ODR: Output-Deterministic Replay for Multicore Debugging

Gautam Altekar and Ion Stoica
UC Berkeley
{galtekar, istoica}@cs.berkeley.edu

ABSTRACT

Reproducing bugs is hard. Deterministic replay systems address this problem by providing a high-fidelity replica of an original program run 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 altogether. The result is a system that replays real multiprocessor applications, such as Apache, MySQL, and the Java Virtual Machine, and provides low record-mode overhead.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Reliability, Design, Performance

Keywords

Deterministic replay, Multicore, Debugging, Inference

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; 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 [3, 7, 8, 13, 22].

The ideal record-replay system has three key properties. First, it produces a high-fidelity replica of the original program run, 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 parallel applications running on commodity multi-core machines. However, despite much research, the ideal replay system still remains out of reach.

A major 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 [3, 5]. Other systems achieve low record overhead, but rely on non-standard hardware [18]. Still others assume data-race freedom, but fail to reliably reproduce failures [21].

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 run, though sufficient, is *not* necessary for replay-debugging. Instead, it suffices to produce *any* run that exhibits the same output, even if that run differs from the original. This observation permits `ODR` to relax its fidelity guarantees and, in so doing, enables it to circumvent the problem of reproducing and hence recording data-race outcomes.

The key problem `ODR` must address is that of reproducing a failed run 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` in-

SOSP'09, October 11–14,2009, Big Sky, Montana, USA.

fers the data-race outcomes of an output-deterministic run. Once inferred, ODR substitutes these values in subsequent program runs. The result is output-deterministic replay.

To infer data-race outcomes, ODR uses a technique we term *Deterministic-Run Inference*, or DRI for short. DRI searches the space of runs for one that produces the same outputs as the original. An exhaustive search of the run space is intractable. But carefully selected clues recorded during the original run in conjunction with memory-consistency relaxations often enable ODR to home-in on an output-deterministic run in polynomial time.

We evaluate ODR on several sophisticated parallel applications, including Apache and the Splash 2 suite. Our results show that, although our Linux/x86 implementation incurs an average record-mode slowdown of only 1.6x, inference times can be impractically high for many programs. However, we also show that there is a tradeoff between recording overhead and inference time. For example, recording all branches slows down the original execution by an average 4.5x. But the additional information can decrease inference time by orders of magnitude. Overall, these results indicate that ODR is a promising approach to the problem of reproducing failures in multicore application runs.

2. THE PROBLEM

ODR addresses the *output-failure replay problem*. In short, 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. 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 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. But to be practical, the system must also 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.

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.

```
              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 failures 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.

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 run 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 [3] 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-race values. Data-races are often benign and intentionally introduced to improve performance. Sometimes they are inadvertent and result in software failures. Regardless of whether data-races are 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 [3]. They can be implemented entirely in software and can replay all output-failures, but incur high record-mode overheads (e.g., 5x slowdown [3]). 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 [5] or no data-races [21]. Finally, hardware-assisted systems can replay data-races at very low record-mode costs, but require non-commodity hardware [10, 17, 18].

(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 showcases a different determinism guarantee.

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 we evaluate in this paper.

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 run 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: it makes no guarantees about non-output properties of the original run. For instance, output determinism does not guarantee that the replay run will read and write the same values as the original. As an 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`. Moreover, output determinism does not guarantee that the replay run will take the same path as the original run.

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 altogether. 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 thread, consisting 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 program location iff its register parameter is non-zero. This simple program model assumes that hardware interrupts, if desired, are explicitly programmed as an input and a conditional branch to a handler, done at every instruction.

A *run* or *execution* of a program is a finite 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 thread's instruction sequence, one at a time and interleaved 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 thread.

A run's *determinant* is a triple that uniquely characterizes the run. This triple consists of a schedule-trace, input-trace, and a read-trace. A program *schedule-trace* is a finite sequence of thread identifiers that specifies the ordering in which instructions from different threads are interleaved (i.e., a total-ordering). An *input-trace* is a finite sequence of bytes consumed by input instructions. And a *read-trace* is a finite sequence of bytes returned by read instructions. For example, Figure 2(b) shows a run-trace with schedule-trace $(2, 2, 1, 2, 1, 1)$, input-trace (`DIE`), and read-trace (`DEAD`, 0), where $r0$ reads `DEAD`, and `*reaped` reads 0. Figure 2(c) shows another run-trace with the same schedule and 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 *instantiate* a run (a sequence of states), by (1) executing instructions in the interleaving specified by the schedule-trace, (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, *decomposition*, is straightforward.

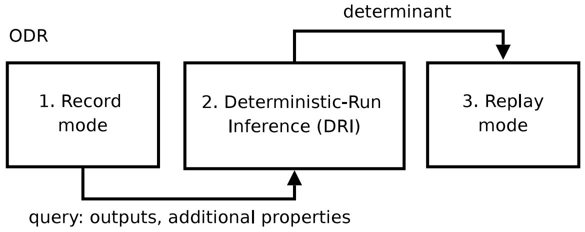
ODR
determinant
1. Record mode
2. Deterministic-Run Inference (DRI)
3. Replay mode
query: outputs, additional properties

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 instantiate future program runs.

We say that a run or determinant is *M-consistent* iff its read-trace is in the set of all possible read-traces for the run or determinant's schedule-trace, input-trace, and memory consistency model M. For example, the run in Figure 2(a) is strict-consistent 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-traces, 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 consistency 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 to quickly instantiate an output-deterministic run.

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 strict-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* toward runs that share the same properties as the original run. Figure 4(b) shows an example illustrating this idea at its extreme. 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 schedule, input, and read trace allows DRI to converge on an output-deterministic run after exploring just one run.

The second technique is to *relax the memory-consistency* of all runs in the run space. In general, a weaker consistency model permits more runs matching the original's output (than a stronger model), hence enabling DRI to find such a run with less effort.

To see the benefit, Figure 5 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 5(b), 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.

Although relaxing the consistency model reduces the number of search iterations, there is a drawback: it becomes harder for a 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 in the next section we identify our targets within it.

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. 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 memory-inconsistency, respectively.

The smallest and most desirable query-size is that of a query with just the original outputs. The largest and least desirable query-size is that of a query with all of a run's determinant–the schedule, input, and read trace. The latter results in constant-time search-complexity but carries a penalty of large record-mode slowdown. For instance, capturing a read-trace may result in a 5x penalty or worse on modern commodity machines [3].

(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 schedule, input, and read trace, all from the original run given in Figure 2(a). DRI converges in an exponential number of iterations for case (a), and in just one iteration for case (b) since the full determinant is provided.

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 points in the 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. Specifically, 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 9 indicate that polynomial complexity holds for several real-world applications. SI-DRI also targets a query-size that, while larger than a query with just the outputs, is still small enough to be useful for at least periodic production use. Finally, SI-DRI relaxes the memory consistency model of the inferred run from strict consistency to the hypothetical lock-order consistency, a slightly weaker model in which only the runs of data-race free programs are guaranteed to be strict-consistent.

The second design point we evaluate, Query-Intensive DRI, is almost identical to Search-Intensive DRI. The key difference is that QI-DRI calls for a query containing the original branch-trace of all threads–considerably more information than required by SI-DRI. Meeting this query requirement inflates recording overhead, but results in polynomial search-complexity, independent of application.

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.

(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 than strict consistency.

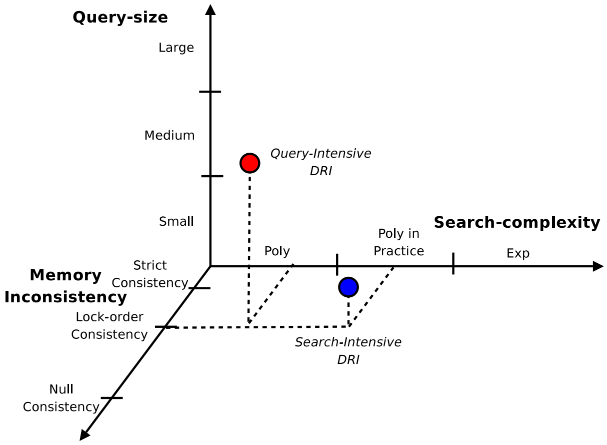

Figure 6: The design space for DRI. Exhaustive search is intractable, but providing more information in the query or relaxing consistency can make it tractable.

5.1 Overview

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

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.
- *lock-order*, a total ordering of lock instructions executed 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 executed 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 count, and l is the program location of that instruction. The path-sample includes

(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.

one such tuple for each input, output, and lock instruction executed in the original run.

Section 5.3 details how si-dri leverages this query information.

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 7 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 7(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 8, performs an exhaustive depth-first search on the space of strict-consistent determinants. Since the space may be infinite in size (due to program loops), si-dri searches only determinants of length n. The n denotes the length of the original run, which we assume is in the original output. The search is performed iteratively and in four steps.

In the first step, *schedule-select*, si-dri selects an n-length schedule-trace from the space of all n-length schedule-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 n-length 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 determinants 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-trace of length n that conforms to it. By considering only such schedule-traces, lock-order guidance avoids searching all schedules as done in the core algorithm.

The second guidance-method is *input-trace guidance*, used in the input-selection stage. Input guidance simply reproduces the original input-trace found in the query. It does not have much work to do because all of the input is given in the query. Input guidance 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 read-traces that are more likely to result in output-deterministic runs. 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 detailed in Section 6.2.

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 conforming to 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 7 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 7(b) are both (DEAD, 0)–a trivial rearrangement.

6. STAGES IN DETAIL

In this section, we detail the operation of the read-trace select and output-matching stages. But first, we provide definitions of terms used throughout the section.

6.1 Definitions

The *path* of a multi-threaded program is a sequence of thread-paths, one for each thread, in ascending thread index

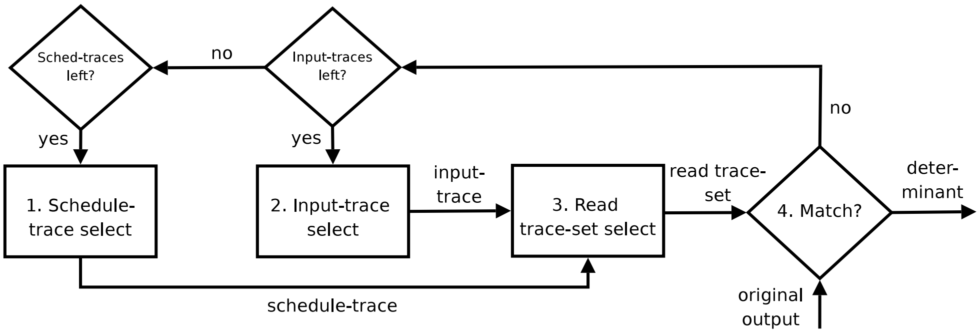

Figure 8: The core Search-Intensive DRI algorithm exhaustively searches the space of n-length, strict-consistent determinants for one that, when instantiated, outputs the same values as the original run. It makes use of only the original outputs in the query and consequently has exponential search-complexity.

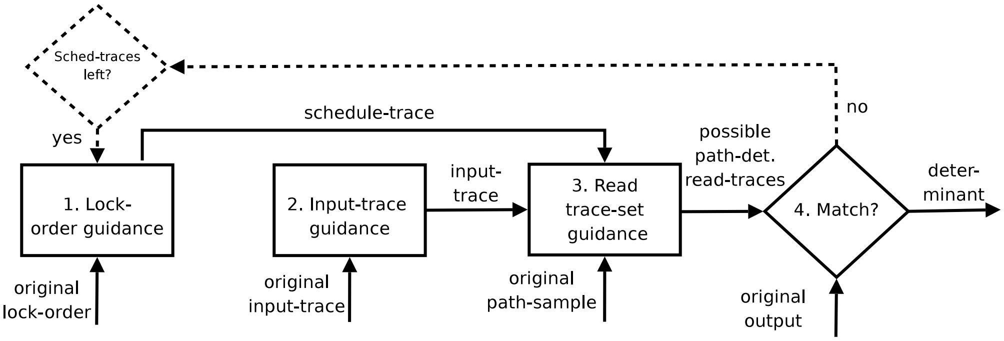

Figure 9: The Search-Intensive DRI algorithm with (dashed and solid) just query-guidance methods applied and (just solid) with both query-guidance and consistency relaxation applied. Consistency relaxation eliminates the need to search multiple schedules.

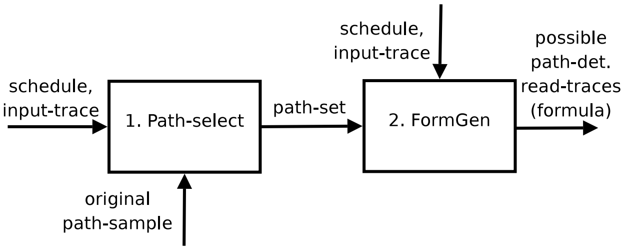

Figure 10: Read trace-set guidance up close. The goal is to consider only those read-traces that conform to the selected schedule-trace, input-trace, and a small set of potentially output-inducing paths.

order. The thread-path of a thread is a sequence of instruction locations executed by the thread. For example, path $((1, 2), (1, 3))$ denotes the fact that there are two threads, where thread 0 executes the instruction at location 1 first and then the one at location 2, while thread 1 executes the instruction at location 1 first, and then the one at location 3.

6.2 Read Trace-Set Guidance

The idea behind read trace-set guidance is to focus the search on *path-deterministic read-traces* – those read-traces that when instantiated along with the selected schedule and input trace, result in a run that takes 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 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*. When instantiated with the selected schedule and input trace, the possible trace-set yields a superset of path-deterministic runs. While searching this superset is not as efficient as searching the set of path-deterministic runs, it is in practice 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 set of paths in which at least one member is the original path, henceforth called a *path-set*. Path-select performs this computation using the provided schedule and input-trace, and original path-sample. We detail path-select's operation in Section 6.2.1. The second stage, *FormGen*, computes a logical formula that represents the set of possible path-deterministic read-traces that describe a LOR-consistent run, based on the previously selected path-set, schedule-trace, and input-trace.

6.2.1 Path Select

Path-selection generates a set 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 provided (1) schedule-trace, (2) input-trace, 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, under

LOR-consistency, the original path can be obtained in some run of the program along the provided input and schedule trace. The set of path-template conformant paths is relatively small, so the resulting path-set will also be small. Specifically, the size of the set is exponential only in the number of data-races. In the special case of data-race free execution, for example, the set has only one path – the original path. Without data-races, LOR-consistency guarantees that the input and schedule trace completely determine the program path.

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 an input-trace, $L$ is a schedule-trace, and $S$ is a path sample, all from the original run
Require: A candidate 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
```

To obtain path-template conformant paths, the path-select stage repeatedly invokes PATH-SELECT–the path-selection algorithm in Algorithm 1. PATH-SELECT explores a super-set of all path-template conformant paths and returns the first path-template conformant path it finds that has not already been returned in a previous invocation. If all conformant paths have been returned then the algorithm returns NIL. Our results in Section 9 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 given schedule-trace, input-trace, and a candidate path (if not empty), and checks it for path-template conformance. The running and checking is done by the IS-CONFORMANT-RUN subroutine, which returns a conformant path and *true* if the run passes the conformance check. If a run does not pass the conformance check, for example because it did not match the path-sample, then IS-CONFORMANT-RUN immediately terminates the run, and returns the failed path and *false*. As our results in Section 9 show, the latter is likely to happen soon after the run *diverges* (i.e., does not pass the conformance check) from the original path.

If a run diverges, then 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: an RTB may evaluate either way, depending on the outcome of the race influencing it, and hence is likely to be the cause of divergence. Race-tainted indirect branches (e.g., due to function pointers) are trickier to handle efficiently because, at the extreme, they may branch to an arbitrary memory location. In this paper, we ignore race-tainted indirect branches. Note, however, that 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 schedule-trace, input-trace, and failed 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 (Section 6.2.3) to identify a set of racing accesses along the given schedule and input trace, and failed path. In the second stage, it performs a conservative taint-flow analysis of all runs along the given schedule and input trace, and failed path. We provide more details of the RTB-ORACLE in [1].

6.2.2 Formula Generation

The goal of *FormGen* is to produce a quantifier-free, first-order logic formula that represents a set of read-traces. The symbolic variables of this formula represent read-values of instructions executed along each path in the path-set. Constraints within the formula limit the values that each symbolic read-value may take on. The set of all concrete read-traces represented by this formula, then, is the set of all constraint-satisfying assignments for symbolic variables.

The formula produced by *FormGen* must meet two requirements. First, a satisfying read-trace for the formula must result in a run that takes a path in the given path-set, when instantiated with the given schedule and input-trace. To meet this requirement, *FormGen* employs symbolic execution [12]–a method for deriving a formula representing the set of all read-traces along a given path. *FormGen* symbolically executes the program along each path in the path-set and outputs the disjunction of all per-path formulas. For further details of symbolic execution, we refer the reader to [4].

The second formula requirement is that a satisfying read-trace, when instantiated along with a given schedule and input trace, must result in a LOR-consistent run. Meeting this requirement is challenging because determining whether a read-trace is LOR-consistent or not requires that we know which reads may race and what values those racing reads may return. To address this challenge, *FormGen* relies on a race oracle. We describe the oracle in Section 6.2.3, but here it suffices to know that *FormGen* invokes the oracle to determine if an instruction races and then constrains the corresponding symbolic read-value to the possible read-values reported by the oracle.

6.2.3 Race Oracle

We have built a race-oracle in the form of a static race detector. In this section, we provide a specification of the detector, but elide operational details. The interested reader may find these details in [1].

The fundamental *soundness* requirement of the detector is that it returns a superset of *Races*(E) – the set of memory access instructions that race in E, where E is the set of runs that conform to a given lock-order, input-trace, and program path. If it does not report all races in *Races*(E), then PATH-SELECT (Section 6.2.1) will miss backtracking points, in turn rendering path-selection unable to find a valid path-set. Moreover, missed races will break formula generation; the formula may not describe the complete set of path-deterministic read-traces. Consequently, the output-matching phase may fail.

The detector should also have high *precision*, meaning that it reports few races other than those in $Races(E)$. Imprecision has two consequences. The first is performance. Specifically, PATH-SELECT may have to explore more paths before converging as a result of the RTB-ORACLE identifying false RTBs. Recall that it relies on the race oracle. The second consequence is some loss of memory-consistency. That is, SI-DRI's consistency relaxation will be applied, unnecessarily, to reads that could never race in E. This in turn increases the likelihood the read will return the value of a subsequently scheduled write, potentially confusing the developer during debugging.

6.3 Output Matching

To determine if a selected schedule, input, and read-trace set is output-deterministic, we could instantiate a run for each selected schedule, input and read-trace in the read-trace set, and then check the outputs. But instantiation is costly; we would have to execute the program along each such determinant. Instead, the output matching stage leverages a formula solver to conceptually perform this running and checking of outputs without actually instantiating a run.

Output matching works in two steps. In the first step, we augment the formula generated in the previous phase (*FormGen*) 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 formula to STP [4]– a constraint solver that, in the domain of software-analysis, can often find a satisfying assignment to an input formula in polynomial time. If there is a satisfying solution, then the solver reports one possible assignment of read values (i.e., produces a read-trace).

7. 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 calls for one sampling point for every instruction following a branch in the original run, in effect encoding the original run's path. While this additional information increases the recording overhead, it leads to a significant reduction of inference time. This is to be expected since knowledge of all branch outcomes enables our path selection algorithm (PATH-SELECT, Section 6.2.1) to recover from any divergence in just one backtrack. In practice, knowledge of the original run's path precludes the need to even invoke PATH-SELECT, hence providing an exponential reduction of the search-space.

8. IMPLEMENTATION

We implemented `ODR` as a lightweight user-level middleware layer for Linux/x86 applications. It 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 [16] and LibVEX [19], an open-source symbolic execution tool and binary translator, respectively. We encountered many challenges when developing `ODR`. In this section we describe the key challenges most relevant to our inference method.

8.1 Tracing Inputs and Outputs

The primary challenge of user-level I/O tracing is in achieving completeness. 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()`, were easy to handle; `ODR` just records their return values. But many others such as `sys_kill()`, `sys_clone()`, `sys_futex()`, `sys_open(), and sys_mmap()` required more extensive emulation work, largely to ensure deterministic signal delivery, task creation and synchronization, task/file identifiers, file/socket access, and memory management, respectively.

A secondary challenge is that of intercepting I/O events. There are many ways to do it, for instance via `sys_ptrace()`. But we found that a small degree of custom kernel support was necessary to achieve efficiency and completeness. Specifically, we employ a kernel module that 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 without penalty for most user-level applications.

8.2 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 [14] for the acquire operation preceding the memory operation and increments the Lamport clock for the subsequent release operation. We could have 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 schedule-trace search space.

8.3 Tracing Branches

QI-DRI requires a branch-trace, as discussed in Section 7. `ODR` captures branches in software using the Pin binary instrumentation tool [15]. Software binary translation incurs some overhead, but it is faster or more flexible than the alternatives (e.g., LibVEX or x86 hardware branch tracing [11]). To obtain low logging overhead, we employ an idealized, software-only 2-level/BTB branch predictor [9] 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.

8.4 Formula Generation

There are two key challenges in generating a formula. First, we must do so at the instruction level on Linux/x86. Using source code can make this problem easier [4], but would hinder `ODR`'s goal of working with arbitrary programs. Second, the generated formulas must be small in size. Otherwise, the constraint solver may not be able to solve them in a reasonable amount of time.

To address these challenges, our formula generator leverages the Catchconv [16] symbolic execution tool. Catchconv addresses the first challenge by producing instruction-level constraints on Linux/x86. Catchconv addresses the second challenge by generating a formula with size proportional only to the number of instructions tainted (i.e., whose parameters are influenced) by symbolic variables, as opposed to the total number of instructions. Since symbolic variables in our problem domain represent racing reads, and because racing reads have little influence on instructions (as verified by our evaluation), the resulting formulas are very small.

9. PERFORMANCE

In this section, we evaluate ODR under 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 on the record-inference plane of the DRI tradeoff space.

9.1 Setup

We evaluate seven parallel applications: *radix*, *lu*, and *water-spatial* from the Splash2 suite [23], 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 Splash2 suite because some (e.g, FMM) generate floating point constraints, which our current implementation does not support (see Section 10). Henceforth, we refer to the Splash2 applications 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 two-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.

9.2 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 (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 8.1).

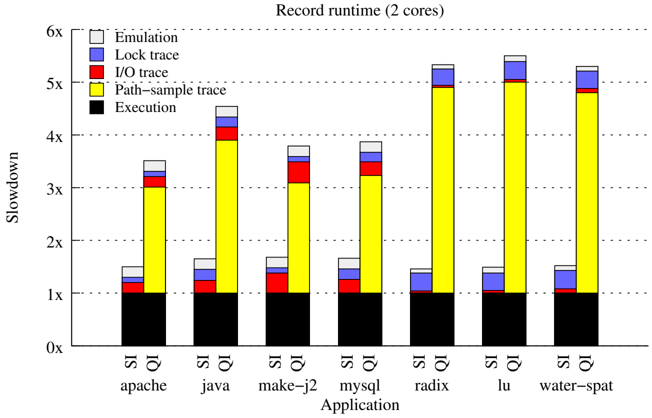

Figure 11: ODR's record-mode runtimes, normalized with native application execution time, for both SI-DRI and QI-DRI.

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 comparable on several others. For instance, ODR outperforms SMP-Revirt [5] on *make-j2* (by 0.3x) and *radix* (by 0.1x) for the 2-processor case. ODR does better because these apps exhibit lots of false-sharing. False-sharing induces frequent CREW faults on SMP-ReVirt, but not on ODR since it does not record races. ODR approaches RecPlay's [21] performance (within 0.4x) for the SP2 apps. 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 apps 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, in the two-processor case, SMP-ReVirt and RecPlay achieve near-native performance on several SP2 apps (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 app performance 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 [18] 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

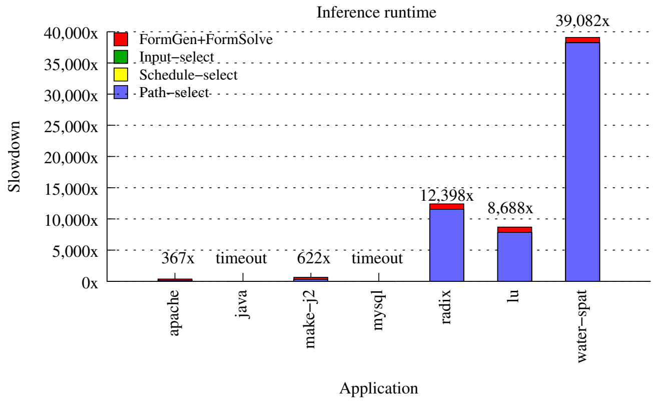

Figure 12: Inference runtime, normalized with native application execution time, for SI-DRI. Applications that did not finish in the first 24 hours are denoted by *timeout*.

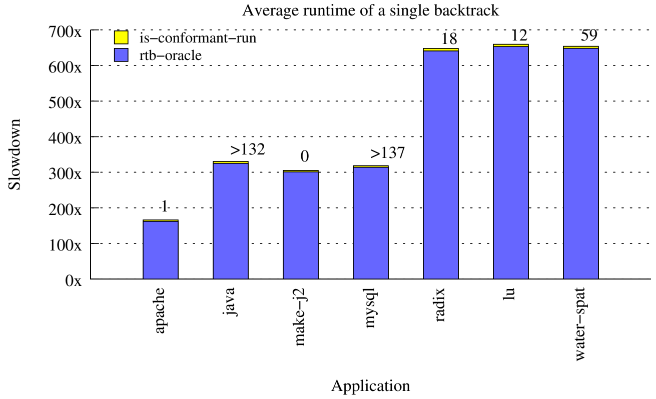

Figure 13: The average runtime, normalized with native runtime, of one backtrack performed by SI-DRI during PATH-SELECT, broken down into its two subroutines. The total number of backtracks is given at the top of each bar. For applications that timeout, this number is just a lower-bound.

make-j2, which transfer large volumes of data. The second bottleneck is the emulation. As discussed in Section 8.1, triggering a signal on each syscall and emulating task and memory management at user-level can be costly.

9.3 SI-DRI **Inference Mode**

Figure 12 gives inference slowdown for each application. The slowdown is broken down into 4 major SI-DRI stages: schedule-select, input-select, path-select, and formula generation and solving (*FormGen+FormSolve*). The path-select and *FormGen+FormSolve* stages account for a vast majority of the inference time. Since its query does not contain a path or read-trace, SI-DRI has 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 *FormGen+FormSolve* within the timeout period (24 hours). The secondary bottleneck is *FormGen+FormSolve*, taking up the remaining average 25% percent of inference time. We investigate each bottleneck in the following sections.

9.3.1 Path-Selection

We expected path-selection to be the primary cause of SI-DRI's slowdown. After all, SI-DRI's path-selection algorithm (PATH-SELECT, Section 6.2.1) may backtrack an exponential number of times. We also expected the cost of each backtrack to play a strong secondary role. To verify these expectations, we counted the number of backtracks and measured the average cost of a backtracking operation. The results, shown in Figure 13, contradict our expectations. 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.

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, Section 6.2.1). *make-j2*, for instance, does not have any shared-memory races at user-level, and hence no divergences that induce backtracks. 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, hence resulting in divergences that require a large number of backtracks to resolve. Consider a loop that contains a race, and no sampling points. Thus, the earliest point 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 the backtracking cost 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

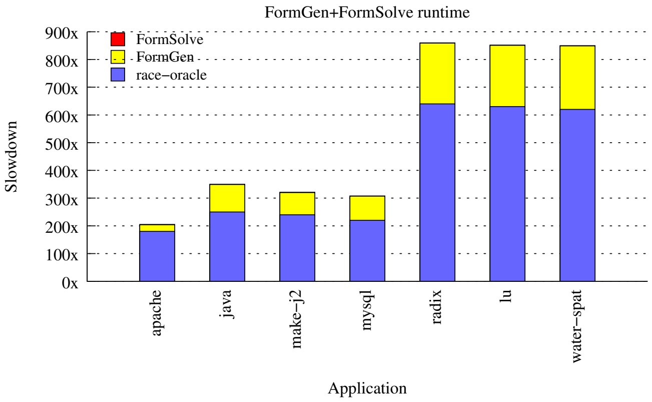

Figure 14: Runtime of the *FormGen* and *FormSolve* phases, normalized with native runtime, broken down into its three stages.

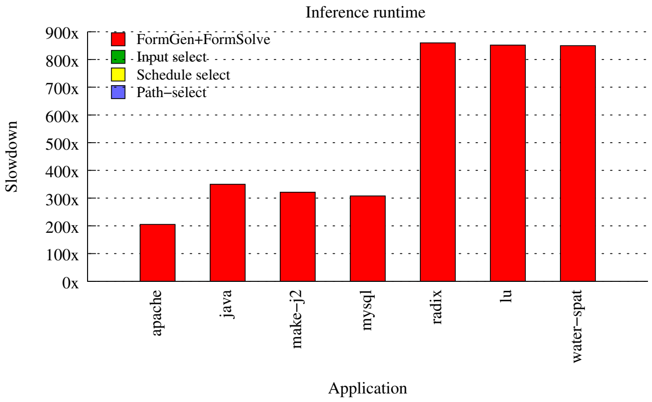

Figure 15: Inference runtime overheads for QI-DRI. All applications finish well within the first 24 hours.

major parts, shown in Figure 13. The first part is the cost of invoking the RTB-ORACLE (Section 6.2.1), needed to inteligently identify backtracking points. The second is the cost of invoking IS-CONFORMANT-RUN (Section 6.2.1), 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 expected the RTB-ORACLE to be expensive because it involves race detection and taint-flow analyses (Section 6.2.1) over the entire path up till the point of divergence. In theory, the RTB-ORACLE need not be run over the entire failed path on every backtrack, but we leave that to future work.

9.3.2 Formula Generation and Solving

We expected formula generation (*FormGen*) and solving (*FormSolve*) to be slow, especially since *FormSolve* 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 6.2.3), needed to identify which accesses may race. The second part is *FormGen*, used to encode a set of candidate read-traces as a logic formula. And the third part is *FormSolve*, needed to find an output-deterministic read-trace from the candidate set, which in turn involves invoking a formula solver. The breakdown contradicted our hypothesis in that most of the inference is spent in the race-oracle (a polynomial time procedure 6.2.3), not *FormGen* or *FormSolve*.

FormGen and *FormSolve* are fast for two reasons. The first is that our formula generator (Section 6.2.2) generates formulas 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 apps 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 constraints are desirable because the constraint solver we use can solve them in polynomial time [6]. We did not encounter any races that influenced the inputs of hash functions or other hard nonlinear 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 a given path. Because a set may contain millions of accesses, the process is inherently slow, even with our $O(n)$ set-intersection algorithm. We refer the reader to [1] for a more detailed explanation.

9.4 QI-DRI Record and Inference Modes

As we have shown so far, SI-DRI leads to a low recording overhead, but its inference time is prohibitive. In this section, we evaluate 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 8.3. 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 [3], it is greater than other software-only approaches, for some apps. *R*adix, for example, takes 3 times longer to record on ODR when using QI-DRI than with SMP-ReVirt [5].

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 sub-stage of read-trace guidance (the most expensive part of SI-DRI) since the original path of each thread has been already recorded. In the absence of path-selection overhead, the *FormGen+FormSolve* stage dominates. As explained in Section 9.3.2 and illustrated in Figure 14, the largest percentage of time in the *FormGen+FormSolve* stage is spent in the race-oracle (Section 6.2.3).

9.5 Replay Mode

Figure 16 shows replay runtime, normalized with native runtime, broken down into the cost of replaying each component of the determinant produced by DRI.

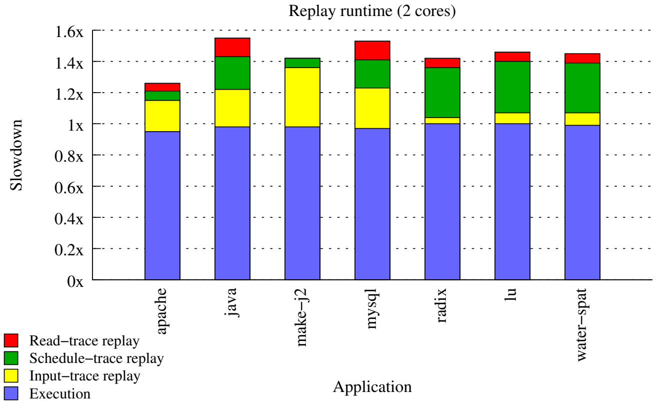

Figure 16: Two-processor replay-mode runtime overheads normalized with native execution time.

The surprising result here is that schedule-trace replay and read-trace replay are much faster than what one might expect of a serialized execution in which all reads are intercepted. Two additional optimizations employed by ODR explain these results. For the first optimization, rather than follow the schedule-trace precisely, ODR simply replays the lock-order of the schedule-trace (which for SI-DRI and QI-DRI is the original lock-order). For the second optimization, rather than replay all read-values in the read-trace, ODR replays just those of racing reads.

Replay speed is not near-native largely due to the cost of intercepting and replaying lock-instructions, a key bottleneck in record mode as well. Applications 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. 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 [13].

10. LIMITATIONS AND FUTURE WORK

ODR has several limitations that warrant further research. Here we present key limitations most relevant to the evaluated inference methods (SI-DRI and QI-DRI) along with possible improvements.

Inference time. The inference times for SI-DRI and, to a lesser extent, QI-DRI are impractically high for many applications. The key bottleneck, path selection (Section 6.2.1), can be fundamentally improved in many ways. One is to find a middle ground between SI-DRI and QI-DRI, for instance by placing only potentially-race-tainted branches in the path-sample. Another fundamental improvement is to offload the task of finding the path to the formula solver. The solver can apply reasoning to search paths more efficiently than PATH-SELECT, which employs a primitive guess-and-check method.

At the implementation level, a big improvement would be to simply avoid invoking the race oracle on the entire path under consideration for each PATH-SELECT backtrack.

Recording overhead. ODR's recording slowdown under SI-DRI, though low, is still too high for always-on production use. One bottleneck, lock-tracing overhead, can be reduced by tracing locks at the library rather than instruction granularity. The other bottleneck, system-call emulation overhead, can be reduced by shifting more work into the kernel.

	Data races	Multiple CPUs	Efficient and scalable recording	Software-only	Determinism
Jockey [22]	Yes	No	Yes	Yes	Value
RecPlay [21]	No	Yes	Yes	Yes	Value
SMP-ReVirt [5]	Yes	Yes	No	Yes	Value
iDNA [3]	Yes	Yes	No	Yes	Value
CapoOne [18]	Yes	Yes	Yes	No	Value
ODR	Yes	Yes	Yes	Yes	Output

Figure 17: Summary of key related work.

ODR's recording slowdown under QI-DRI is much too high even for periodic production use. The key bottleneck, branch-tracing, can be dramatically reduced using program analysis. For instance, the TraceBack branch tracing system [2] slows down native execution by only an average of 60% for CPU intensive benchmarks.

Formula solving. For inference to work, the formula solver must be able to find a satisfying assignment for the generated formula. In reality, formula solvers have hard and soft limits on the kinds of formulas they can solve. For example, no solver can invert hash functions in a feasible amount of time, and the constraint solver we use (STP [4]) does not support floating-point arithmetic.

All of the formulas we encountered were limited to linear bit-vector arithmetic operations, which STP solves in polynomial time [6]. However, some applications from the Splash2 suite (e.g., FMM) do generate floating point constraints. A potential workaround is to not generate any constraints for such unsupported operations and, instead, treat them as blackbox functions that we can simply skip during replay.

11. RELATED WORK

Figure 17 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 [22] or incur huge slowdowns [3]. Systems such as RecPlay [21] and more recently R2 [8] 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 (tracing and inference) can be ported to a variety of commodity architectures.

The current implementation of ODR is not as record-efficient as hardware multiprocessor replay systems, but is comparable. CapoOne [18], for instance, outperforms ODR by an average factor of 0.5x. On the other hand, ODR is, overall, the most record-efficient system of existing software multiprocessors systems that replay races. SMP-Revirt, ODR's closest competitor, succumbs to false sharing (for some applications) as the number of cores increases, while ODR does not.

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 merely goes one step further. Relaxed determinism was recently re-discovered in the Replicant system [20], 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.

12. 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 a run 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 10. Looking forward, output-deterministic replay of networked applications seems promising.

13. ACKNOWLEDGEMENTS

We are indebted to our shepherd Steven Hand for his detailed feedback. The paper is fundamentally better because of his efforts. We also thank the anonymous reviewers, Dennis Geels, Shan He, Jayanth Kannan, David Molnar, and Lucian Popa for feedback on early drafts of this paper. Finally, this research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Facebook, Hewlett-Packard, Network Appliance, and VMWare, and by matching funds from the State of California's MICRO program (grant 06-149) and the UC Discovery grant COM07-10240.

14. REFERENCES

[1] G. Altekar and I. Stoica. Output-deterministic replay for multicore debugging. Technical Report UCB/EECS-2009-108, EECS Department, University of California, Berkeley, Aug 2009.

[2] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: first fault diagnosis by reconstruction of distributed control flow. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 201–212. ACM, 2005.

[3] 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*, pages 154–163, New York, NY, USA, 2006. ACM.

[4] 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.

[5] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*, pages 121–130, New York, NY, USA, 2008. ACM.

[6] 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.

[7] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference, General Track*, pages 289–300. USENIX, 2006.

[8] 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.

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

[10] 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.

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

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[13] 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.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] 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.

[16] 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.

[17] 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.

[18] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In M. L. Soffa and M. J. Irwin, editors, *ASPLOS*, pages 73–84. ACM, 2009.

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

[20] 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.

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

[22] 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.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, February 1995. ACM Press.