ROOT: Replaying Multithreaded Traces with Resource-Oriented Ordering

Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin-Madison
{zev,harter,dusseau,remzi}@cs.wisc.edu

Abstract

*We describe ROOT, a new method for incorporating the nondeterministic I/O behavior of multithreaded applications into trace replay. ROOT is the application of **R**esource-**O**riented **O**rdering to **T**race replay: actions involving a common resource are replayed in an order similar to that of the original trace. ROOT is based on the idea that how a program manages resources, as seen in a trace, provides hints about an application's internal dependencies. Inferring these dependencies allows us to partially constrain trace replay in a way that reflects the constraints of the original program. We make three contributions: (1) we describe the ROOT approach, (2) we release ARTC, a new ROOT-based tool for replaying I/O traces, and (3) we create Magritte, a file-system benchmark suite generated by applying ARTC to 34 Apple desktop application traces. When collecting traces on one platform and replaying on another, ARTC achieves an average timing inaccuracy of 10.6% on our benchmark workloads, halving the 21.3% achieved by the next-best replay method we evaluate.*

1 Introduction

Quantitatively evaluating storage is a key part of developing new systems, exploring research ideas, and making informed purchasing decisions. Because running actual applications on a variety of storage stacks can be a painful process, it is common to collect statistics or traces on a single system in order to understand an application [4, 5, 8, 13, 18, 19, 20, 24, 25]. Trace replay is a useful technique for evaluating the performance of different systems [3, 10, 11, 14, 15, 17, 22]. An application trace may be collected on one system (the *source*) and replayed on another (the *target*) in order to predict how the application would perform on the target. Replaying traces at the system-call level is an appealing approach for this use case, since its high level of abstraction allows the user to evaluate changes in a wide variety of system components. In contrast, replaying a lower-level trace restricts the set of system changes that can be effectively evaluated to those at or below the level of the trace itself. For example, replaying a block-I/O trace is of little use if the user wishes to evaluate the performance of an alternate file system.

At first it might seem that trace replay would offer easy insight to an application's performance on an alternate storage stack, since the actions replayed are precisely the actions the real application performed. However, replay must take into account feedback loops between the workload and the storage stack [6, 16]; for example, faster storage could cause a program to issue requests in a different order. Complicating trace replay further is the increasing complexity of many modern applications [7, 23]. Such applications often have many threads, and it is common for one thread to open a file, a second thread to write to it, and a third to close it.

There are two criteria by which we judge the quality of trace replay for performance prediction: *semantic correctness* and *performance accuracy*. The former measures how well the semantics of the operations recorded in the trace are reproduced by the replay; the latter measures how close the replay's performance on the target system predicts that of the original program.

In some trace replay scenarios, semantic correctness is nearly trivial; for example, there is little difficulty in replicating the semantics of a single sequential stream of block-I/O requests. With system-call replay, however, semantic correctness is less simple: files of the appropriate size must be set up at the appropriate paths, possibly with extended attributes and other metadata cor-

Copyright is held by the Owner/Author(s).
SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522734

rectly initialized. Considering multithreaded traces with the possibility of system-call reordering introduces further complexity: if an `open` and a `read` in two different threads are reordered with respect to each other, the `read` may fail with `EBADF`, deviating from the semantics of the original application.

Trace-replay tools should reflect the characteristics of applications, including the ordering dependencies of their execution. There are two artifacts that provide information about dependencies: the original program and traces. Unfortunately, a program's source code is often unavailable, and deducing full, application-level semantic dependencies from one trace collected on one storage device is generally not possible. However, the ways programs manage resources, as shown in a trace, can provide hints about program dependencies. In this paper we propose a new technique for extracting these hints from a trace: Resource-Oriented Ordering for Trace replay (ROOT). The ROOT approach is to observe the ordering of actions touching each resource in a trace and apply a similar ordering to those actions during replay.

We build a new tool, ARTC (an "approximate-replay trace compiler"), that applies the ROOT approach to UNIX system-call traces. ARTC constrains replay based on resource-management hints extracted from a trace. In order to extract meaningful hints, ARTC uses a detailed UNIX file-system model and knowledge of over 80 system calls to infer the complex relationships between actions and resources. For example, symlink awareness allows ARTC to track all the pathnames that point to a single file resource; similarly, a directory-tree model allows ARTC to determine the entire set of resources that are affected by directory renames.

We use ARTC to automatically generate a new cross-platform benchmark suite, Magritte, from 34 traces of Apple desktop applications [7]. Because many of these traces contain OS X-specific system calls, we employ novel emulation techniques for 19 different calls, allowing replay of the traces on other systems.

We compare ARTC against three simpler replay strategies: a single-threaded approach, a multithreaded replay that disallows reordering, and an unconstrained multithreaded replay with no inter-thread synchronization. We use the complex Magritte workloads to evaluate semantic correctness, finding that ARTC achieves error rates nearly identical to those of the more heavily constrained replays. For timing accuracy, we demonstrate the weaknesses of the simple replay methods with microbenchmarks designed to illustrate feedback effects involving workload parallelism, disk parallelism, cache size, and I/O scheduling. We also replay traces of an embedded database, and find that ARTC reduces average timing error from 21.3% (for the most accurate alternative) to 10.6%.

The rest of this paper is organized as follows. We explore different approaches to inferring program behavior from traces (§2) and define the ROOT approach to this problem (§3). We then describe our ROOT-based trace compiler, ARTC (§4), evaluate it in comparison to a set of simpler replay methods (§5), and provide a case study with Magritte (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Trace Mining

We now consider what types of information can be mined from traces for the purpose of replay. A single trace presents a set and ordering of actions that the program may generate when run on a specific system with a certain set of inputs.

Ideally, however, we would like to infer the entire I/O space of the program for a given input. In different environments, a program may generate different sets of I/O actions for different runs, and for each of these *I/O sets*, different orderings may or may not be possible. We define an *I/O space* as the set of all feasible I/O sets and the associated valid orderings. For example, a simple I/O space consisting of two I/O sets and four orderings could be described with the following set notation:

```
{{1,2}   => {[1,2], [2,1]},
 {1,2,3} => {[1,2,3], [2,1,3]}}
```

Depending on the type and quantity of the available trace data, different techniques may be used to infer the I/O space, and different degrees of accuracy will be achievable. We now define various types of trace data that may be available (§2.1) and describe three inference techniques, including our new technique, ROOT (§2.2).

2.1 Trace Inputs

There are three primary attributes of parallel trace data: the number of traces, active vs. passive collection, and inclusion of synchronization information.

First, some inference techniques require many traces. Each trace represents one point in the I/O space of the application; observing many points makes it easier to guess the shape of the whole space. Unfortunately, collecting many traces on the same system will tend to explore only certain areas of the whole I/O space.

Second, traces may be collected passively or actively. Passive tracing simply records an application's I/O requests, doing nothing to interfere. In contrast, active tracing may perturb I/O; certain operations may be artificially slowed so as to see which other operations are delayed. This allows direct deduction of dependencies and methodical exploration of the I/O space.

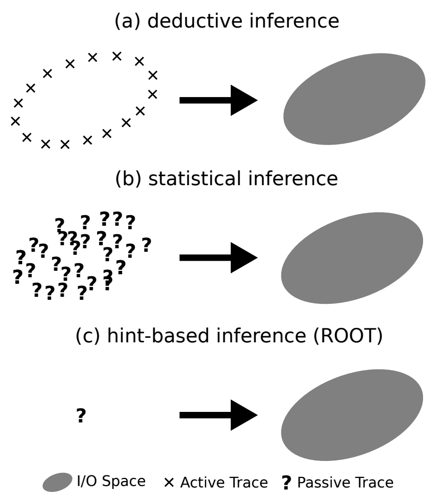


Figure 1: Techniques for I/O-space inference.

Third, traces may consist of only calls that occur at the boundary of an external storage API; alternately, they may also include internal synchronization operations. Details about internal synchronization may reveal certain dependencies; for example, if two I/O requests at different times were both issued while a given lock was held, we can infer that an ordering where the two I/O requests are concurrent is not part of the program's I/O space. Internal program logic also affects ordering, however, so tracing locks is not a complete solution.

2.2 Inference

We now describe three I/O-space inference techniques, including ROOT, based on three different types of trace information. These are summarized in Figure 1.

Figure 1(a) illustrates a *deductive inference* approach based on active tracing. Active traces allow methodical exploration of the I/O space via controlled experimentation. //TRACE is an example of an active-tracing tool [16]. An I/O space can be determined by collecting numerous traces, artificially slowing different requests each time, and observing which other requests are delayed as a result. While this is an elegant approach, it is inconvenient and time consuming to collect many traces, especially at the slowed speed. In production, slowing I/O may be unacceptable, and collecting traces multiple times with the same input may not be possible.

Figure 1(b) illustrates a *statistical inference* approach based on passive tracing. Some debugging tools use this approach to infer the causal relations between RPC calls, but not for replay [2]. Although this approach has the advantage that traces are much easier to collect, it is likely that much of the I/O space will not be explored unless traces are collected in many different environments.

Figure 1(c) shows the goal of the ROOT approach: to infer as much as possible about an I/O space given a single passively-collected trace, with no details about application internals (*e.g.*, locking). Inferring anything about an I/O space given a single data point might seem impossible; however, the resource usage patterns of a trace provide useful hints about the program's I/O space.

For example, if a program performs two reads from the same file, the reads may use the same file descriptor for both requests, or different file descriptors. The use of different file descriptors may indicate that the reads are unrelated, and hence could be replayed concurrently. Likewise, they may be issued from the same thread or two different threads.

While a human reading through a trace would likely be able to infer more application-level logic than an automated tool, creating benchmarks via manual trace inspection would be an unpleasant task. Thus we propose a new approach called ROOT: resource-oriented ordering for trace replay. ROOT defines a trace model, making it easier to create tools that reason about traces. ROOT also defines a notation for expressing the "hints" a human reading a trace might use to make a reasonable guess about the target program's dependency properties. We describe ROOT in §3.

Of course, the ROOT approach can sometimes make incorrect inferences because hints can be misinterpreted. We do not attempt to make more accurate inferences than the deductive or statistical methods; those techniques have the advantage of being based on a great deal more data. The ROOT approach is useful when a benchmark is desired, but knowledge about the original application is limited. This will typically be the case when studying traces of production systems, where inputs may be uncontrollable and the overheads of active tracing are unacceptable. Furthermore, it is relatively uncommon for companies to collect and share traces; motivating them to collect active traces or enough traces to apply statistical inference may be infeasible.

One weakness of ROOT is that it assumes the I/O space will consist of a single I/O set. Given a series of actions in a trace, it is reasonable to infer how they might be reordered; however, it is essentially impossible to correctly guess that a program sometimes generates a certain request if that request never appears in the trace. We do not view this as a severe limitation; inference based on methodical exploration could hypothetically deduce I/O spaces consisting of multiple I/O sets, but existing tools based on this approach (*e.g.*, //TRACE) also only work for I/O spaces consisting of a single I/O set.

3 ROOT: Ordering Heuristics

By enforcing an approximately-correct partial ordering on replay actions, replay tools can generate realistic I/O that resembles the original program's behavior. In this section, we define ROOT's hint-based ordering rules for replay. Our constraints are oriented around resources, such as files, paths, and threads. The key idea is that the set of actions involving a given resource should be replayed in a similar order as in the original trace. If all actions in a trace interact with the same resource, then replay will be highly constrained, but if there is little overlap between the resources touched by different actions, there will be little constraint on the replay order.

Although resource-oriented ordering is simple in theory, real storage systems have complex, many-to-many relationships between actions and resources; some types of actions (*e.g.*, directory renames) can impact an arbitrarily large set of resources (*e.g.*, paths). The relationship between an action and the resources it touches cannot be inferred by looking at the trace record for the action by itself. Rather, inferring the relationships requires a trace model that considers each action in the context of the entire trace and an initial snapshot of system state.

In §3.1, we describe a general trace model applicable to traces from a variety of storage systems (*e.g.*, key-value stores or file systems). In §3.2 we define and intuitively justify several rules that can be applied to a trace to obtain a partial ordering of actions with which to guide replay. In §4 we describe ARTC's use of our trace model and ordering rules to replay system-call traces.

3.1 Trace Model

A *trace* contains a totally-ordered series of *actions*. The types of actions are system specific; a key-value store might have `put`, `get`, and `delete` actions, whereas a file system might have `open`s, `read`s, and `write`s. Each action interacts with one or more *resources*; threads, keys, values, paths, and files are examples of resources.

A simple file rename across directories might involve five resources: the thread performing the rename, source and destination paths, and the directories containing these paths. Conceptually, an *action series* is associated with each resource, consisting of all the actions related to the resource in the order they occurred in the original execution. All our rules are based on action series; it is, however, unnecessary to ever materialize such lists.

Some resources point to other resources. For example, a path might point to a directory, which in turn might point to other paths. Some actions that touch a resource also touch all other resources it transitively points to.

Some resources have *names* that appear in the trace. A file resource does not itself have a name, but it might be pointed to by a path, which does. The same name might apply to different resources at different points in a trace; for example, "3" could be a name designating different file descriptors at different times. Our model differentiates uses of the same name with *generation numbers*, increasing integers associated with each such use, which together with a name uniquely identify a resource.

(a) Example Trace

```
1 [T1] mkdir("/a/b")                = 0
   Resources:
   T1,dirA,dirB,path(/a/b)
2 [T1] open("/a/b/c",CREATE)        = 3
   T1,dirB,file1,path(/a/b/c),fd3
3 [T1] write(3, ...)                = 8
   T1,file1,fd3
4 [T1] close(3)                     = 0
   T1,file1,fd3
5 [T1] rename("/a/b", "/a/old") = 0
   T1,dirA,dirB,file1,four paths...
6 [T2] open("/x/y/z")               = 3
   T2,dirY,file2,path(/x/y/z),fd3
7 [T2] open("/a/b")                 = 4
   T2,dirA,file3,path(/a/b),fd4
                  ...
```

(b) Action Series

Resource	Actions
thread(T1)	1,2,3,4,5
thread(T2)	6,7
dirA	1,5,7
dirB	1,2,5
dirY	6
file1	2,3,4
file2	6
file3	7
path(/a/b)@1	1,5
path(/a/b)@2	7
path(/a/b/c)@1	2,5
path(/a/old)@1	5
path(/a/old/c)@1	5
path(/x/y/z)@1	6
fd3@1	2,3,4
fd3@2	6
fd4@1	7

Figure 2: Example action series. *A snippet from a simple system-call trace for two threads is shown in (a). Beneath each event, a comment lists the resource touched by each system call. (b) shows the action series corresponding to each resource that appears in the trace.*

Figure 2 provides an example showing how action series are derived from a system-call trace. The series for thread `T1` is simply the set of actions executed by the thread (1, 2, 3, 4, 5), in the order they were executed. The series for `dirA` (1, 5, 7) is the set of actions that accessed `dirA`, in the order they occurred. Note that action series do not distinguish between subjects (*e.g.*, threads) and objects (*e.g.*, directories). The figure also shows different action series for `fd3@1` and `fd3@2`. This "name@generation" notation is used to distinguish between resources when the same name is used for different resources at different times. Here, `3` is a shared name for the file descriptors created in actions 2 and 6.

3.2 Ordering Rules

We suggested in §2.2 that how a program manages resources, as shown in a trace, provides hints about its I/O space. Given a trace model, we can now discuss these hints more formally and define replay rules.

The rules we define determine an I/O space for a replay benchmark. Ideally, the I/O space for the benchmark will be similar to that of the original application. However, there are two ways we might deviate from this goal. First, a rule might be too restrictive, resulting in *overconstraint*. In this case, the original I/O space may contain an ordering for an I/O set that the replay I/O space does not contain. Second, a rule might be insuf-

Rule	Definition
Stage	*acts*[create] < *acts*[i] < *acts*[delete]
Sequential	*acts*[i] < *acts*[i+1]
Name	N@G.*acts*[last] < N@(G+1).*acts*[first]

Table 1: Ordering Rules. *a1 < a2 means action a1 must be replayed before action a2. acts[create] and acts[delete] represent acts[first] and acts[last] respectively when the first action in a series is a create or when the last action is a delete. When this is not the case, the constraint does not apply.*

ficiently restrictive, resulting in *underconstraint*. In this case, the replay I/O space may contain an ordering for an I/O set that the original I/O space does not contain.

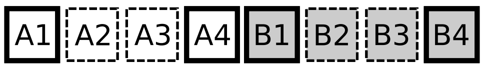
We say that a *stronger* rule A *subsumes* a *weaker* rule B if the orderings allowed by rule A are a strict subset of those allowed by rule B. In this case, if B causes overconstraint, A will as well. Likewise, if A allows underconstraint, B will as well.

We have identified three rules based on action series that are useful for replay; these are summarized in Table 1. The first rule, *stage ordering*, simply says that an action that creates a resource must be played before any uses of the resource, and also that any uses of the resource must be played before a deletion. The intuition behind stage ordering is that when we observe a successful event in a trace, we assume the program took some action to ensure success, so replay should do likewise.

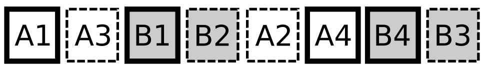
The second rule, *sequential ordering*, forces all actions involving a resource to replay in the same order as in the original trace. Sequential ordering is a stronger constraint, subsuming stage ordering, but may lead to overconstraint. For example, if multiple reads from the same file all touch the same resource, it is very possibly correct to allow these reads to be reordered during replay, but sequential ordering would disallow this. In contrast, stage ordering might be too weak: reordering two reads from the same file could be incorrect if the first retrieves indexing information and the second relies on the result of the first to determine where in the file to read from. The intuition behind sequential ordering is that data dependencies may be more likely when actions access the same resources rather than disjoint sets of resources; constraints should be tighter in such cases.

The third rule, *name ordering*, requires that the action series of different generations of the same name are neither overlapped nor reordered during replay. Sequential- and name-ordering each allow some orderings not allowed by the other. The intuition behind name ordering is that when a programmer reuses the same name for different resources, the resources are likely related.

Figure 3(a) shows an example trace of actions on two resources, A and B, that use the same name at different times. Figure 3(b) gives an example replay ordering, and Figure 3(c) describes how the replay would violate different ROOT rules. The replay of generation A is allowed by stage ordering because the sequence begins and ends with create and delete actions, respectively, but violates sequential ordering because the two middle actions (A2 and A3) are reordered. The replay of generation B violates stage ordering because the deletion action is not last, and thus also violates sequential ordering. Finally, actions belonging to generation B start replaying before A is finished, which violates name ordering since A and B are different generations of the same name.



(a) Original trace order



(b) Replay order

	Stage	Sequential	Name
Generation A	*none*	A3<A2	B1<A4
Generation B	B4<B3	B4<B3	

(c) Violations

Figure 3: Examples of valid and invalid orderings. *Each square represents an action. White and grey squares belong to two consecutive generations of the same name. Thick borders indicate creation and deletion events.*

Because rules vary in strength, one must decide which rules to apply to which resources when employing ROOT. In §4.2, we describe ARTC's default use of the rules for UNIX file-system resources and the reasoning for each. More broadly, however, we suggest three guidelines for applying the rules in a new context. First, domain knowledge should be used. For example, if it is known that a programmer generally intentionally chooses names for a certain resource (*e.g.*, a path name), name ordering should apply, but if the names are chosen arbitrarily, name ordering might cause overconstraint. Second, the costs of different types of mistakes should be taken into account; overconstraining a replay might skew the timings of certain actions, but underconstraining might cause the actions to fail, and thus finish instantly. Third, if many actions fail during replay, underconstraint is a likely cause.

4 ARTC: System-Call Replay

We now describe ARTC, a benchmarking tool that applies the ROOT approach to system-call trace replay on UNIX file systems. We now discuss goals for the tool (§4.1), demonstrate how the three ROOT rules abstractly defined in the previous section concretely apply to UNIX file systems (§4.2), and detail our implementation (§4.3).

Resource	Stage	Sequential	Name
program		●	
thread		●$_{\mathit{req}}$	
file	○	●	
path	●$_{\mathit{joint}}$	○	●$_{\mathit{joint}}$
fd	●	●	
aiocb	●	○	○

Table 2: Replay modes. *Circles represent reasonable ways to apply rules to resources; filled circles are modes currently supported by ARTC.* `thread_seq` *is always required.* `path_stage` *and* `path_name` *must be applied jointly.*

4.1 Goals

The aim of ARTC is to be a broadly applicable storage benchmarking tool, offering a flexible set of parameters while remaining easy to use.

Portability: ARTC attempts to support realistic cross-platform replay. Because traces from one system often include system calls that are not supported on others, ARTC emulates these calls, issuing the most similar call (or combination of calls) on the target system.

Ease of use: ARTC benchmarks make it simple for end users to apply them to a file system. All that is required for basic use is the compiled benchmark and a directory in which to run the benchmark (perhaps the mountpoint of a file system to be evaluated). There is no need to describe a benchmark using a specialized configuration language or determine the values of non-default parameters to measure the performance of a file system. Also, ARTC makes it easy to create new benchmarks by supporting standard tracing tools that are often preinstalled in UNIX environments (*e.g.*, `strace`).

Flexibility: ARTC provides a variety of optional tuning parameters, controlling how initialization is done, the speed at which actions are replayed, the ability to disable specific ordering constraints, and how certain actions are emulated during cross-platform replay.

Correctness: ARTC attempts to generate benchmarks with nondeterministic behaviors resembling the nondeterminism of the original applications as closely as possible given the information available in the traces. Despite this nondeterminism, ARTC's ordering constraints enforce that the replay's *semantics* should match those of the original trace as closely as possible.

4.2 ROOT with System-Call Traces

We now discuss the application of ROOT to system-call traces. We consider six types of resources: programs, threads, files, paths, file descriptors (FDs), and asynchronous I/O control blocks (AIOCBs). We focus on single-process replay, so all the actions in a trace are associated with a single program resource, as well as one of the many thread resources. Many actions will access file resources via paths and file-descriptor resources. Finally, AIOCBs are used to manage asynchronous I/O on file descriptors; AIOCBs point to file descriptors.

Table 2 shows which rules could reasonably be applied to which resources and which are supported by ARTC's replay modes. Though all supported constraints except `program_seq` are enforced by default, ARTC allows any combination of ordering modes to be selected for replay, with two restrictions. First, sequential ordering is always applied to threads; second, for paths, stage and name ordering may only be applied jointly. A discussion of the replay modes follows:

Programs: All actions in a trace involve a single *program* resource. Applying sequential ordering to the program represents the `program_seq` replay mode. `program_seq` is ARTC's strongest replay mode, subsuming all other modes; however, `program_seq` forces a total ordering on replay, typically resulting in severe overconstraint (the performance impact of `program_seq` is demonstrated in §5). Stage ordering does not make sense for the program resource because no action in the trace can be said to "create" the program; name ordering is irrelevant as there are not multiple generations of program resources in a single trace.

Threads: Each action in a trace is performed by exactly one *thread* resource. ARTC always enforces `thread_seq` mode, as it has no simple way to reorder actions within a thread during replay. In general, the order of actions performed by a single thread provides a good hint about program structure. Some patterns, however, such as thread pools, are clear exceptions; ARTC cannot infer these types of program structures. However, we are not aware of any other replay tools that can do so without additional details about program internals. Stage and name ordering do not apply to threads for the same reasons they do not apply to programs.

Files: We define a *file* as the data associated with a specific piece of metadata, such as an inode number. Inode numbers, however, do not appear in our traces, so the existence of files is only implicit. An accurate file-system model that considers symbolic links, hard links, and the behavior of various system calls allows us to determine when different paths (or file descriptors) refer to the same file, as well as when the same path name refers to different files at different times. Because files do not appear explicitly in traces, name ordering is irrelevant. Stage and sequential ordering apply, though; ARTC supports the latter with `file_seq`, a fairly strongly-constrained replay mode. When other resources refer to files, as they often do, `file_seq` subsumes stage or sequential ordering when applied to those resources. However, the rules for the following resources do prevent some orderings `file_seq` allows, such as when name ordering is relevant or when the resources refer to directories rather than regular files.

Paths: *Path* resources point to file resources and have names that appear in traces. All our ordering rules could be applied to paths; ARTC supports the joint application of stage and name ordering with `path_stage+` mode. We do not support stage ordering without name ordering because doing so would require the use of substitute names during replay. For example, if a trace shows that a path `"/a/b"` referred to different files at different times, replay would have to either prevent concurrent access to those files during replay (*i.e.*, use name ordering), or use substitute names (*e.g.*, `"/a/b1"` and `"/a/b2"`).

Applying stage ordering to paths assumes that when a trace action makes a successful access to a path, the program must have taken some measure to ensure its success. We believe this is a good hint in general, but it may sometimes cause overconstraint. For example, programs may use the `stat` call (which fails when a path does not exist) to determine whether a path exists. If a `stat` call succeeds during the original execution, it may be a coincidence; during replay, if certain actions finish sooner than they did during trace collection, it may be correct to replay a `stat` call sooner, even if the call would fail.

Similarly, applying name ordering assumes that different files are related if they use the same path name at different times. Because programmers or users choose most path names, we believe this to be a meaningful hint. While this is usually the case, one common exception is when path names are chosen arbitrarily (*e.g.*, names for temporary files). In this case, `path_stage+` may lead to overconstraint, but we suspect this situation is rare in practice since random file names are not generally chosen from a small set of possibilities and hence are unlikely to collide with each other.

File descriptors: Successfully opening a path produces a file descriptor (FD), which acts as another type of pointer to a file. ARTC supports stage ordering (`fd_stage` mode) and sequential ordering (`fd_seq` mode) for FDs. Although FDs have integer names that appear in a trace, these names are usually chosen by the operating system, so they provide no hints about the I/O space; thus, name ordering is of no real use for FDs. Additionally, since FD names are small integers, they can be easily remapped using a simple array, allowing descriptors that used the same name in the original trace to coexist simultaneously during replay.

Asynchronous I/O control blocks: Asynchronous I/O may be performed by wrapping a file descriptor in an asynchronous I/O control block (AIOCB) structure and submitting it in a request to the file system. Because file descriptors point directly to files, AIOCBs point indirectly to files. ARTC supports stage ordering for AIOCBs with `aio_stage` mode. Applying sequential ordering could also be potentially useful, even though ARTC does not currently support it.

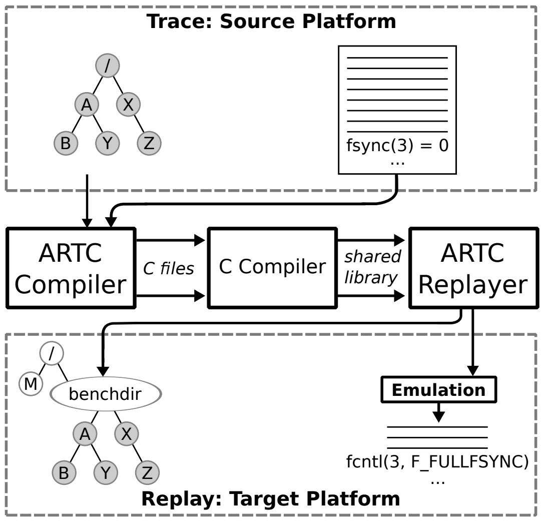


Figure 4: ARTC Components.

4.3 Implementation

Figure 4 show an overview of the main components of ARTC. Given a system-call trace and an initial file-tree snapshot collected on a source system, the ARTC compiler automatically generates a benchmark (§4.3.1). The ARTC replayer initializes an equivalent file-system tree on the target machine to match the snapshot (§4.3.2), in which the actions in the trace are replayed (§4.3.3). File-system APIs vary slightly across systems, so ARTC emulates recorded actions via the closest equivalent on the target machine when necessary (§4.3.4), supporting replay on Linux, Mac OS X, FreeBSD, and Illumos.

ARTC's implementation consists of approximately 12,000 lines of C and 4,000 lines of `bison` and `flex` grammars (as measured by `wc -l`), and is capable of replaying over 80 different system calls. A significant portion of the code is shared between the ARTC compiler and the ARTC replayer, but the two components comprise roughly equal fractions of the code size.

4.3.1 Compilation

ARTC currently supports `strace` output and a special `dtrace`-generated format used by the iBench traces (see §5.1), but trace parsing is cleanly separated from the core processing functionality, so ARTC can be readily extended to support new input formats. However, the core functionality assumes the following information will be available for each system call in the trace:

- Entry/return timestamps
- Numeric ID of issuing thread
- Type of call (*e.g.*, `open`, `read`, *etc.*)
- Parameters passed
- Return value

Some system-call parameters are not actually required; for example, ARTC ignores the buffer pointers passed to `read`. While our trace model could theoretically treat buffer pointers as another type of resource, we suspect buffer reuse would make it impossible to derive meaningful hints from the additional information.

In addition to a trace of actions, ARTC requires an initial snapshot of the parts of the file-system tree that the program accesses. It is unnecessary to record actual file contents in the snapshot; however, it is important to record the contents of directories, the sizes of files, and references made by symbolic links. Having an accurate model for symbolic links is crucial to enforcing the `file_seq` rule. Even when the same file is accessed via different paths, `file_seq` must constrain the accesses to be replayed in the same order as in the trace.

Given a trace and an initial snapshot, ARTC automatically generates C code, which is then compiled into a shared library. The shared library is later loaded by a general tool for replay (§4.3.3). The generated code consists of tables of static data (arrays of `struct`s) describing the resources and actions in the trace. We chose to generate C code as a simple way to serialize the replay information; generating input files that the replay program parses would work as well, though using pre-built data structures saves the runtime overhead of parsing a more generic input format.

4.3.2 Initialization

Before replay, it is necessary to restore the initial state snapshot in the directory where the benchmark will execute. During this stage, ARTC creates the necessary directories, populating them with files of the appropriate size containing arbitrary data, and creates any necessary symbolic links. Some special files (such as `/dev/random`) are created as symlinks to the corresponding special files in the target's root file system.

Because initialization may take much longer than the actual replay of some traces, ARTC can perform a *delta init* that is useful when most of the init files are already in place (*e.g.*, the file tree was previously initialized, and a prior replay only slightly modified the tree). Delta init only creates, deletes, or changes of the sizes of existing files as necessary to restore the initial state.

Initialization is not a major focus of our work, but ARTC could be extended to use initial snapshots with richer information about invisible file-system state. For example, for a log-based file system, replay speed will depend greatly on the order in which the initial files are created. A more sophisticated initialization could account for this, and even reproduce the fragmentation that occurs due to aging in real-world deployments [1, 21].

ARTC also includes options that make it easy to initialize overlaid file-system trees based on the snapshots for multiple traces, so that multiple traces can be replayed concurrently. For example, one could use Magritte (§6), our benchmark suite of Apple desktop applications, to run a workload similar to a user browsing photos in iPhoto while listening to music in iTunes.

4.3.3 Replay

ARTC's replayer is the component that actually performs system-call replay, enforcing the enabled ordering modes while doing so. Although our discussion of ordering modes has been in terms of action series, ARTC, like the programs that generate the traces to begin with, does not need to explicitly materialize such lists. Rather, ARTC enforces rules using standard synchronization primitives and the dependency information determined by the compiler. Each system call (action) includes a condition variable that other threads can wait on if an action they are about to replay is dependent on that action. For example, before a given thread replays an action that uses a certain file descriptor, it checks if the `open` call that created that file descriptor has already been replayed, and if not, waits on the `open` action's condition variable. When the replay of an action completes, the thread that replayed it performs a broadcast operation on the action's condition variable in order to wake any threads that may be waiting on it.

Stage ordering: except for a resource's *create* action, all other actions will wait on the create action before replaying, enforcing that it is the first of that resource's associated actions to replay. *Delete* actions have a dependency on each other use of the resource, though for space-efficiency reasons our current implementation uses a separate structure for the resource with a count of remaining uses and a condition variable of its own.

Sequential ordering: Each action belongs to the action series of one or more events. For each such series, the action in question has a dependency on the previous action in the series, and correspondingly waits for its completion before proceeding with its own replay.

Name ordering: When an action is the first of a new generation of a resource on which name ordering is applied, it has a dependency on the last event of the preceding generation, and waits for it to complete.

We use this resource and action bookkeeping to enforce all ordering rules except `thread_seq` and `program_seq`. Because sequential ordering is always enabled for threads, we simply use a replay thread for every thread that appeared in the original trace. Each of these threads loops over its own actions from the original trace, playing each one in order once all its dependencies are satisfied. When `program_seq` is used, all trace actions are instead replayed from a single replay thread in the order in which they appeared in the original trace.

Besides enforcing ordering rules during replay, ARTC is also capable of considering timings from the original trace. For example, the original trace might show that even after all the inferred dependencies for an action are satisfied, the action is executed after some time interval, which we call *predelay*. Predelay may be due to computation. It is not our goal to have a sophisticated model of computation, but ARTC provides some basic options for incorporating predelay during replay. ARTC may ignore predelay (AFAP, or as-fast-as-possible mode), sleep for the predelay time (natural-speed mode), or use some multiple of predelay, perhaps based on CPU utilization information (if available). Given our very simplistic model of computation, we do not expect ARTC to produce accurate timings for compute-bound workloads.

After finishing replay of the entire trace, the replayer outputs basic timing information, such as the elapsed wall-clock time, as well as detailed data about why a replay performed the way it did, such as per-thread timing reports and latencies for each call. Additionally, details about the similarity of system-call return values during replay to return values during trace collection are generated (*i.e.*, the semantic accuracy of the replay), providing indications of possible underconstraint.

4.3.4 Emulation

Supporting cross-platform replay is challenging, as every UNIX-like platform has its own slightly distinctive API for accessing the file system. For such system calls, there are usually near equivalents on other platforms, but occasionally a call provides a unique primitive. In order to support such calls, ARTC converts them to pseudo-calls. During replay, ARTC emulates pseudo-calls by using the most similar system calls available, sometime executing multiple calls on the target system to emulate a single call on the source system.

ARTC performs emulation for 19 different calls. 11 of these cases are for special metadata-access APIs (*e.g.*, extended attributes); not only do the names of the calls differ in these cases, but some systems support parameters and options not supported by others. When emulating these calls, we simply ignore such parameters.

Another three cases pertain to file-system hints; in particular, prefetching, caching, and preallocation hints are all treated slightly differently on each platform. Linux, Mac OS X and Illumos generally offer equivalent functionality, though sometimes via different APIs; emulation for these is straightforward. On FreeBSD, however, we simply ignore some of these calls where analogous APIs are not available. Three more emulations are required for obscure, undocumented Mac OS X system calls, that appear to be metadata related and are hence emulated with small metadata accesses.

Another case addresses a difference in `fsync` semantics on different systems. Linux file systems typically force data to persistent storage when `fsync` is called, but on Mac OS X semantics are different, and data is merely flushed to the device, which may cache it in volatile memory; `fcntl(F_FULLFSYNC)` is necessary to achieve true durability. When replaying traces collected from Linux on a Mac, a replay option determines which semantics are used to emulate `fsync`.

The final case is the `exchangedata` call, a unique atomicity primitive provided by Mac OS X. Given two files, `exchangedata` performs an atomic swap such that each file's inode points to the other file's data, preserving inode numbers and other metadata. Although there is no truly atomic equivalent on other platforms, we emulate this via a `link` and two `renames`.

5 Evaluation

We evaluate ARTC by establishing its preservation of semantic correctness and comparing its performance accuracy with a set of simpler strategies.

The simplest approach we compare against is *single-threaded* replay, which issues all calls in the trace from a single replay thread in the same order in which they were issued in the trace. This approach precludes not only reordering but also any concurrency between system calls. *Temporally-ordered* replay also issues calls during replay in the order they were issued during tracing, but uses one replay thread per traced thread, so calls that overlapped during tracing may be issued concurrently during replay. While it permits some concurrency, this approach allows no real reordering to occur during replay. *Unconstrained* replay falls at the opposite end of the ordering spectrum, employing multiple threads but enforcing no synchronization between them. This allows maximal reordering (within the constraints of `thread_seq`, which is still implicitly enforced) but is vulnerable to race conditions involving shared resources.

5.1 Semantic Correctness: Magritte

We evaluate the semantic correctness of ARTC's replay by examining its behavior with 34 traces of Apple's iLife and iWork desktop application suites [7]. The complex inter-thread dependencies and frequent metadata accesses found in these traces make them an excellent correctness stress test. We also believe these traces are useful beyond this evaluation, and so we release the compiled traces as a new benchmarking suite called Magritte[1]. Before presenting the results, we de-

[1]Magritte is named for a Belgian artist who created a number of paintings prominently featuring apples, most notably *The Son of Man*.

Trace	UC	**ARTC**	Events
iPhoto			
start400	74	**2**	35K
import400	377K	**7**	827K
duplicate400	53K	**2**	210K
edit400	881K	**2**	1660K
delete400	298	**2**	472K
view400	76K	**2**	278K
iTunes			
startsmall1	3	**0**	5.5K
importsmall1	1.5K	**0**	10K
importmovie1	56	**0**	5.3K
album1	549	**0**	9.7K
movie1	2.6K	**0**	9.5K
iMovie			
start1	43	**2**	21K
import1	4.4K	**7**	35K
add1	51	**3**	24K
export1	4.5K	**5**	42K
Pages			
start15	4	**4**	13K
create15	36	**4**	16K
createphoto15	401	**4**	56K
open15	4	**4**	15K
pdf15	4	**4**	15K
pdfphoto15	106	**4**	54K
doc15	4	**4**	15K
docphoto15	139	**4**	205K
Numbers			
start5	0	**0**	10K
createcol5	59	**0**	15K
open5	0	**0**	12K
xls5	0	**0**	14K
Keynote			
start20	0	**0**	17K
create20	269	**0**	36K
createphoto20	733	**2**	38K
play20	0	**0**	28K
playphoto20	208	**0**	30K
ppt20	4	**0**	51K
pptphoto20	4	**0**	126K

Table 3: Replay failure rates. *The number of event-replay failures in each trace is shown for a completely unconstrained multithreaded replay (UC) and for ARTC, both in AFAP mode. The Events column shows the total number of replayed actions in the trace.*

scribe some of the difficulties we encountered in the process of replaying these traces:

Special files: Some of the traces include reads from `/dev/random`, which resulted in very slow reads on Linux (tens of seconds for less than a hundred bytes of data). On Mac OS X, `/dev/random` is a non-blocking source of random bytes, whereas on Linux, reads from `/dev/random` block when the kernel judges that its entropy pool is depleted. We solve this by creating `/dev/random` as a symlink to `/dev/urandom`, which does not block, when replaying on Linux.

External bugs: We encountered some behaviors on Mac OS X that appear to simply be kernel bugs. Calling `close` on a file descriptor returned from `shm_open`, for example, consistently reports failure with `EINVAL`, which is not listed in its documentation. Interestingly, the call appears to succeed, since subsequent `opens` then return file descriptors re-using the same value. ARTC generally outputs warnings when replayed calls do not conform to its expectations, but sometimes suppresses them in cases such as this.

Missing trace details: There are a handful of sequences in the traces for iTunes that show system calls of the form `open(path, O_CREAT|O_EXCL)` executing successfully, but at points where prior events in the trace would indicate that `path` should already exist. While we cannot be entirely sure of the cause of this, it may be due to a mistake in the collection of the traces from the original applications. ARTC handles these by simply replaying them without the `O_EXCL` flag.

After addressing these issues, we replayed the traces with each of the four modes. In order to amplify concurrency and best exercise each mode's enforcement of the trace's semantics, we performed these replays in AFAP mode on an SSD-backed ext4 file system, and did not clear the system page cache between each benchmark's initialization and execution. Table 3 shows the number of errors in trace replay for unconstrained mode (UC) and ARTC; with the exception of `iphoto_edit400`, the failure counts for single-threaded and temporally-ordered modes (not shown) are identical to those of ARTC on all traces. The reported error counts are the maximum number of errors across five runs.

Although unconstrained replay is semantically correct when replaying some traces (*e.g.*, `keynote_start20`), many replays produce thousands of errors; on `iphoto_edit400` over half the trace's events replay incorrectly. Not only are the failure rates for ARTC and the other highly constrained modes several orders of magnitude lower, further investigation reveals that almost none of ARTC's errors are due to invalid reordering. Rather, except for four failures in `iphoto_import400`, all of ARTC's failures are due to a lack of extended attribute initialization information in the iBench traces; replay initialization thus does not create these attributes, and replayed calls attempting to access them fail. The four failures caused by reordering in `iphoto_import400` are due to an edge case involving a directory rename un-breaking a broken symlink, which ARTC's file-system model does not currently handle, causing it to miss some path dependencies and thus allow some invalid reorderings.

Given the unconstrained mode's extreme error rate, we do not consider it a viable option, and thus do not consider it in the remainder of our evaluation. We do not use Magritte for the performance accuracy aspect of our evaluation because the workloads are interactive and thus not consistently I/O-bound, an operating mode ARTC does not focus on modeling accurately.

5.2 Performance Accuracy

Here we employ micro- and macro-benchmarks to evaluate ARTC's performance accuracy, which we find is substantially better than that of the simpler single-threaded and temporally-ordered replay methods.

5.2.1 Microbenchmarks

In this section, we use microbenchmarks to explore interaction effects between workloads and storage systems, showing how each naturally affects the other. In one experiment, we adjust the degree of parallelism in the workload and show how the storage system takes advantage of the additional flexibility offered by increased queue depths. In three further experiments we construct feedback loops, changing aspects of the storage system

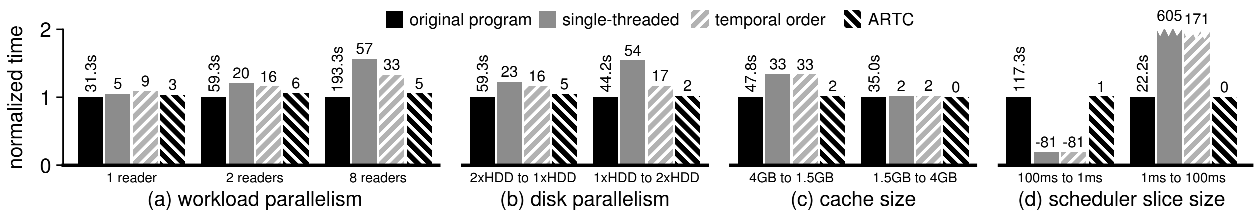


Figure 5: Microbenchmarks. *Effect of feedback loops on accuracy. Labels on the solid black bars indicate timings for the original program on the target system. Labels on other bars indicate a percentage error relative to the original.*

in ways that should change the workload's behavior. We experiment with varying disk parallelism, cache size, and I/O scheduler slice size. We show that in each of these scenarios ARTC adapts in a natural way, but the simpler single-threaded and temporally-ordered replay methods do not.

Workload parallelism: For our first experiment, we wrote a simple program the spawns a variable number of threads, each of which reads 1000 randomly selected 4KB blocks from its own 1GB file. We ran and traced the program with 1, 2, and 8 threads. We then performed single-threaded, temporally-ordered, and ARTC replays of each trace. The timing results for the three traces are indicated by the three groups of bars in Figure 5(a). Within each group, the first bar indicates the time it takes the original program to run, and the next three bars indicate how long each of the replay methods take. If replay is accurate, the bars in each group will be similar in size to the first bar of the group.

Figure 5(a) shows that going from 1 to 2 readers increases execution time from 31.3s to 59.3s, slightly less than double. Going from 1 reader to 8 performs 8× as much I/O, but execution time increases only 6.2×, to 193.3s. The sub-linear slowdown is due to the increased I/O queue depths of the more parallel workload giving the I/O scheduler and disk more freedom to optimize access patterns, increasing average throughput. These optimizations change the order in which I/O requests complete, which in turn affects the subsequent pattern of requests issued by the program. ARTC's replay adapts to these optimizations similarly, and thus achieves a mere 5% error in elapsed time on the 8-thread workload. The simpler replay methods, however, are not so flexible, and thus overestimate elapsed time by 57% and 33%.

Disk parallelism: Here we compare accuracy when tracing on a single-disk source and replaying on a two-disk RAID 0 target with a 512KB chunk size (and *vice versa*). We use the same simple program as above, running with two threads. Figure 5(b) shows ARTC is accurate moving in either direction (2-5% error), and temporal ordering achieves accuracy similar to the 2-thread case of Figure 5(a), but single-threaded replay does significantly worse when replaying the single-disk trace on the RAID, as its serial nature renders it incapable of exploiting the array's increased I/O parallelism.

Cache size: The program for this experiment has two threads and is similar to the previously used program with one difference: thread 1 sequentially reads its entire file before entering the random-read loop. For both tracing and replay, we use a two-disk RAID 0 and 4GB of memory. To limit the cache size during tracing and replay, we run a utility that simply pins 2.5GB of its address space in RAM, leaving only 1.5GB for the cache and other OS needs. The results of tracing with a normal cache and replaying with a small cache (and *vice versa*) are shown in Figure 5(c). ARTC is accurate for both source/target combinations, but the simpler methods are accurate only for replay on the 4GB target, producing timings that are 33% too long for the 1.5GB target.

In the trace collected on the 4GB system, thread 1's random reads are all cache hits, and thus all finish long before the vast majority of thread 2's reads are issued. On a target with a 1.5GB cache, most of thread 1's reads become cache misses, but the simple replay methods wait for thread 1 to finish before issuing most of thread 2's requests; this prevents the system from taking advantage of the parallelism provided by the RAID. In the other direction (1.5GB source to 4GB target), the simple replay methods are accurate. This accuracy asymmetry arises because when replaying the 1.5GB source system's trace on the 4GB target, all of thread 1's random reads are cache hits, so playing them at the wrong time does not degrade performance.

Scheduler slice size: Here we tune Linux's Completely Fair Queuing (CFQ) I/O scheduler to explore a tradeoff between efficiency and fairness. The CFQ scheduler implements anticipation [9] by giving threads slices of time during which requests are serviced. A large slice means the scheduler will attempt to increase throughput by servicing many requests from the same thread before switching to a different thread, at the cost of increasing the latencies seen by other threads. The length of these slices can be adjusted by tuning the scheduler's `slice_sync` parameter; we experiment with values of 1ms and 100ms. In our microbenchmark program, two threads compete for I/O throughput, each performing sequential 4KB reads from separate large files. Figure 5(d) shows that both simple replays dramatically overestimate performance when decreasing `slice_sync` from 100ms to 1ms, and even

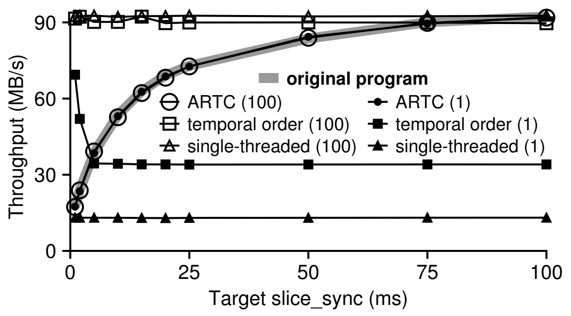
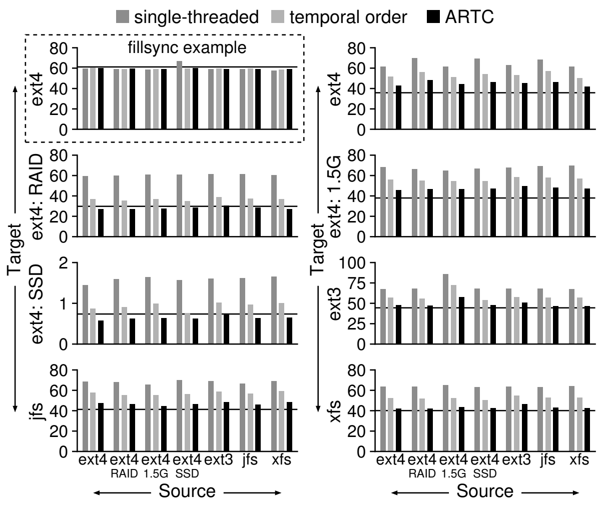
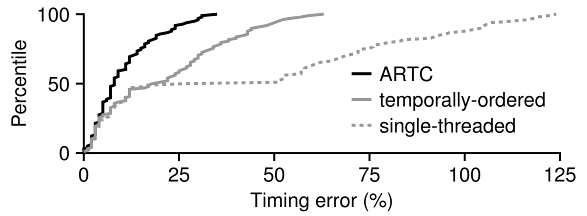


Figure 6: Varying anticipation. *Throughput achieved by executions with varying `slice_sync` values. Performance is shown for the original program and three replays of two traces (source `slice_sync` values of 1ms and 100ms).*



(a) Individual replay timing errors.



(b) CDF of all timing errors.

Figure 7: LevelDB replay combinations. *The first plot of (a) shows an example of `fillsync` timings. The remaining seven show the timings for every `readrandom` source/target combination. On each plot, a baseline shows how long the original program runs on the target platform. If replay is accurate, the bars will be near this line. (b) shows a CDF of the timing errors for the 98 replays for each mode.*

more drastically underestimate it when moving in the opposite direction. ARTC, however, is extremely accurate in both scenarios.

Figure 6 shows the inaccuracy of the simpler replays in greater detail, comparing the original program's performance to each of the three replays on both 100ms and 1ms traces. While ARTC predicts the performance of the target system flawlessly, the simple replay methods tend to predict timings that reflect the performance of the source system rather than that of the target. When a trace is collected with a large `slice_sync`, it will show long periods of time servicing requests from a single thread. During replay, even with a smaller slice, a simple replay method will only submit requests from the thread that dominated that period; this effectively reproduces the source system's scheduling decisions at the application level on the target.

5.2.2 Macrobenchmarks

In this section, we stress ARTC's ability to make accurate timing predictions by tracing and replaying the file I/O of an embedded database, LevelDB, on 49 different source/target combinations. We explore various file systems (ext4, ext3, JFS, and XFS) and hardware configurations (HDD, 2-disk RAID 0, small cache, and SSD). For each combination, we compare ARTC against single-threaded and temporally-ordered replay, as in §5.2.1. We run two benchmark workloads distributed with LevelDB, `fillsync` and `readrandom`, each with 8 threads; `fillsync` threads insert records into an empty database, and `readrandom` threads randomly read keys from a pre-populated database.

Figure 7(a) shows performance accuracy results for one source/target combination with `fillsync` (results for other combinations are similar), and every combination with `readrandom`. For `fillsync`, all replay modes on all source/target combinations are very accurate. When multiple LevelDB threads want to issue writes, all writes are issued by one thread; the others simply hand off their data to it. This essentially reduces the I/O pattern to a simple single-threaded write workload, so simple replay methods are not at a disadvantage. For `readrandom`, however, both simple methods significantly overestimate execution time in every case. ARTC sometimes overestimates and sometimes underestimates, but its errors tend to be much smaller.

Figure 7(b) shows the distribution of timing errors across all replays. ARTC does best at avoiding extreme inaccuracy; among the least accurate 10% of each method's replays, ARTC averages 28.7% error, compared to 52.9% for temporal ordering and 113.3% for single-threaded replay. Across all replays, temporal ordering and single-threaded replays achieve mean timing errors of 21.3% and 43.5%, respectively, whereas ARTC's replays average within 10.6% of the original program's execution time.

Simple replay methods overestimate `readrandom`'s execution time due to a lack of ordering flexibility, as

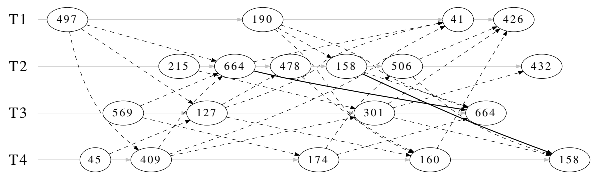


Figure 8: LevelDB dependency graph. *A directed graph showing replay dependencies enforced by ARTC's resource-aware ordering (bold) and temporal ordering (dashed). Solid grey edges indicate thread ordering; thus each row of nodes represents a thread. The ordering of the nodes in the horizontal direction is based on their ordering in the original trace. All calls in this window of time are* `preads`*; each node is labeled with the number of the file descriptor accessed by the call.*

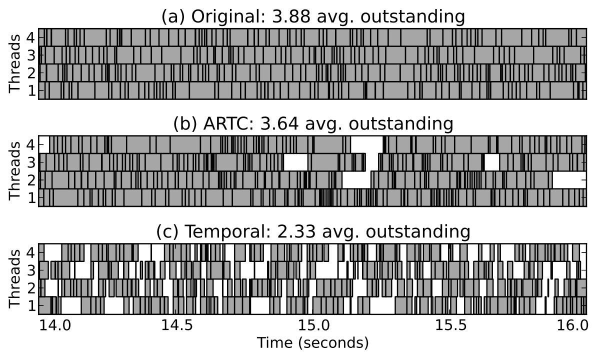


Figure 9: Concurrency. *System-call overlap achieved by different replays of a 4-thread LevelDB* `readrandom` *trace.*

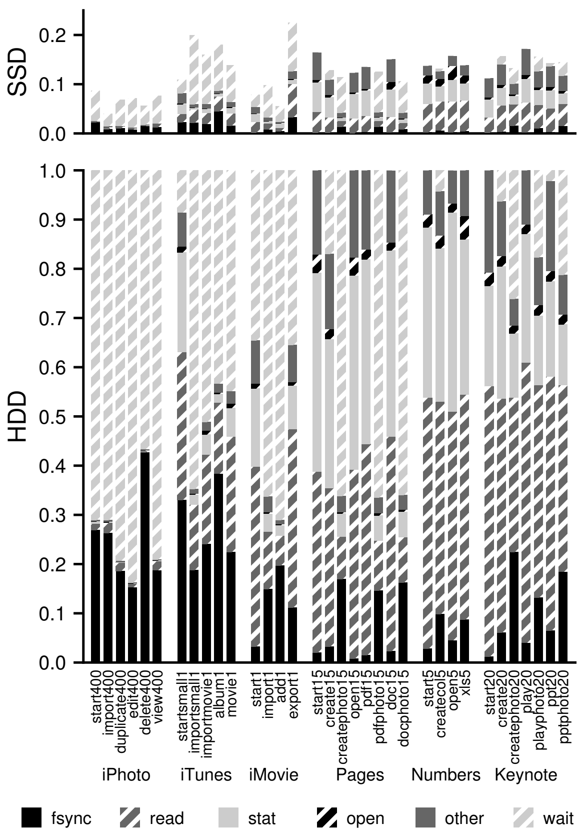


Figure 10: Operation times: *SSD vs. HDD.*

shown in Figure 8, a dependency graph of a representative period of time in a trace of a 4-thread LevelDB `readrandom` workload. Note that there are many more ARTC resource-dependency edges than are shown in this subgraph; however, these edges tend to be between nodes (system calls) that are separated by a long period of time and thus do not fit in the window of time shown here (only edges whose endpoints are both within that span of time are included). Over the entire trace, there are 9135 temporal-ordering edges and 6408 ARTC edges. However, what gives ARTC's replay its flexibility is not having slightly *fewer* dependency edges, but much more importantly having far *longer* edges. Measured in time between calls in the original trace, the average temporal-ordering edge is 10ms, whereas ARTC's average edge length is 8.9 seconds.

Figure 9 shows how enforcing the edges in Figure 8 affects when requests are issued during replay. Representative two-second samples are shown for the original program, ARTC replay, and temporally-ordered replay in parts (a), (b), and (c), respectively. For each subfigure, each of the four threads is represented by a row, with grey rectangles indicating spans of time spent in system calls issued by those threads. We observe that in the original program, each thread almost always has an outstanding request, giving the scheduler and disk plenty of flexibility. The replays deviate from this in that some gaps between system calls are visible where the replay threads spent time waiting for ordering dependencies to be satisfied. ARTC, however, shown in Figure 9(b), suffers far fewer such stalls than the temporally-ordered replay shown in Figure 9(c), achieving 94% of the system-call concurrency shown in Figure 9(a), in contrast to temporal ordering's 60%.

6 Case Study: Magritte

Here we demonstrate the use of the Magritte benchmark suite to evaluate the relative performance characteristics of two storage systems, using ARTC's detailed output to determine what types of operations dominate thread-time during replay. Thread-time is a measure of time used by individual threads, and will usually be greater than wall-clock time since threads typically run concurrently (for example, two threads running concurrently for two seconds yields four thread-seconds). Figure 10 shows a breakdown of how thread-time is spent when replaying on a disk and an SSD. Both times are normalized to HDD thread-time.

The SSD plot indicates a thread-time speedup of 5×-20× for most applications. Many of the categories with a significant presence for the HDD experiments also have a significant presence on the SSD; however, time spent waiting for `fsync`s is much less significant.

The applications each show distinct patterns. When run on disk, thread time in iPhoto and iTunes tends to be dominated by `fsync`; Numbers and Keynote, on the other hand, are dominated by `read`s and `stat`-family calls (*e.g.*, `stat`, `lstat`, *etc.*). iMovie and Pages are divided across a greater number of categories.

7 Related Work

The use of a compiler to transform input traces prior to replay is somewhat similar to previous work by Joukov *et al.* [10]. ARTC, however, is more focused on trace analysis and inferring event dependencies.

In other work on I/O trace replay, Anderson *et al.* argue for maximum accuracy, since even slight deviations can produce significant behavioral changes [3]. Tarasov *et al.*, however, argue for merely approximate replay based on general workload characteristics [22]. Our work falls somewhere in between: we replay the exact I/O set in the original trace, though we allow variations in ordering, much like real multithreaded applications. While ARTC may not necessarily produce exactly the same behavior from one run to the next, it more realistically emulates the behavior of real applications (which are likewise not necessarily consistent across runs).

Different approaches have been suggested for mining information from traces. Aguilera *et al.* perform statistical analysis on passively-collected RPC traces to infer inter-call causality [2] for debugging purposes. Mesnier *et al.*'s //TRACE actively collects traces, perturbing the I/O in order to deduce dependencies between operations [16], and incorporates this information into its replay. ROOT also attempts to infer dependency information from traces, but we rely on hints to glean as much information as possible from a single data point.

SCRIBE [12] is a replay tool that also partially orders replay events based on resources. Unlike ARTC, SCRIBE is oriented more toward debugging and diagnostics than performance analysis, and thus aims for perfect reproduction of applications' in-memory state. This necessitates intricate platform-specific kernel instrumentation for tracing and replay (which must be done on the same platform), whereas ARTC operates purely with system calls, allowing cross-platform replay and simple trace collection with existing tools.

8 Conclusion

The problems faced by real systems should be a primary motivation for systems innovation. Detailed traces can provide insight into these problems, yet high-quality traces of production systems remain scarce. Furthermore, the improvements made possible by new system-design ideas should be measured against real-world production workloads, yet the nondeterminism of modern programs makes accurate trace replay challenging. We propose ROOT, a new approach to trace replay based on the idea that even a single trace can provide hints about the nondeterministic structure of a program. This strategy helps to maximize the utility of scarce trace data.

We hope the ROOT approach will be applied to traces for a variety of storage APIs. We have created a new tool, ARTC, that applies ROOT to UNIX system-call traces, automatically generating realistic benchmarks. ARTC supports over 80 different system calls, using novel emulation techniques where necessary to support cross-platform replay. Its replay combines faithful reproduction of trace semantics with accurate performance predictions. Furthermore, we apply ARTC to 34 traces of Apple desktop applications to create Magritte, a new benchmark suite.

While ARTC is sufficiently robust and featureful to be generally useful, further developments remain for future work. Other possible resource dependencies would allow more fine-grained ordering constraints. For example, analysis of dependencies on file size rather than mere existence would allow a replay mode for file resources somewhere between stage and sequential ordering in strength.

ARTC and Magritte are available for download at: `https://research.cs.wisc.edu/adsl/Software/artc`

Acknowledgments

We thank the anonymous reviewers and our shepherd Bryan Ford for their feedback, which helped improve the quality of the paper enormously. This material was supported by funding from NSF grants CNS-1218405 and CNS-1319405, as well as generous support from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Tyler Harter is supported by the Facebook Fellowship and NSF Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *FAST '09*, San Jose, CA, February 2009. USENIX Association.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP '03*, Bolton Landing, NY, Oct. 2003. ACM.

[3] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In *FAST '04*, San Francisco, CA, April 2004. USENIX Association.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *SOSP '91*, Pacific Grove, CA, Oct. 1991. ACM.

[5] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *LISA '03*, San Diego, CA, Oct. 2003. USENIX Association.

[6] G. R. Ganger and Y. N. Patt. Using System-Level Models to Evaluate I/O Subsystem Designs. *IEEE Transactions on Computers*, June 1998.

[7] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *SOSP '11*, Cascais, Portugal, Oct. 2011. ACM.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, Feb. 1988.

[9] S. Iyer and P. Druschel. Anticipatory Scheduling: a Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *SOSP '01*, Banff, Canada, Oct. 2001. ACM.

[10] N. Joukov, T. Wong, and E. Zadok. Accurate and Efficient Replaying of File System Traces. In *FAST '05*, San Francisco, CA, Dec. 2005. USENIX Association.

[11] M. Kluge, A. Knüpfer, M. Müller, and W. E. Nagel. Pattern Matching and I/O Replay for POSIX I/O in Parallel Programs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09. Springer-Verlag, 2009.

[12] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS '10*, New York, NY, June 2010. ACM.

[13] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX '08*, Boston, MA, June 2008. USENIX Association.

[14] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang. CloudProphet: Towards Application Performance Prediction in Cloud. In *SIGCOMM '11*, Toronto, Canada, Aug. 2011. ACM.

[15] J. May. Pianola: A Script-based I/O Benchmark. In *Petascale Data Storage Workshop*, November 2008.

[16] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *FAST '07*, San Jose, CA, Feb. 2007. USENIX Association.

[17] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing. *Journal of Parallel and Distributed Computing*, Aug. 2009.

[18] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-driven Analysis of the UNIX 4.2 BSD File System. In *SOSP '85*, Orcas Island, WA, Dec. 1985. ACM.

[19] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, San Diego, CA, June 2000. USENIX Association.

[20] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads. In *FAST '09*, San Francisco, CA, Feb. 2010. USENIX Association.

[21] K. A. Smith and M. I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *SIGMETRICS '97*, Seattle, WA, June 1997. ACM.

[22] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting Flexible, Replayable Models from Large Block Traces. In *FAST '12*, San Jose, CA, February 2012. USENIX Association.

[23] W. Vogels. File System Usage in Windows NT 4.0. In *SOSP '99*, Kiawah Island Resort, SC, Dec. 1999. ACM.

[24] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of Backup Workloads in Production Systems. In *FAST '12*, San Jose, CA, Feb. 2012. USENIX Association.

[25] N. J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of Application Workloads from Network File Traces. In *FAST '09*, San Francisco, CA, Feb. 2010. USENIX Association.