binaries, even those that come without source-code. This is needed to meet ODR's goal of replaying arbitrary programs.

8.2 Key Challenge

The key challenge in race-detection is to determine whether two static memory accesses may refer to the same location. Fundamentally, this boils down to the problem of figuring out what locations a memory access may refer to – the well-known points-to problem. For direct accesses, this task is straightforward. The address is hard-coded into the access instructions, and so the referenced location can be determined by instruction inspection. However, if the access is indirect, due to pointers or array accesses, then the location dereferenced can depend on up to four unknowns: program inputs, path, lock-order, and data-race outcomes. There are an exponential number of possibilities for each of these unknowns. Hence an exhaustive search of all possibilities is infeasible.

While a variety of race-detectors have addressed the points-to problem, we were unable to find one that meets all of our requirements. For instance, several static detectors meet our soundness criteria. However, since these detectors are geared toward finding races in all possible executions–rather than for those just in E–they require access to source code to produce precise points-to sets. On the other hand, several dynamic detectors can work at the binary-level and provide precise results, since indirect references are easily computed from execution state. Unfortunately, their results do not meet our soundness criteria since they report races for only one execution in E.

8.3 Approach

We have built a static race detector that produces sound and reasonably-precise results at the binary-level. The heart of the detector is a points-to analysis that relies on the following insight: for realistic programs, the references of most indirect accesses in the set E are a function solely of the program input, lock-order, and path; they are rarely influenced by data-races. Fortunately, SI-DRI provides us with the requisite input, lock-order, and path, hence enabling us to compute precise points-to sets despite the lack of source-code.

The key challenge of our approach is to compute complete points-to sets for those indirect references that do depend on data-race outcomes. The points-to set of an access instruction is complete if it contains all references made by the instruction in some execution in E. This definition suggests one possible approach to computing complete points-to sets – consider all possible race outcomes for each execution in E. While this would provide us with complete and precise points-to sets, it is computationally intractable to evaluate all executions in E let along race outcomes in each each execution.

Our approach to this challenge is to be very conservative. That is, we assume that race-influenced indirect accesses may reference any memory location. The benefit of this approach is a complete points-to set for every indirect access, and hence sound race-detection. The drawback, in theory, is that the points-to sets would contain many memory locations that would not be referenced in an execution in E, hence resulting in many false races. In practice, we've found that this drawback rarely applies, namely because data-races rarely (never in our experience) influence references. Our results in Section 10 support this observation.

9. IMPLEMENTATION

ODR consists of approximately 100,000 lines of C code and 2,000 lines of x86 assembly. The replay core accounts for 45% of the code base. The other code comes from Catchconv [13] and LibVEX [15], an open-source symbolic execution tool and binary translator, respectively. We encountered many challenges when developing ODR. Here we describe a selection of those challenges most relevant to our inference method.

9.1 Tracing inputs and outputs

To capture inputs and output, we employ a kernel module – it generates a signal on every system call and non-deterministic x86 instruction (e.g., RDTSC, IN, etc.) that ODR then catches and handles. DMA is an important I/O source, but we ignore it in the current implementation.

Achieving completeness is the main challenge in user-level I/O interception. The user-kernel interface is large – we had to implement about 200 system calls before ODR logged and replayed sophisticated applications like the Java Virtual Machine. Some of these system calls, such as `sys_gettimeofday()`, are easy to handle – ODR just records their return values. But many others such as `sys_kill()`, `sys_clone()`, `sys_futex()`, and `sys_open()`, `sys_mmap()` require more extensive emulation work—largely to ensure deterministic signal delivery, task creation and synchronization, task/file ids, file/socket access, and memory management, respectively.

9.2 Tracing lock-order

Several of our search-space reductions rely on the original run's lock-order. We intercept locks using binary-patching techniques. Specifically, we dynamically replace instructions that acquire and release the system bus lock with calls into tracing code. The tracing code, once invoked, emulates the memory operation. It also logs the value of a per-thread Lamport clock for the acquire operation preceding the memory operation and increments the Lamport clock for the subsequent release operation. We could've intercepted lock-order at the library level (e.g., by instrumenting `pthread_mutex_lock()`), but then we would miss inlined and custom synchronization routines that are common in `libc`, which in turn would result in a larger search space.

9.3 Tracing branches

Branch tracing is employed by QI-DRI, as discussed in Section 6. We capture branches in software using the Pin binary instrumentation tool [12]. Software binary translation incurs some overhead, but it's a lot faster than the alternatives (e.g., LibVEX or x86 hardware branch tracing [10]). To obtain low logging overhead, we employ an idealized, software-only 2-level/BTB branch predictor [8] to compress the branch trace on the fly. Since this idealized predictor is deterministic given the same branch history, compression is achieved by logging only the branch mispredictions. The number of mispredictions for this class of well-studied predictors is known to be low [8].

9.4 Symbolic execution

There are many symbolic execution tools to choose from, but we needed one that worked at user-level and supports arbitrary Linux/x86 programs. We chose Catchconv [13] – a user-mode, instruction-level symbolic execution tool. Though designed for test-case generation, Catchconv's constraint generation core makes few assumptions about the target program and largely suits our purposes.

Rather than generate constraints directly from x86, Catchconv employs LibVEX [15] to first translate x86 instructions into a RISC-like intermediate language, and then generates constraints from this intermediate language. This intermediate language abstracts-away the complexities of the x86 instruction set and thus eases complete and correct constraint generation. Catchconv also implements several optimizations to reduce formula size.

10. PERFORMANCE

In this section, we evaluate ODRunder two configurations– one in which it uses SI-DRI and the other in which it uses QI-DRI. We begin with our experimental setup and then give results for each major ODR phase: record, inference, and replay. In summary, we found that when using SI-DRI, ODR incurs low recording overhead (less than 1.6x on average), but impractically high inference times. For instance, two applications in our suite took more than 24 hours during the inference phase. In contrast, ODR with QI-DRI incurs significantly higher recording overhead (between 3.5x and 5.5x slowdown of the original run). But the inference phase finished within 24 hours for all applications. Thus, SI-DRI and QI-DRI represent opposites in the record-inference axis of the DRI tradeoff space.

10.1 Setup

We evaluate seven parallel applications: *radix*, *lu*, and *water-spatial* from the Splash 2 suite, the Apache webserver (*apache*), the Mysql database server (*mysql*), the Hotspot Java Virtual Machine running the Tomcat webserver (*java*), and a parallel build of the Linux kernel (*make-j2*). We do not give results for the entire Splash 2 suite because some (e.g, FMM) generate floating point constraints, which our current implementation does not support (see Section 11). Henceforth, we refer to the Splash 2 apps as *SP2 apps* and the others as *systems apps*.

All inputs were selected such that the program ran for just 2 seconds. This ensured that inference experiments were timely. *apache* and *java* were run with a web-crawler that downloads files at 100KBps using 8 concurrent client connections. *mysql* was run with a client that queries at 100KBps, also using 8 concurrent client connections. The Linux build was performed with two concurrent jobs (*make-j2*).

Our experimental procedure consisted of a warmup run followed by 6 trials. We report the average numbers of these 6 trials. The standard deviation of the trials was within three percent. All experiments were conducted on a 2-core Dell Dimension workstation with a Pentium D processor running at 2.0GHz and 2GB of RAM. The OS used was Debian 5 with a 2.6.29 Linux kernel with minor patches to support ODR's interpositioning hooks.

10.2 SI-DRI Record Mode

Figure 11 shows the record-mode slowdowns when using SI-DRI. The slowdown is broken down into five parts: (1) *Execution*, the cost of executing the application without any tracing or interpositioning; (2) *Path-sample trace*, the cost of intercepting and writing path-samples (see Section 5) to the log file; (3) *I/O trace*, the cost of intercepting and writing both program input and output to a log file; (4) *Lock trace*, the cost of intercepting bus-lock instructions and writing logical clocks to the log file at each such instruction; (5) *Emulation*, the cost of emulating some syscalls (see Section 9.1 for why).

As shown in Figure 11, the record mode causes a slowdown of 1.6x on average. ODR outperforms software-only multiprocessor replay systems on key benchmarks, and is com-

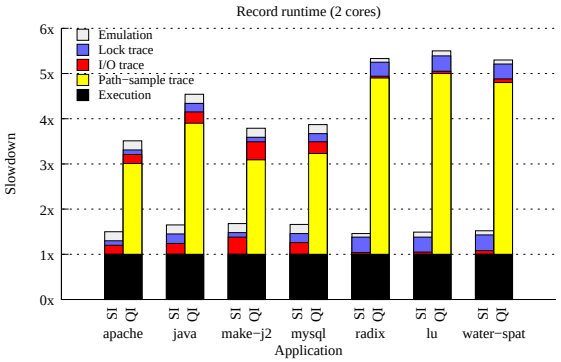

Figure 11: ODR's record-mode runtimes, normalized with native execution time, for both SI-DRI and QI-DRI.

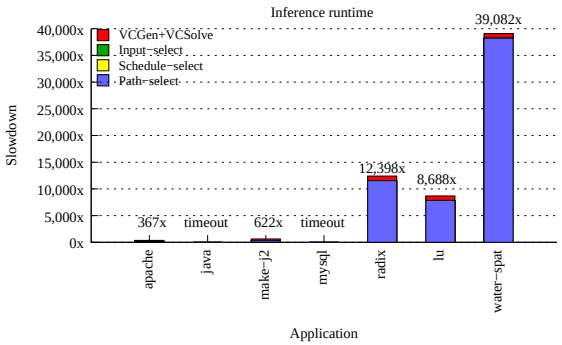

Figure 12: Inference runtime, normalized with native execution time, for SI-DRI. Programs that did not finish in the first 24 hours are denoted by "timeout".

parable on several others. For instance, ODR outperforms SMP-Revirt on *make-j2* (by 0.3x) and *radix* (by 0.1x) for the 2-processor case. ODR does better because these applications exhibit many false-sharing. False-sharing induces frequent CREW faults on SMP-ReVirt [4], but not on ODR, since it ignores recording races. In addition, ODR does not trace into the Linux kernel like SMP-ReVirt, thus avoiding the sharing caused by the kernel. ODR approaches RecPlay's performance (within 0.4x) for the SP2 applications. This is because, with the exception of outputs and sample points, ODR traces roughly the same data as RecPlay (though RecPlay captures lock order at the library level). SP2 applications are not I/O intensive, so the fact that ODR records the outputs does not have a significant impact.

ODR does not always outperform existing multiprocessor replay systems. For instance, SMP-ReVirt and RecPlay achieve near-native (1x) performance on several SP2 applications (e.g., LU), while ODR incurs an average overhead of 0.5x of SP2 apps. As Figure 11 shows, a bulk of this overhead is due to lock-tracing, which SMP-ReVirt does not do. And while RecPlay does trace lock order, it does so by instrumenting lock routines (in `libpthreads`) rather than all locked instructions. Intercepting lock order at the instruction level is particularly damaging for SP2 apps because they frequently invoke `libpthreads` routines, which in turn issue many locked-instructions. One might expect the cost of instruction-level lock tracing to be even higher, but in practice it is small because libpthread routines do not busy-wait under lock contention – they await notification in the kernel via a `sys_futex`. Nevertheless, these results suggest that library-level lock tracing (as done in RecPlay) might provide better results.

Compared with hardware-based systems, ODR performs slightly worse, especially on systems benchmarks. For example, CapoOne achieves a 1.1x and 1.6x slowdown for *apache* and *make-j2*, respectively, while ODR achieves a 1.6x and 1.8x slowdown. Based on the breakdown in Figure 11, we attribute this slowdown to two bottlenecks. The first, not surprisingly, is output-tracing. The effect of output tracing is particularly visible in the case of *apache* and *make-j2*, which transfer large volumes of data. The second bottleneck is the emulation. As discussed in Section 9.1, triggering a signal on each syscall, and emulating task and memory management at user-level can be costly.

10.3 SI-DRI Inference Mode

Figure 12 gives inference slowdown for each application. The slowdown is broken down into 4 major SI-DRI components: schedule-select (Section 5.3), input-select (Section 5.3), path-select (Section 7.2.1, part of read-trace guidance), and VCGen+VCSolve (Section 7.2.2 and 7.3, part of read-trace guidance). Path-select and VCGen+VCSolve account for the vast majority of the inference time – since we do not know the path or read-trace, we have to search for them. In contrast, schedule-select and input-select are instantaneous due to consistency relaxation (Section 5.4) and query-directed search (Section 5.3), respectively.

Overall, SI-DRI's inference time is impractically long. Two applications, *java* and *mysql*, do not converge within the 24 hour timeout period, and those that do converge achieve an average slowdown of 12,232x. As the breakdown in Figure 12 shows, there are two bottlenecks. The primary bottleneck is path-selection, taking up an average 75% percent of inference time. In the case of *java* and *mysql*, path-selection takes so long that it prevents ODR from proceeding to the later stages (i.e., VCGen+VCSolve) within the timeout period (24 hours). The secondary bottleneck is VCGen+VCSolve, taking up the remaining average 25% percent of inference time. We investigate each bottleneck in the following sections.

10.3.1 Path-selection

We expected path-selection to be slow primarily because we expected PATH-SELECT, SI-DRI's path-selection algorithm (Section 7.2.1), to backtrack (i.e., recursively invoke itself) an exponential number of times. We also expected that the cost of each backtrack played a strong secondary role. To verify this hypothesis, we counted the number of backtracks and measured the average cost of a backtracking operation. The results, shown in Figure 13, contradict our hypothesis. That is, the number of backtracks for most apps, with the exception of *java* and *mysql*, is low, hence making the cost of each backtrack operation the dominant factor.

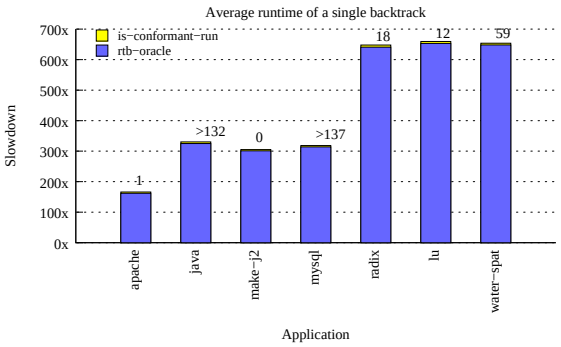

Figure 13: The average runtime, normalized with native runtime, of one backtrack performed by SI-DRI during PATH-SELECT, broken down into its two stages. The total number of backtracks is given at the top of each bar. For apps that timeout, this number is just a lower-bound.

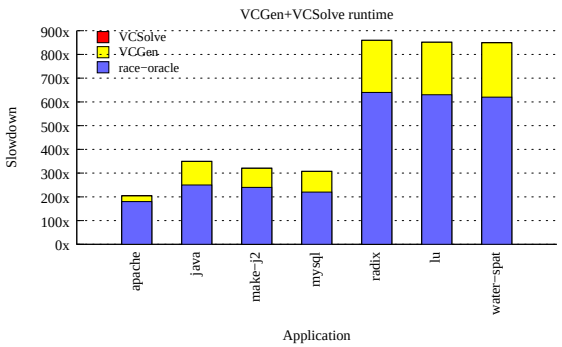

Figure 14: Runtime of the VCGen and VCSolve phases, normalized with native runtime, broken down into its three stages.

There are two reasons for the small number of backtracks. The first, specific to *make-j2* and *apache*, is that there are a small number of dynamic races and consequently a small number of race-tainted branches (RTBs, see Section 7.2.1). *make-j2*, for instance, does not have any shared-memory races at user-level, and hence no RTBs to backtrack to. Most sharing in *make-j2* is done via file-system syscalls, the results of which we log, rather than through user-level shared memory. Apache, in contrast, does have races and RTBs, but a very small number of them. Our runs had between 1 and 2 dynamic races, each of which tainted only 1 branch. Thus in the worst case, we would have to backtrack 4 times. The actual number of backtracks is smaller because PATH-SELECT guesses some of these RTBs correctly on the first try.

The second reason for the small number of backtracks is specific to the SP2 apps. These apps did well despite having a large number of RTBs (an average of 30) because PATH-SELECT was able to resolve all their divergences with just one backtrack, hence making forward-progress without exponential search. Only one backtrack was necessary because, in the code paths taken in our runs, there is a sampling point after every RTB. So if PATH-SELECT chooses an RTB outcome that deviates from the original path, then the following sampling point will be missed, hence triggering an immediate backtrack to the original path.

Unlike the majority of apps in our suite, *java* and *mysql* have a significantly larger number of backtracks. The large number stems from code fragments with few sampling points between RTBs. In those cases, we end up with too many instructions between successive sampling points, which in turn require a large number of backtracks to resolve. Consider a loop that contains a race, and no sampling points. Thus, the earliest time, we can detect a divergence is at the first sampling point after the loop finishes. Now assume that the loop executes 1,000 times, but the divergence is caused at the 500-th iteration. In this case, we need to backtrack 500 times to identify the cause of the divergence.

Overall, our results indicate that reducing backtracking costs is key to attaining practical inference times – 1000 backtracks may be tolerable if each incurs say at most a 2x slowdown. To identify opportunities for improvement, we broke down the average backtracking slowdown into its two major parts, shown in Figure 13. The first part is the cost of invoking the RTB-ORACLE (see Section 7.2.1), needed to intelligently identify backtracking points. The second is the cost of invoking IS-CONFORMANT-RUN (see Section 7.2.1), needed to verify that a selected path is path-template conformant. The results show that the cost of a backtrack is dominated by the invocation of the RTB-ORACLE as expected. We expect the RTB-ORACLE to be expensive because it involves race detection and taint-flow analyses (see Section 7.2.1) over the entire path up till the point of non-conformance. In theory, the RTB-ORACLE need not be run over the entire failed path on every backtrack, but we leave that to future work.

10.3.2 VCGen+VCSolve

We expected VCGen+VCSolve to be slow, especially since VCSolve is an NP-complete procedure for worst-case computations (e.g., hash-functions). To verify this hypothesis, we broke down the phase's runtime into three parts, as show in Figure 14. The first part is the cost of invoking the race-oracle (Section 8), needed to identify which accesses may race (an important part of VCGen). The second part is VCGen, needed to represent a set of candidate read-traces. And the third part is VCSolve, needed to obtain a particular read-trace from the candidate set, which involves invoking a constraint solver. The breakdown contradicted our hypothesis in that most of the inference is spent in the race-oracle (a polynomial time procedure 8), not VCGen or VCSolve.

VCGen and VCSolve are fast for two reasons. The first is that our VC generator (see Section 7.2.2) generates VCs only for those instructions influenced (i.e., tainted) by racing accesses. If the number of influenced instructions is small, then the resulting formula will be small. Our results indicate that, for the programs in our suite, races have limited influence on instructions executed – the average size of a formula is 1562 constraints. The second reason is that, of those constraints that were generated, all involved only linear twos-complement arithmetic. Such

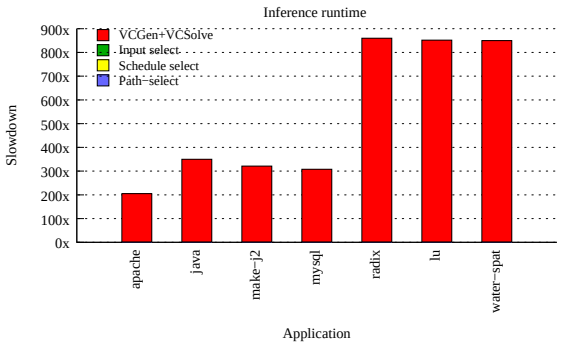

Figure 15: Inference runtime overheads for QI-DRI. All apps finish within the first 24 hours.

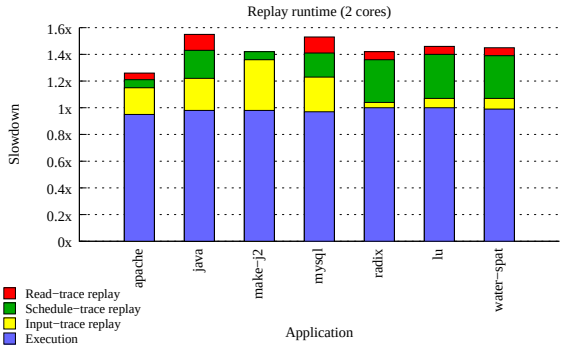

Figure 16: Two-processor replay-mode runtime overheads normalized with native execution time.

constraints are desirable because the constraint solver we use can solve them in polynomial time [5]. We did not encounter any races that influenced the inputs of hash functions or other hard non-linear computations.

The penalty for efficient constraint generation and solving is expensive race-detection. Our race-oracle is slow because it performs set-intersection of all accesses made is the given path. Because a set may contain millions of accesses, the process is inherently slow, even with our $O(n)$ set-intersection algorithm.

10.4 QI-DRI Record and Inference Modes

As we have shown so far, SI-DRI leads to a low recording overhead, but it's inference time is prohibitive. In this section, we evaluate a variant of SI-DRI, QI-DRI, which trades recording overhead for improved inference times. In particular, QI-DRI relies on recording branches during the original run, as explained in Section 9.3. Recoding branches removes the need to invoke PATH-SELECT, the key bottleneck behind the timeouts in SI-DRI. Hence the improvements in the inference time.

Figure 11 shows QI-DRI's slowdown factors for recording, normalized with native execution times. As expected, recording branches significantly increases the overhead, from 1.6x to 4.5x on average. While this overhead is still 4 times less than the average overhead of iDNA [2] [1], it is greater than other software-only approaches, for some applications. Radix, for example, takes 3 times longer to record on ODRwhen using QI-DRI than with SMP-ReVirt [4].

Figure 15 shows the inference time for QI-DRI normalized with native execution times. As expected, QI-DRI achieves much lower inference times than SI-DRI (see Figure 12). The improvements are due to the fact that QI-DRI does not need to spend time in the path-select substage of read-trace guidance–the most expensive part of SI-DRI–as the original path of each thread has been already recorded. In the absence of path-selection overhead, the VCGen+VCSolve stage dominates. As explained in Section 10.3.2 and illustrated in Figure 14, the largest percentage of time in the VCGen+VCSolve stage is spent in the race-oracle (see Section 8).

[1]This is because our most expensive operation, obtaining a branch trace, is not quite as expensive as intercepting and logging memory accesses

10.5 Replay Mode

Figure 16 shows replay runtime, normalized with native runtime, broken down into three parts. The first is *Input+event replay*, which is the cost of intercepting and reading from the log file for input syscalls and event delivery points. The second is *Lock-order replay*, which is the cost of intercepting lock instructions and replaying the lock order, including time spent waiting to meet logical-clock dependencies. The third is *Race-value replay*, primarily the cost of detecting execution points at which race-values should be substituted. As with other replay systems, native execution time in replay mode is smaller for I/O intensive apps because ODR skips the time originally spent waiting for I/O [11].

The key result here is that replay slowdown is low across the board independent of inference time. This is expected. Once race-values are inferred, they are merely plugged-in to future re-executions. Surprisingly, race-value substitution is not a bottleneck, since there are very few race-values to substitute. *make-j2*, for instance, has no races and hence need not perform any substitutions. Replay speed is not near-native largely due to the cost of intercepting and replaying lock-instructions. Programs with a high locking rate (e.g., *java*, SP2 apps), suffer the most. We hope to improve these costs in a future implementation, perhaps by moving to a library-level lock interception scheme.

11. LIMITATIONS

ODR has several limitations that warrant further research. We provide a sampling of these limitations here.

Unsupported constraints. For inference to work, the constraint solver must be able to find a satisfiable solution for every generated formula. In reality, constraint solvers have hard and soft limits on the kinds of constraints they can solve. For example, no solver can invert hash functions in a feasible amount of time, and STP cannot handle floating-point arithmetic.

	Data races	Multiple CPUs	Efficient and scalable recording	Software-only	Determinism
Jockey [18]	Yes	No	Yes	Yes	Value
RecPlay [17]	No	Yes	Yes	Yes	Value
SMP-ReVirt [4]	Yes	Yes	No	Yes	Value
iDNA [2]	Yes	Yes	No	Yes	Value
DeLorean [14]	Yes	Yes	Yes	No	Value
ODR	Yes	Yes	Yes	Yes	Output

Table 1: Summary of key related work.

Fortunately, all of the constraints we've seen have been limited to feasible integer operations. Nevertheless, we are exploring ways to deal with the eventuality of unsupported constraints. One approach is to not generate any constraints for unsupported operations, and instead make the targets of those operations symbolic. This in effect treats unsupported instructions as blackbox functions that we can simply skip during replay.

Symbolic memory references. Our constraint generator assumes that races do not influence pointers or array indices. This assumption holds for the executions we've looked at, but may not for others. Catchconv and STP do support symbolic references, but the current implementation is inefficient – it models memory as a very large array and generates an array update constraint for each memory access, thereby producing massive formulas that take eons to solve. One possible optimization is to generate updates only when we detect that a reference is influenced by a race (e.g., using taint-flow).

Inference time. The inference phase is admittedly not for the impatient programmer, to put it mildly. The main bottleneck, happens-before race-detection, can be improved in many ways. An algorithmic optimization would be to ignore accesses to non-shared pages. This can be detected using the MMU, but to start, we can ignore accesses to the stack, which account for a large number of accesses in most applications. An implementation optimization would be to enable LibVEX's optimizer; it is currently disabled to workaround a bug we inadvertently introduced into the library.

12. RELATED WORK

Table 1 compares ODR with other replay systems along key dimensions.

Many replay systems record race outcomes either by recording memory access content or ordering, but they either do not support multiprocessors [18] or incur huge slowdowns [2]. Systems such as RecPlay [17] and more recently R2 [7] can record efficiently on multiprocessors, but assume data-race freedom. ODR provides efficient recording and can reliably replay races, but it does not record race outcomes – it computes them.

Much recent work has focused on harnessing hardware assistance for efficient recording of races. Such systems can record very efficiently . But the hardware they rely on can be unconventional and in any case exists only in simulation. ODR can be used today and its core techniques (I/O tracing, and inference) can be ported to a variety of commodity architectures.

The idea of relaxing determinism is as old as deterministic replay technology. Indeed, all existing systems strive for value-determinism–a relaxed form of determinism, as pointed out in Section 3. By striving for output-determinism, ODR relaxes even further. Relaxed determinism was recently re-discovered in the Replicant system [16], but in the context of redundant execution systems. Their techniques are, however, inapplicable to the output-failure replay problem because they assume access to execution replicas in order to tolerate divergences.

13. CONCLUSION

We have designed and built ODR, a software-only, replay system for multiprocessor applications. ODR achieves low-overhead recording of multiprocessor runs by relaxing its determinism requirements–it generates an execution that exhibits the same outputs as the original rather than an identical replica. This relaxation, combined with efficient search, enables ODR to circumvent the problem of reproducing data races. The result is record-efficient output-deterministic replay of real applications.

We have many plans to improve ODR. Among them is a more comprehensive study of output-determinism and the limits of its power. We also hope to get more applications running on ODR, so that we may better understand the limits of our inference technique. And of course, we aim to remove the limitations listed in Section 11. Looking forward, output-deterministic replay of data-centers seems promising.

14. REFERENCES

[1] G. Altekar and I. Stoica. Sound race-detection and taint flow for output-deterministic replay. Technical Report To Be Submitted, EECS Department, University of California, Berkeley, 2007.

[2] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.

[4] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2008. ACM.

[5] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[6] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. pages 289–300.

[7] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for

record and replay. In R. Draves and R. van Renesse, editors, *OSDI*, pages 193–208. USENIX Association, 2008.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann, September 2006.

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

[10] Intel. *Intel 64 and IA-32 Architectures Reference Manual*, November 2008.

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 1–15. USENIX, 2005.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 190–200, New York, NY, USA, June 2005. ACM Press.

[13] D. A. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, 2007.

[14] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[16] J. Pool, I. S. K. Wong, and D. Lie. Relaxed determinism: making redundant execution on multiprocessors practical. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[17] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

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