TARDIS: Affordable Time-Travel Debugging in Managed Runtimes

Earl T. Barr
University College London
e.barr@ucl.ac.uk

Mark Marron
Microsoft Research
marron@microsoft.com

Abstract

Developers who set a breakpoint a few statements too late or who are trying to diagnose a subtle bug from a single core dump often wish for a time-traveling debugger. The ability to rewind time to see the exact sequence of statements and program values leading to an error has great intuitive appeal but, due to large time and space overheads, time-traveling debuggers have seen limited adoption.

A managed runtime, such as the Java JVM or a JavaScript engine, has already paid much of the cost of providing core features — type safety, memory management, and virtual IO — that can be reused to implement a low overhead time-traveling debugger. We leverage this insight to design and build *affordable* time-traveling debuggers for managed languages. TARDIS realizes our design: it provides affordable time-travel with an average overhead of only 7% during normal execution, a rate of 0.6 MB/s of history logging, and a worst-case $0.68s$ time-travel latency on our benchmark applications. TARDIS can also debug optimized code using time-travel to reconstruct state. This capability, coupled with its low overhead, makes TARDIS suitable for use as the *default* debugger for managed languages, promising to bring time-traveling debugging into the mainstream and transform the practice of debugging.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids

Keywords Time-Traveling Debugger, Managed Runtimes

"Debugging involves backwards reasoning"
Brian Kernighan and Rob Pike
The Practice of Programming, 1999

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660209

1. Introduction

Developers spend a great deal of time debugging. Skilled developers apply the scientific method: they observe a buggy program's behavior on different inputs, first to reproduce the bug, then to localize the bug in the program's source. Reproducing, localizing, then fixing a bug involves forming and validating hypotheses about the bug's root cause, the first point at which the program's state diverges from the intended state. To validate a hypothesis, a developer must halt program execution and examine its state. Especially early in debugging, when the developer's quarry is still elusive, she will often halt execution too soon or too late to validate her current hypothesis. Overshooting is expensive because returning to a earlier point in a program's execution requires re-running the program.

Although modern integrated development environments (IDEs) provide a GUI for halting program execution and examining program state, tool support for debugging has not changed in decades; in particular, the cost of program restart when placing a breakpoint too late in a program's execution remains. A "time-traveling" debugger (TTD) that supports reverse execution can speed the hypothesis formation and validation loop by allowing developers to navigate backwards, as well as forwards, in a program's execution history. Additionally, the ability to perform efficient forward and reverse execution serves as an enabling technology for other debugging tools, like interrogative debugging [31, 36] or automatic root cause analysis [12, 22, 27, 29].

Despite the intuitive appeal of a time-traveling debugger[1], and many research [4, 20, 30, 36, 54] and industrial [15, 21, 26, 47, 51] efforts to build TTD systems, they have not seen widespread adoption for one of two reasons. The first issue is that TTD systems can impose prohibitive runtime overheads on a debugger during forward execution: 10–$100\times$ execution slowdown and the cost of writing multi-GB log files. The second issue is long pause times when initiating reverse execution, the additional functionality TTD offers, which can take 10s of seconds. Usability research shows that wait times of more than 10 seconds cause rapid system aban-

[1] Time-traveling debuggers have been variously called omniscient, trace-based, bi-directional, or reversible.

donment and that wait times of more than a few seconds frustrate users and trigger feature avoidance [40]. Grounded on these results, we define an *affordable* time-traveling debugger as one whose execution overhead is under 25% *and* whose time-travel latency is under 1 second.

We present TARDIS, the first TTD solution to achieve both of these goals. On our benchmarks, forward debugging overhead averages 7% and time-travel latency is sub-second, at $0.68s$, as detailed[2] in Section 5. In short, TARDIS works to realize Hegel's adage that a sufficient quantitative change, in TTD performance, becomes qualitative change, in practical utility. TARDIS rests on the insight that managed runtimes, such as the .Net CLR VM or a JavaScript engine, have paid the cost of providing core features — type safety, memory management, and virtual IO — that can be reused to build an affordable TTD system. Thus, constructing TTD within a managed runtime imposes 1) minimal additional cost bounded by a small constant factor and 2) more fully realizes the return on the investment managed runtimes have made in supporting these features.

A standard approach for implementing a TTD system is to take snapshots of a program's state at regular intervals and record all non-deterministic environmental interactions, such as console I/O or timer events, that occur between snapshots. Under this scheme, reverse execution first restores the nearest snapshot that precedes a reverse execution target, then re-executes forward from that snapshot, replaying environmental interactions with environmental writes rendered idempotent, to reach the target.

Memory management facilities allow a system to traverse and manipulate its heap at runtime; type information allows a system to do this introspection precisely. Without these features, a TDD system must snapshot a process's entire address space (or its dirty pages), copying a variety of useless data such as dead-memory, buffered data from disk, JIT state, code, *etc.* [20, 30]. Managed runtimes provide both features. Section 3 shows how TARDIS exploits them to efficiently identify and snapshot only live objects and program state, thereby improving on the basic snapshot and record/replay TDD design. Section 3.2 leverages insights from work on garbage-collection to further reduce the cost of producing state snapshots. TARDIS is the first system, two decades after it was initially suggested [54], to implement piggybacking of snapshots on the garbage collector. Beyond this work sharing optimization, TARDIS also utilizes *remembered-sets* from a generational collector to reduce the cost of walking the live heap and a *write-barrier* to trap writes so it can snapshot only modified memory locations.

In addition to enabling affordable snapshots, managed runtimes also ease the capture of environmental interactions between snapshots. Managed runtimes virtualize resources by restricting how calls to native methods, including calls to the underlying OS APIs, can be made. Thus, by construction, managed runtimes provide a small instrumentation surface for adding functionality to record/replay non-deterministic environmental interactions, including file system, console, and network I/O. Section 3.4 describes how TARDIS captures these interactions using specialized library implementations. These implementations examine program state as well as memory layout to make decisions such as storing only a file offset to obviate seeking over a large immutable file, or converting environmental writes or operations that overwrite data on which nothing depends into NOOPs during replay. To ensure deterministic replay of multi-threaded programs, TARDIS, like other systems [30, 48, 51], multiplexes all threads onto a single logical CPU and logs the context switches and timer events. Section 3.3 explains how TARDIS uses efficient persistent [19] implementations of specialized data structures to virtualize its file system. This section also shows how to leverage standard library file semantics in novel ways to optimize many common file I/O scenarios, such as avoiding the versioning of duplicate writes and writes that are free of data dependencies.

This paper makes the following contributions:

- We present TARDIS, the first *affordable* time-traveling debugger for managed languages, built on the insight that a managed runtime has already paid the cost of providing core features — type safety, memory management, and process virtualization — that can be reused to implement a TTD system, improving the return on the investment made in supporting these features.
- We describe a baseline, nonintrusive TTD design that can retrofit existing systems (Section 3.1); on our benchmarks, this design's performance results — an average slowdown of 14% (max of 22%) and 2.2 MB/s (max of 5.6 MB/s) of logging — apply to any managed runtime and demonstrate the generality of our approach.
- We present an optimized design of TARDIS in Section 3.2 that we tightly integrated into a .Net CLR runtime to leverage its features and (optionally) additional hardware resources to reduce TARDIS's average runtime overhead to only 7% (max of 11%), record an average of 0.6 MB/s (max of 1.3 MB/s) of data, and achieve time-travel latency of less than $0.68s$ seconds on our benchmarks.
- We show how TARDIS can run heavily optimized code, so long as 4 lightweight conditions are met, by time-traveling to reconstruct source-level debugging information (Section 4) and how, thanks to its low overhead, TARDIS can make time-travel debugging an "always on", standard feature of debuggers (Section 5).

2. Time-Traveling Debuggers

This section demonstrates the generality of our design and approach. We first present the generic top-level algorithms

[2] All programs in our benchmark suite have live heaps under 20 MB. When load testing the runtime overhead reaches 25% at a live heap size of 50 MB.

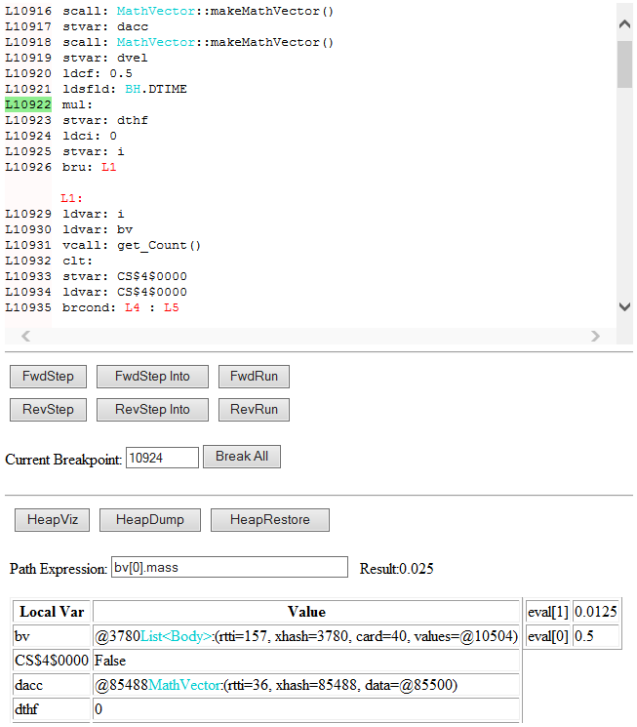


Figure 1. TARDIS running the BH n-body simulation benchmark. TARDIS provides support for the standard set of *forward step/step into/run* commands and data value inspection. The developer has set the breakpoint at line 10924, then used the *reverse step* to step backwards in "execution time", reversing the program's execution to line 10922.

for a *record-replay* time-traveling debugger, which we then instantiate in Section 3 to produce the TARDIS debugger. During forward execution, a record-replay TTD takes snapshots of the program state at regular intervals and logs external interactions, such as file or console I/O, that occur between these snapshots. To time-travel, we restore the nearest snapshot that precedes the time-travel target, then re-execute forward from that snapshot, replaying external interactions from the log, until reaching the target point in the program.

We define the algorithms in the context of a core procedural language shown in Figure 2. The language has a type system with the standard `bool`, `int`, and `float` types, along with pointer-referenced heap allocated record and array types. The stack-based execution model supports the usual set of operators. A program is a set of record type definitions and procedures. Procedures are either user-defined or one of the *builtin* operations that provide access to resources virtualized by the managed runtime.

2.1 Forward Execution with Snapshots and Logging

Algorithm 1 shows the generic implementation of the forward execution algorithm for record-replay TTD in a basic interpreter. The stack of call-frames *cs* maintains program execution state. Each call-frame *cf* is a tuple consist-

Global Vars		$g, \cdots$
Local Vars		$v, \cdots$
Fields		$f, \cdots$
Record	::=	$struct : \{f_1 : \tau_1, \cdots, f_k : \tau_k\}, \cdots$
Type τ	::=	`bool` | `int` | `float` | $struct*$ | $\tau[]$
Constant c	::=	$i \in \mathbb{Z}$ | $r \in \mathbb{R}$ | `true` | `false`
Procedure *Proc*	::=	$Proc(arg_1, \cdots, arg_k)\ block_1 \cdots block_j$
Block *block*	::=	$stmt_1 \cdots stmt_k$
Statement *stmt*	::=	*load* | *store* | *exp* | `cjmp` $block_t\ block_f$ | `ret` v | `call` *Proc* | `new` *struct* | `new` $\tau[]$
Load *load*	::=	`push` c | `load` v | `load` f | $\cdots$
Store *store*	::=	`store` v | `store` f | $\cdots$
Expression *exp*	::=	`nop` | `add` | `and` | `pop` | $\cdots$

Figure 2. Core imperative language.

ing of the current procedure, the current *pc* offset, the local variable-to-value map *vars*, and the current evaluation stack *eval*. The execution loop is standard: it switches on the current statement to dispatch the code implementing the operation. The TDD implementation alters only a few cases in a standard interpreter's main loop; we omit the other cases. The newly inserted TTD code is underlined and highlighted in blue (if color is available).

First we modify the initialization of the runtime (Line 1), which must set up the data-structures needed for the persistent I/O libraries and logging. Line 5 introduces *TraceTime*; we use this variable to impose a total order on the events logged during the program execution. The next line initializes and starts *SnapTimer* to control the snapshot interval. *DoSnapShot*() snapshots program execution state for use during time-travel and then resets the *SnapTimer*. Inserting checks for the snapshots at each loop head (line 15) and procedure call (line 23) bounds the time between the expiration of the *SnapTimer* and the next call to *DoSnapShot*(). Since one of the optimizations in Section 3.2 is to piggyback snapshot work on top of the GC, we also insert snapshot checks at each point where we may invoke the GC (line 28).

We must also ensure that (1) all logged events and snapshots obey a total order and that (2) every execution of a statement has a unique *TraceTime* timestamp. The first condition ensures that execution replay is deterministic. To enforce this condition, we advance the *TraceTime* whenever a builtin procedure, which can generate log data, may execute (line 17) or whenever a snapshot is taken. The second condition ensures that, during time-travel, we can use *TraceTime* to determine the correct visit to a target statement, of the possibly many visits during execution, on which to break. This condition is enforced by advancing *TraceTime* on every branch to a loop header (line 14) and every procedure call (line 17).

2.2 Capture and Replay for Time-Travel

To time-travel, we use Algorithm 2. This algorithm first finds the snapshot (*snp*) that is the nearest to and precedes the target trace time (*targettime*). Then it resets the globals and

Algorithm 1: ExecuteForward

```
   Input: Program p
1  Initialize(globals, allocator, iosystem(TTD));
2  cs ← new Stack();
3  vars ← InitLocals(main, ∅);
4  cs.Push(main, &main.block1[0], vars, ∅);
5  TraceTime ← 0;
6  SnapTimer ← new Timer(SNP_INTERVAL);
7  while cs ≠ ∅ do
8      cf ← cs.Top();
9      switch cf.StmtAtPC() do
10         case cjmp block_t block_f
11             block ← cf.eval.Pop() ? block_t : block_f;
12             cf.pc ← &block[0];
13             if IsLoopHeader(block) then
14                 TraceTime++;
15                 if SnapTimer.Expired then DoSnapShot();
16         case call Proc
17             TraceTime++;
18             if Proc is BuiltIn then
19                 ExecuteBuiltin(Proc, cf, globals);
20             else
21                 vars ← InitLocals(Proc, cf.eval);
22                 cs.Push(Proc, &Proc.block1[0], vars, ∅);
23                 if SnapTimer.Expired then DoSnapShot();
24             cf.pc++;
25         case new τ
26             DoGC();
27             TraceTime++;
28             if SnapTimer.Expired then DoSnapShot();
29             if τ is struct then
30                 cf.eval.Push(AllocStruct(τ));
31             else
32                 len ← cf.eval.Pop();
33                 cf.eval.Push(AllocArray(len));
34             cf.pc++;
35         ...
```

Algorithm 2: ExecuteReplay

```
   Input: Program p, Target (targetpc, targettime)
1  snp ← FindNearestPreSnapshot(targettime);
2  InitializeState(snp.globals, snp.cs);
3  InflateHeap(snp.serializedheap, snp.allocator);
4  iosystem.ReverseExecuteIOLogTo(snp.TraceTime);
5  TraceTime ← snp.TraceTime;
6  InReplayMode ← true;
7  while targettime ≠ TraceTime ∧ targetpc ≠ cs.Top().pc do
8      cf ← cs.Top();
9      withtrace ← (targettime − TraceTime < TRACE_RNG) ;
10     ReplayStmt(cf.StmtAtPC(), cf, withtrace);
11 InReplayMode ← false;
```

the call stack from the snapshot (*InitializeState*) followed by reinflating the heap state and resetting the the memory allocator as needed (*InflateHeap*). Next it reverts the I/O system to the snapshot time. After restoring the snapshot, the *InReplayMode* flag is set so that library functions can avoid re-executing non-idempotent operations, *e.g.* sending network packets, during replay.

During replay, we collect a full trace [4] to instantaneously respond to additional time-travel requests to targets after the restored snapshot. However, with overheads of 5–20×, enabling full tracing throughout replay would cause unpleasantly long replay times. Thus, we delay the start of trace collection until the replay is *close* to the *targettime* (line 9). Once at the time-travel target, *i.e.* the desired *targettime* and *targetpc*, the algorithm resets the replay mode flag and returns to the debugger interface.

3. Implementation of TARDIS

Here, we describe the design and implementation of TARDIS, which instantiates the generic record-replay TTD design from Section 2. TARDIS resides within the .Net CLR whose features it exploits and extends to enable low-overhead tracking of the needed program state and history information. We first present a nonintrusive implementation of the *DoSnapShot*() method in Section 3.1 that is suitable for use with almost any managed runtime. Section 3.2 then presents an optimized implementation of *DoSnapShot*() that fully exploits features in the host managed runtime and available hardware resources to reduce TTD overhead, at the cost of additional implementation effort or introducing hardware dependencies. Many programs extensively interact with the file system and do so predictably. Section 3.3 describes how we layer persistence onto files, track data dependency, and log file operations all to efficiently support time-travel across file operations. Section 3.4 presents the mechanisms TARDIS uses to capture, via API interposition when possible, other external state events. TARDIS, like any large engineering effort, has limitations; Section 6 discusses them.

3.1 Baseline Nonintrusive State Snapshots

The baseline nonintrusive implementation of *DoSnapShot*() minimizes changes to an existing managed runtime. In particular, it does not change (1) the allocator, (2) the memory layout of records or objects, or (3) the garbage collector. Thus, this formulation can retrofit, with minimal effort, existing managed runtime systems via small changes to the interpreter/compiler or via bytecode rewriting.

Given the buffer (or file) to write the snapshot into, Algorithm 3, the baseline nonintrusive snapshot algorithm, serializes globals and the current call-stack. Then, it serializes the heap and allocator state which, depending on the allocator, includes information on the max heap size, generation infor-

Algorithm 3: BaselineSnapshot

```
   Input: Buffer snapbuff
1  SerializeEnvironment(globals, cs, snapbuff);
2  SerializeAllocatorState(allocator, snapbuff);
3  worklist ← NonNullRootSet(globals, cs);
4  while worklist ≠ ∅ do
5      cptr ← worklist.Dequeue();
6      type ← TypeResolve(cptr);
7      Mark(cptr);
8      SerializeAddr(cptr, snapbuff);
9      SerializeMemory(cptr, sizeof(type), snapbuff);
10     foreach childptr ∈ Pointers(cptr, type) do
11         if childptr ≠ null ∧ !IsMarked(childptr) then
12             worklist.Enqueue(childptr);
```

mation, free list, *etc.* Line 3 initializes the worklist with the set of non-null roots for the live heap.

The main heap walk-copy loop, starting at line 4, is a standard copying traversal of the live heap. To process each pointer, it determines the referenced record or array type on line 6 and marks the pointer as *visited* on line 7. Next, the address (line 8) and then the contents of the record/array itself (line 9) are serialized into the buffer. Once this finishes, the walk-copy loop iterates over the set of all pointer fields, extracted by the *Pointers* function, for a record or array of the given type (line 10). Each non-null and unvisited *childptr* is added to the worklist. When the processing loop terminates, the entire state of the program has been suitably serialized for re-instantiation and re-execution in Algorithm 2.

3.2 Optimized Snapshots

The basic snapshot computation presented in Algorithm 3 makes few assumptions about, and requires minimal modifications to, the data-layouts and GC algorithms in the runtime system. This nonintrusive design simplifies the process of extending existing managed runtimes/debuggers with TTD functionality. However, it also means that the snapshotting algorithm cannot fully leverage a variety of features commonly found in managed runtime environments to reduce both the time to produce a snapshot and the amount of data that is serialized.

Opportunistic GC Piggybacking The purpose of the walk/-copy loop in Algorithm 3 is to compact the live heap memory into a contiguous range for efficient storage. However, if the runtime's memory allocator uses copying or compacting GC [28], then, after the GC call on line 26 in Algorithm 1, the live heap memory will already be compacted into a contiguous range. In this case, the call to *DoSnapShot*() on line 28 can skip the redundant heap walk-copy loop and directly output the live memory range from the GC[3]. Thus, the first system specific optimization we propose is to co-ordinate the GC and snapshot computation. To do so, we alter the GC collection trigger to

$$FreeMem = 0 \vee (FreeMem < \delta \wedge SnapTimer.Remaining < \epsilon).$$

This condition triggers the GC to run slightly sooner than it usually would, when δ bytes may remain in *FreeMem*; alternatively, the original GC trigger can be preserved by setting δ to 0. By forcing the collection early, we can, at low cost, piggyback a snapshot, which was due to happen within the next ϵ seconds anyway, on the collector run. If we cannot opportunistically piggyback on a GC run, *i.e.* plenty of *FreeMem* remains, we then fall back on the copy-walk loop in Algorithm 3.

Generational Optimizations In Algorithm 3, the baseline snapshot implementation, long-lived records or arrays may be visited and serialized multiple times. This increases both the cost of the heap walk-copy loop and the amount of data written. However, if the runtime system has a *generational* collector [28], we can eliminate much of the redundant visit/copy work; we can

1. Perform the walk-copy loop only on the *nursery* during an explicit heap walk for the snapshot; and
2. Use an *object*-based *write-barrier* [7] to track which objects in the *old space* were modified since the last snapshot and serialize only these objects.

The latter optimization has the immediate consequence that snapshots are partial and their restoration may need to load prior snapshots to get values for unchanged objects, increasing their cost. However, the time between gen-0 collections, which forces a full heap snapshot, bounds the number of previous snapshots needed to restore a snapshot. To avoid the need to fully expand and inspect every snapshot since the last full heap snapshot, we separate the old-space portion of the snapshot from the nursery and use the standard approach [20, 30, 42, 54] of keeping a (byte) map from the old-space entries to the snapshot which holds their value.

A common alternative to software write-barriers are write-barriers based on hardware page protection. The runtime can use hardware page protection to lazily snapshot the old space by (1) protecting all pages in the old space at snapshot time, then (2) lazily performing *copy-on-write* on pages when a fault occurs. In this paper, we opt for the object-based write-barrier leaving a comparison of these approaches for future work.

Snapshot Compression Producing snapshots at small time intervals, even with the previous optimizations, can easily produce 100s of MB of snapshot data for longer running programs. To reduce the cost of writing this data to disk, and the storage overheads on disk, we compress the heap snapshots before saving them. Previous work [49] has shown that it is possible to obtain large reductions in the amount of data written by running a standard lossless compression

[3] Any pinned objects or stationary objects, such as objects in a *large space*, will still need to be copied explicitly.

algorithm, such as GZip, on the live memory buffer. The cost of running the compression algorithm is minimized by offloading the compression and write operations to a helper thread when there are spare computational resources available, *e.g.*, an idle CPU core.

Dynamic Interval Adjustment Even with the optimizations above, the overhead of extracting snapshots at a fixed rate, `SNP_INTERVAL` on line 5 in Algorithm 1, may impose an unacceptable overhead. To control the cost of the TTD system and ensure its overhead remains near a targeted fraction of execution time $cost_{trgt}$, we can construct a basic *proportional controller* for the snapshot interval. Initially, we set the dynamic snapshot interval as $t_{snp} = \texttt{SNP_INTERVAL}$ and, to avoid pathologically low or high intervals, we enforce $t_{snp} \in [l, u]$ where l, u are lower and upper limits on the sample rate. At each snapshot, we use a timer, $cost_{est}$, to estimate the recent snapshot overhead and, approximately every second of wall clock time, we update the dynamic snapshot interval in the standard way as $t_{snp} = K * (cost_{trgt} - cost_{est}) + \texttt{SNP_INTERVAL}$. Based on our experience with the system, we select $cost_{trgt} = 0.1$ for an approximate target overhead rate of 10% and $K = -2.5$ for the gain value to balance stability and response as program behavior changes.

3.3 File System History

In a record-replay TTD system, the standard approach for handling external events is to log every value received from the external environment during forward execution and then, during replay, to read the needed values from this log. This approach is general and, for most event sources, has a low overhead. It does not scale, however, to high bandwidth external event sources, like the filesystem, where logging every byte read/written can quickly generate huge log files and unacceptable overhead. To account for this, we constructed a novel mechanism, based on the additional semantic information present in the runtime and standard library semantics, to efficiently handle filesystem operations.

Fundamentally, a program may interact with a file in one of three modes: read-only, write-only, or both read and write. We observe that (1) any data that a program reads from a file must already exist on the filesystem and (2) any values written are, by construction, deterministic during replay. These observations imply that reads and writes to a file, so long as they are not intermixed, do not need to be logged. When a file is exclusively read or written, then closed, and later reopened for writing, we simply copy the original file, before allowing writes to the file. During a single open, programs read from then overwrite locations in some files. Restricting full capture to these files allows us to eliminate file write logging for many common file access idioms, including processing data read from one file and writing the results to a second file or overwriting an existing file without reading its contents.

Algorithm 4: PersistentFileWriteBase

```
Input: PFile pf      // the persistent file to update
Input: long pos      // the file offset
Input: char c        // the byte to write
1  bpos ← pos / 512;
2  cpos ← pos % 512;
3  pb ← pf.map[bpos];
4  if pb = null ∨ pb.buff[cpos].Item1 ≠ 0 then
5      buff ← new Tuple⟨long,byte⟩[512];
6      pb ← new PBlock(buff, pf.map[bpos]);
7      pf.map[bpos] ← pb;
8  origc ← pf.f.ReadFromPos(pos);
9  pf.f.WriteAtPos(c, pos);
10 pb.buff[cpos] ← (TraceTime, origc);
```

Persistent File Content Implementation For the intermixed read-write case, we construct a *persistent implementation* [19] of the file that provides fine-grained access to any version of the file during a program's execution[4] and can track those parts of the file that represent program dependencies. In our construction, the primary on-disk copy contains the current file contents; the persistent structures contain the history of previous versions for each byte written. The following structures efficiently encode file contents, and read/write dependencies, as they appear at every step in the execution of the program.

```
PFile := { Dictionary<long, PBlock> map; File f; }
PBlock := { Tuple<long, byte>[512] buff; PBlock next; }
```

When a file is opened for reading and writing, *i.e.* the `FileAccess` flag is `ReadWrite`, we create a `PFile` entry for it with a map of all null entries; the map is a `Dictionary` because file writes are generally sparse. When TARDIS closes the file, it saves the `PFile` data into a temp directory. Writes to the file position *pos* may need to create a new `PBlock` entry to represent the old version of the file. Algorithm 4 depicts the basic algorithm for writing the persistent file structure. Algorithm 4 creates a new `PBlock`, writes *TraceTime* and the about-to-overwritten character from the file on disk into it, sets it to the head of the list at *cpos*, then overwrites character in the file.

A read converts a file offset, or position, into a block index as in the write algorithm. If the resulting `PBlock` list is null, we directly read from the underlying file. Otherwise we walk the list of `PBlock` entries. We return the value of the first `Pblock` whose the time is non-zero and larger than the current *TraceTime*. If we reach the needed block is not found, we read the desired data directly from the file.

By construction, the primary on-disk copy contains the current file contents. Thus, it is possible to perform time-

[4] An alternative implementation could use snapshots in a *copy-on-write* filesystem such as Btrfs, ReFS, or ZFS. However, these systems are block granular, not byte, which single-step TTD needs, and are unaware of application-level read/write dependencies, resulting in redundant copying.

travel debugging when multiple processes (even processes that are not TTD-aware) are accessing the same set of files simultaneously. Implementing this functionality to work with both TTD-aware and unaware processes requires support from the OS to notify each TTD application that a file modification is about to occur. This enables the TTD system to update (or create) the needed `PFile` structures and, when two processes are simultaneously reading/writing the same file, to make a copy of the original file.

As the file system we are using (NTFS) does not support such notifications, our implementation instead uses a simple file system broker which allows us to debug multiple TTD processes and non-TTD processes — that only read from shared files — simultaneously. Whenever a TTD process is about to modify a file, it notifies the broker, which generates a global modification time and sends a message, ⟨`char, pos, time`⟩, indicating that each TTD process should update its `PFile` information as needed. Extending the broker to push debugger step/reverse step commands to other TTD processes in file timestamp order enables the various TTD processes to remain synchronized during debugging, up to the reorderings of local process events that respect the order of file system events.

Dependency Optimization The conversion of the default implementation of the filesystem API into the persistent version enables the reconstruction of the state of the filesystem at any point in the execution of the program. The implementation described above may log many data-dependence-free writes in a file. We utilize the fact that we need to preserve the old contents of a file only if they were read. Thus, we update the `PFile` structure to track the upper and lower bounds of the file positions that program actually reads.

```
PFile := { long minpos, maxpos; ... }
```

Initially both values are set to -1. On read operations, we adjust the upper and lower bounds to include the *pos* that is read from. We update the write algorithm to first check if the target write *pos* is outside the range of used file positions $pos \notin [\,\texttt{minpos}, \texttt{maxpos}\,]$. If *pos* is outside the range, then it has never been read and the old value is simply overwritten.

We also change the conditional guarding the creation of a new `FBlock` to to check whether the new and old values are identical, $pb.buff[cpos].Item2 = c$. If the values are the same, then there is no need to update the persistent structure. This ensures that writes that store mostly unmodified data back to the file from which that data was read do not generate extensive updates to the `PFile` structure. This feature allows us to eliminate large amounts of logging for file access idioms such as exclusively appending new content to an existing file or reading data into a cache (perhaps modifying the data), then writing it out to disk as the cache fills.

Filesystem Operation Logging Guided by the high-level library semantics of the operation, a combination of logging and the creation of a set of temporary files can manage the majority of filesystem operations. For example, consider the standard directory operations and their extensions with logging for TTD:

```
Delete(string fn)                ⇒ Move fn to a fresh name.
CreateNew(string fn)             ⇒ Log creation of fn.
Copy(string src, string dst)     ⇒ Move dst to a fresh name.
Seek(File f, long pos)           ⇒ Log previous position.
Length(File f)                   ⇒ Log returned size.
Open(string fn, Mode m)          ⇒ Log size and, as needed,
                                   copy fn, create PFile.
Close(File f)                    ⇒ Save PFile as needed.
...
```

The logged information makes it easy to reverse each of these filesystem operations. For example, to reverse `Delete`, TARDIS simply copies the file back into place from the temporary file created to preserve its contents, while reversing writes to a file opened in `Append` mode simply requires resetting its file size. Thus, for each API call that may modify the file system, we add code to revert the effect of the operation using the data in the log.

3.4 Other External Interactions

TARDIS handles all other external events, such as interrupts, console interaction, network I/O, thread switches, *etc.*, via standard logging [30, 38, 55]. For each of these external events, we modify the standard library code to timestamp and append the event to the log. This log provides data for reverting file system state and deterministically replaying all the external inputs seen by the program during re-execution.

A concern with this approach is that it may entail instrumenting a potentially very large API surface. Such a large surface would create two problems: the large manpower investment to fully instrument the API and the fact that missing (or incorrectly handled) API calls would cause strange failures during time-travel. Identifying a suitable set of interface points ameliorates these problems by restricting the number of modifications need to capture all environmental interactions to a small fraction of the API space [44].

However, in a managed runtime, each external API call already imposes an implementation cost, *e.g.* ensuring memory management conventions are observed, and generally incur additional runtime costs. As a result, designers of managed runtime systems seek to minimize the number of these external APIs and have already invested substantial time into annotating and structuring the code that calls them. In our system, we used a custom implementation of the basic class libraries consisting of 72 external API calls which required 1215 lines of code. The additional code to enable TTD consists of 448 lines of additional replay and logging code. Thus, as an additional benefit of the API surface minimization and virtualization done during managed runtime construction, the TTD functionality can be implemented with little additional work, increasing the return on the substantial investment in building a managed runtime.

I/O events and NOOP-able calls During replay, input events, such as receiving a network packet or getting the current time, are implemented by reading and returning the needed data from the log. TARDIS converts output events that are non-idempotent or redundant, such as network sends or timer registration, into NOOPS and updates the state of the environment as needed, *e.g.* writing to the console (UI).

Threads and Asynchronous Events Scheduling related events, such as thread switches or interrupts that trigger event handlers, are handled via a *uniprocessor execution model* and the information in the event log. The uniprocessor model [30, 48, 51] multiplexes all threads on a single logical CPU (even if multiple CPUs are available). This ensures that there are no thread races during the program execution aside from those that are due to explicit context switches. Thus, replaying these explicit thread context switches recorded in the log ensures deterministic re-execution of the program.

Native Invocation APIs In addition to the native APIs that managed runtimes virtualize, some systems, including Java and C#, also expose mechanisms, like JNI and P/Invoke, that allow a managed program to to call arbitrary native methods. TARDIS requires that these native calls do not mutate environmental state, such as files or UI, that are also accessed from within the managed runtime. This restriction prevents side-effects to the external state of which the TTD system is unaware. Our implementation supports native calls using the standard set of C# *blittable* types, `Byte`, `Int16`, `Int32`, *etc.*, as well as opaque native pointers `IntPtr`.

During normal execution, the result values of native invocations are stored in the event log. When we take/restore snapshots we cannot serialize or restore the data that the native `IntPtr` values reference. At snapshot time, we ignore the memory pointed to by these references; at restore time, we save the native pointer values into an array for access during replay. Native invocations become "NOOP"s during replay that return the value produced during their forward execution from the log. Thus, TARDIS can time-travel over native invocations at the cost logging their return values during forward execution.

4. Debugging Optimized Code

Thus far we have assumed that TARDIS is debugging a *debug build* of a program which guarantees that, at each step in the execution, the values in the home locations of local and global variables are consistent with the source-level view of the program. This is a common assumption for debuggers and limits the optimizations a compiler can perform, since it requires, among other things, the preservation of all assignments in order. For debug builds, the compiler therefore produces slower code (frequently by 20% or more) than an optimized build. TARDIS is not limited to debug builds: assuming the compiler is correct, it enables the full-fidelity source-level debugging of a program, that a debug build p_d provides, while actually executing an optimized build p_o. To accomplish this feat, TARDIS

1. Executes the optimized build p_o and collects state snapshots and the execution log as described in Algorithm 1 and Section 3;
2. Launches the debugger and its replay system when encountering a breakpoint or exception; and
3. Replays logs starting from snapshots taken from the optimized execution against p_d, the debug build, instead of the optimized build p_o, in Algorithm 2.

These steps allow us to run the optimized build until a breakpoint or an exception, when replay reconstructs the full source-level view of the execution state in the debug build. The key to this approach is ensuring that we can replay a snapshot and an event log from optimized code against its debug build.

Snapshot Safe Points Algorithm 1 introduces three classes of *snapshot points*: loop headers, procedure calls, and allocation operations, where snapshots are conditionally taken. To swap snapshots and logs between different versions of a program, their executions must be convertible. We can safely use a snapshot from the optimized build p_o taken at a *snapshot safe point* in the debug build p_d of a program if:

1. Calls to builtin procedures and *TraceTime* operations are not reordered or moved over any snapshot point in p_o;
2. The arguments to all builtin procedure calls in p_o and p_d are identical;
3. Stores to the heap are not removed or moved over any snapshot point in p_o; and
4. At snapshot safe points, and for any procedure call location that may transitively reach the safe points, the compiler has constructed a *recovery map* [24] for each local and global variable in p_d based on state values in p_o.

The first two conditions ensure that the logs are identical, and that the *TraceTime* timestamps are synchronized; they prevent any reordering of or argument changes to any operation that writes to the event log or updates the *TraceTime* timestamp. The third condition ensures that all heap objects are identical across the optimized snapshot and its corresponding snapshot in the debug build. We could force the optimized build to store all locals and globals to their home locations before each snapshot safe point, but this would inhibit many important optimizations — copy propagation, loop-invariant hoisting, *etc.* Instead, the final condition allows the compiler to perform optimizations that may result in a location not containing the needed variable value as long as it provides a function to reconstruct the correct value for the variable, much like GC *stack maps* [28].

The compiler must enforce these conditions so we rely on its correctness and cannot debug the compiler itself. Under these assumptions, however, we can guarantee that snapshots

Optimization	Level	Permitted
Inline statically resolved calls	O0	Yes
Inline object/array allocation	O1	Yes
Inline virtual calls with guard	O1	Yes
Eliminate null/bounds checks	O1	Yes
Constant Propagation	O1	Yes
Copy Propagation	O1	Yes
Common subexpression Elimination	O1	Yes
Create hot traces in CFG based on static heuristics	O1	Yes
Convert fields in non-escaping objects (arrays) into local variables.	O1	Partial
Convert field loads to variable reads inside basic-blocks	O1	Yes

Table 1. Optimizations used in Jikes Java RVM 2.1.1 compiler on O1 optimization level and their compatibility with TTD-based debugger state reconstruction.

from a program's optimized build can be used in its debug build. To better understand the range of permissible optimizations, Table 1 shows optimizations that the Jikes Java RVM 2.1.1 [3] compiler enables at its O1 optimization level. These optimizations only increase compilation time slightly while producing very efficient executables [33].

State reconstruction in TTD-based debuggers permits all but one of the optimizations in Table 1. The conversion of fields to local variables (often called *scalar replacement of aggregates* [39]) eliminates objects that are allocated on the heap in the source and in debug builds but are not allocated in optimized builds. This conversion can still be performed but with additional restrictions on when. Taking a snapshot when an object has been converted violates the third safety requirement for using a snapshot from an optimized execution in a debug build. Thus, we restrict the compiler to only convert objects that are non-escaping *and* whose lifetimes do not cross snapshot safe points. For the remaining optimizations, we require the compiler to emit recovery maps to extract variables from the appropriate registers and stack locations in the optimized build.

Figure 3 shows a small code snippet at the source level at the top left, then, at the top right, the IR produced after optimizing and performing register allocation on that code, and, at the bottom, the recovery map created for the method call, which may trigger a snapshot and is marked with †. In this example, the compiler has applied constant propagation and folding, eliminating the assignments to the variables c and z and replacing all their uses with constant values. The recovery map therefore maps c to the constant 4 and z to 8. The variable o was allocated to register r0 and the value of the field load, which is stored in x, is stored in register r1. Thus, when recovering the values of these two variables the map indicates that they should be read out of r0 and r1

```
L1:
  int c = 4;
  int x = o.f;
  int y = o.f + c;
  int z = c * 2;
† foo(y + z);
  goto L2;
```

Source Code

```
L1:
  r1 = ld r0 f;
  r2 = addi r1 4;
  r3 = addi r2 8;
† call foo r3;
  jump L2;
```

Optimized IR

$\texttt{RecoveryMap}_{\dagger}$ = {o→r0, c→4, x→r1, y→r2, z→8}

Figure 3. Top Left: a small code snippet at the source level; Top Right: the IR produced after optimizing and performing register allocation; Bottom: the recovery map created for the method call marked with †.

respectively. Similarly, the value of y is placed in the register r2 and the recovery map is set similarly. Here the compiler has converted the repeated field read when computing y into a local variable read. If the call to foo triggers a snapshot we can reconstruct the values of all the source level variables that are needed from the information in the recovery map, $\texttt{RecoveryMap}_{\dagger}$, and the state of the optimized program.

Snapshot safe points are less restrictive than the *interrupt points* used in *dynamic deoptimization* [24] because the snapshot safe points are more widely spaced. Specifically, snapshot safe points are loop headers, method body entries, allocations, and call sites, while the interrupt points required for dynamic deoptimization include any point where a breakpoint may be set or a runtime error may occur. As a result, TTD-based reconstruction and snapshot safe points offer a much larger scope for the application of optimizations and impose less book-keeping on the compiler as compared to dynamic deoptimization and interrupt points. In particular, state reconstruction in TDD-based debugging safely permits almost all of the standard (and most generally beneficial) optimizations used in a modern JIT, as as Table 1 shows. Thus, TARDIS can simultaneously provide both the high-performance execution of an optimized build and the full-fidelity source-level value inspection of a debug build.

5. Evaluation

We implemented TARDIS, an optimized TTD system, as described in Section 3, for a C# runtime. We use the *Common Compiler Infrastructure* (CCI) framework [14] to add hooks for the debugger, insert code to monitor call frames and local variables, replace the allocation/GC with our *GenMS* allocator [28], and rewrite the standard C# libraries to log environmental interactions. After rewriting, the bytecode runs on the 32 bit JIT/CLR in Windows 8, on an Intel Sandy Bridge class Xeon processor at 3.6 GHz with 16 GB of RAM and a 7200 RPM SATA 3.0 magnetic disk drive. As TARDIS currently executes on a single core and uses a helper thread for snapshot compression/write, we use only 2 CPU cores.

To evaluate the overhead of the system, we selected a suite of programs designed to evaluate both the individual subcomponents and to understand how the TTD performs under varied workloads.

I/O Focused To evaluate TARDIS's file I/O and network subsystem behavior, we implemented two I/O specific benchmarks: FileIO and Http. The FileIO program reads all the files from a directory, computes a histogram of the words in each file, and writes the results to a fresh file. The Http program downloads a set of pages from Wikipedia and computes a histogram of the words and HTML tags on the page.

Compute/Allocation Focused To evaluate the system behavior on compute and allocation heavy tasks, we selected C# ports of well-known benchmark programs from the Olden suite: BH and Health. We also selected C# ports of standard application benchmarks, DB and Raytracer from Spec JVM and C# ports of the benchmarks Compress and LibQuantum from Spec CPU.

To evaluate the overhead of managed runtime resident TTD systems, and their subcomponents, we report their overheads relative to the baseline execution time in which the program is run through the rewriter but no TTD code, instrumentation, or debugger hooks are added. All values were obtained from the average of 10 runs, discarding the min/-max times, of the benchmark. In all of our tests, the standard deviation from the calculated mean was less than 8%. To cross-validate these overhead measurements, we used the high-resolution `StopWatch` class in .Net which provides sub-300 nanosecond resolution via hardware performance counters. The overhead values according to the stopwatch measurements were within 10% of the values calculated by comparing total runtimes. Thus, we have high confidence that our measurements reflect the true overheads for the TTD execution system.

5.1 Nonintrusive Runtime Overhead

We first evaluate the overhead and data write rates when running each benchmark under the TTD using the *nonintrusive* implementation from Section 3.1 with snapshots taken every 0.5 seconds. As the nonintrusive baseline approach is almost entirely independent of the specifics of the underlying runtime, these results are representative of what can be achieved with (1) almost any managed runtime regardless of the GC algorithm or runtime implementation used, and (2) with minimal implementation effort.

The results in the *Overhead* column in Table 2 show that the system tracks all of the information needed for time-travel debugging with a low overhead. The specialized tracking of the files allows the TTD system to eliminate almost all of the logging overhead for the disk access dominated FileIO benchmark. The Http benchmark requires the TTD to log all data received over the network. However, due to the relative speed of the hard-drive *vs.* the network, even in the network focused Http program, the overhead and data-write rates are low. Thus, the cost of extracting the snapshots dominates the overhead since nonintrusive implementation must execute the live heap walk-copy loop, Algorithm 3, for every snapshot taken. Even incurring this work per snapshot, the performance overhead is low — 22% or less — across all of our benchmarks.

Program	Overhead%	Raw MB/s	GZip MB/s
BH	18	1.6	0.9
Compress	6	2.6	1.2
DB	22	13.7	5.6
FileIO	5	2.2	0.8
Health	15	14.9	4.1
Http	11	2.7	1.1
LibQuantum	14	0.3	0.2
Raytrace	15	8.1	2.7

Table 2. Nonintrusive Baseline TTD performance with snapshots taken every 0.5 seconds. The *Overhead* column shows the runtime overhead percentage; the raw data generation rate (per second) is shown in *Raw* and the rate when applying gzip compression before writing is shown in *GZip*.

The *Raw* data column in Table 2 shows the rate at which the TTD system writes data. As described in Section 3.2, the snapshots can be compressed before storing. The *GZip* column shows the data rate when compressing snapshots before writing them. As can be seen, this substantially decreases the rate at which data is written (a factor of 2.8× on average). The nonintrusive TTD implementation copies out all live memory for each snapshot. Thus, the programs with larger live heaps can generate large, but still manageable, amounts of data, *e.g.* DB at 5.6 MB/s.

5.2 Optimized Runtime Overhead

We next look at the overhead and data write rates when running each benchmark under TARDIS, the optimized TTD from Section 3.2 taking snapshots every 0.5 seconds. To reduce the runtime overhead and data write rates of the baseline implementation, TARDIS (1) eliminates executions of the walk-copy loop through opportunistic piggybacking on the GC, (2) for non-piggybacked snapshots, performs the walk-copy loop only on the nursery, and, (3) for the old-space, copies only objects modified since the last snapshot.

The results in the *Overhead* and *Compressed* columns in Table 3 show that the optimizations dramatically decrease the cost of supporting TTD functionality. With the optimizations, the largest runtime overhead drops to 11% and the max data write rate improves to 1.3 MB/s. The impact of the piggybacking optimization on the runtime overhead is dependent on the nature of the workload. LibQ allocates at a very low rate so there are very few opportunities to take a low-

Program	Overhead%	Raw MB/s	GZip MB/s
BH	5	0.6	0.4
Compress	4	1.4	0.4
DB	11	1.4	0.6
FileIO	4	0.4	0.1
Health	6	3.9	1.3
Http	9	1.2	0.4
LibQuantum	9	0.2	0.1
Raytrace	8	3.0	1.0

Table 3. TARDIS TTD performance with snapshots taken every 0.5 seconds. The *Overhead* column shows the runtime overhead percentage; the raw data generation rate (per second) is shown in the *Raw* data column and the rate when applying gzip compression before writing is shown in *GZip*.

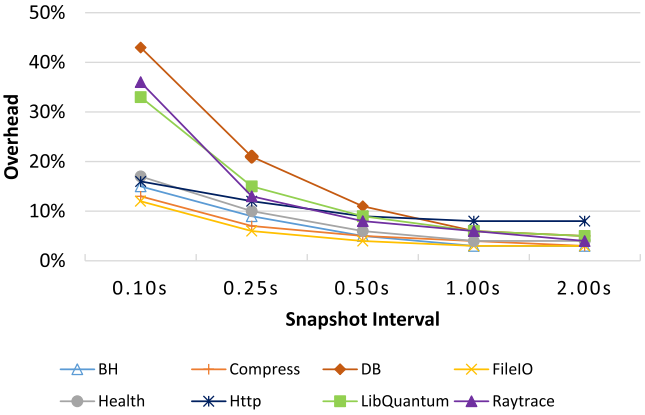
cost (*i.e.* constant additional cost) snapshot while BH (and Health) allocate rapidly and constantly providing many opportunities to piggyback snapshot extraction on the GC. Despite triggering the GC early on some occasions to combine the GC/snapshot activity, we observed no major disruption to the behavior of the GC and had at most 1 additional GC for any benchmark run. Just as in a generational collector, TARDIS's generational optimization greatly reduces the time spent traversing the heap and the amount of data that must be copied. This is a significant factor in the improvements in DB, which has a large number of strings that are live (but unmodified) during the program execution.

Given the low runtime overheads and data write rates in Table 3, TARDIS can easily be used as the default debugger for day-to-day development. Further, with an average slowdown of 7% and data rate of 0.6 MB/s, it is even feasible to deploy time-travel enabled builds in the wild to assist in gathering execution and crash data.

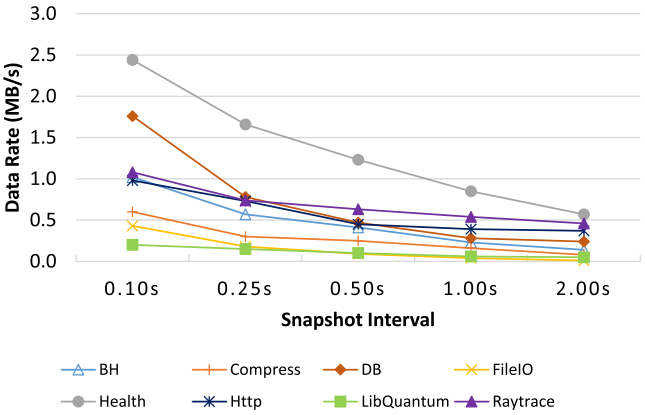
5.3 Sampling Interval Selection

The previous evaluations use a fixed snapshot interval of 0.5 seconds. We chose this interval to minimize the overhead of the TTD system while still ensuring low latency when initiating time-travel. Increasing the snapshot interval, at the cost of increasing time-travel latency, reduces the overhead and data write rates. Thus, we want to identify both a good default value for the snapshot interval and a reasonable range of snapshot rates to support the dynamic interval adjustment from Section 3.2.

Figure 4(a) shows the runtime overhead TARDIS, our optimized TTD implementation, imposes for a range of snapshot rates ranging from one snapshot every 0.1 seconds to one every 2 seconds. As expected, the overhead is large for frequent snapshots and asymptotically decreases as the interval between snapshots lengthened. The time-travel latency is directly proportional to the snapshot interval (Table 4),



(a) Percentage increase in execution time.



(b) Data written per second.

Figure 4. Cost of TARDIS with various snapshot intervals.

while the runtime overhead drops off rapidly between 0.1s-0.25s and then more slowly for successive increases in the sampling interval. Thus, we select a sampling rate of one snapshot every 0.5 seconds as the sweet spot on the curve where the runtime overhead drops to 11% for any of the benchmarks but the latency remains well under one second (Table 4). Interval values between 0.25 and 2 seconds are reasonable choices, depending on the particulars of the program being executed, and provide the dynamic adjustment a suitable operating range.

Figure 4(b) shows the rate at which TARDIS writes snapshot and logging data. The overall behavior is similar to the runtime overhead, generally larger for small snapshot intervals, decreasing to below 1.5 MB/s at an internal of $0.5s$, and asymptotically approaching a baseline rate of around 0.5 MB/s at an interval of $2s$. In most benchmarks, increasing the snapshot interval leads to a rapid reduction in the amount of data written due to the increase in time for an allocated objects to become garbage (*i.e.* the generational hypothesis) and thus never be included in a snapshot. However, the Health benchmark continually allocates new objects, keeps them live, and modifies them randomly. Thus, it

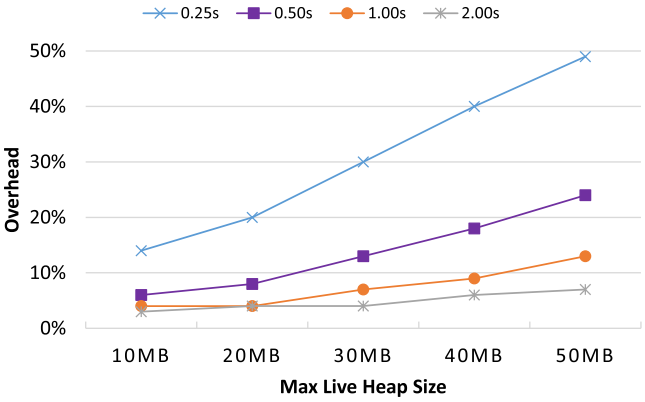


Figure 5. Percentage increase in runtime with TARDIS for various heap sizes of the Health program and four possible snapshot intervals.

does not benefit as much as the other programs but increasing the interval still effectively amortizes the cost of tracking the object modifications over longer periods. Based on these results, we identify a default snapshot rate of $0.5s$ as providing a low overall rate of data writes (under 1.3 MB/s) and all the data rates between 0.25 and 2 seconds as suitable for the dynamic adjustment to use.

5.4 Heap Size and Overhead

The results in Table 2 and Table 3, suggest that the snapshot overhead follows the overhead trend of the related GC algorithm [5]. The baseline results in Table 2 mirror those of a basic full heap collector, such as a semispace or compacting collector, and the optimized results in Table 3 and Figure 4(a) are similar to those of a generational collector. To better understand the relationship between heap size and TTD overhead, we took the Health program and varied its command line parameters to measure the overhead of the TARDIS TTD system for a range of live heap sizes and snapshot intervals.

Figure 5 shows, for four snapshot intervals, the overhead of the TTD system on max live heap sizes ranging from 10 MB to 50 MB. As expected, the overhead increases as the live heap size increases and the curves for the smaller snapshot intervals lie above the curves for the larger intervals. However, even when the max live heap size is 50 MB the overhead remains just under our target overhead of 25% for the default 0.5 second snapshot interval and is well under 10% when the snapshot interval is set to 2 seconds. The figure also shows that the increase in overhead, slope of the curve, is also smaller for the larger intervals. Again, the larger interval provides more time for allocated objects to become garbage and amortizes the cost of tracking multiple writes to the same object in the old space. As a result, the rate of increase in overhead for a snapshot interval of 2 seconds, approximately 2% per 10 MB, is much smaller than the rate of increase for intervals of both 0.5 and 0.25 seconds.

Program	Avg. Latency (s)	Max Latency (s)
BH	0.37	0.68
Compress	0.23	0.45
DB	0.41	0.62
FileIO	0.27	0.41
Health	0.29	0.48
Http	0.23	0.52
LibQ	0.21	0.45
Raytrace	0.23	0.51

Table 4. Time-travel latency.

For all intervals between 0.5 seconds and 2 seconds, the average data write rate is less than 19 MB/s uncompressed and 6 MB/s compressed. Using a generational design and write barriers in its snapshot algorithm allow TARDIS to pay only for old, modified objects, not every live object. The resulting snapshot algorithm scales well even on programs that have large live heaps. Leveraging other recent advances in GC design [8, 41] may yield similar performance improvements.

5.5 Time-Travel Latency

The final aspect of TARDIS we evaluate is the latency a user encounters when initiating a time-travel operation — *reverse-step*, *reverse-step into*, or *reverse-run* — before control returns to the debugger. The results in Table 4 show the average/max latency when initiating time-travel when performing reverse execution operations at 10 randomly selected points.

The results in Table 4 show that the time required for a reverse execution is, as expected, close to the average or max time a point in the program execution can be from the most recent snapshot. At the selected snapshot interval of 0.5 seconds these values are 0.25 seconds and 0.5 seconds respectively. Due to delays between the expiration of the snapshot timer and the next safe snapshot point, along with the cost of snapshot restoration and trace replaying, these values are, in practice, slightly larger with an average of $0.28s$ seconds and a maximum of $0.68s$ seconds. Although this latency is large enough for a user to notice, exceeding $0.1s$, time-travel is far faster than restarting the program from the beginning. Further, as we compute a full trace for the last *TRACE_RNG* instructions in the replay, the latencies for additional operations is near zero.

6. Discussion

Without OS support, TDD systems cannot monitor file system events and cannot time-travel debug multi-process ensembles that communicate via the file system. Similarly, if the debugged processes communicate via other IPC mech-

anisms for which the OS does not provide notifications or that the broker does not co-ordinate, then we cannot capture and replay their interactions. As future work, we plan to investigate building TTD functionality into the core of a managed OS, such as Singularity [25], to enable the same type of lightweight TTD augmented with awareness of OS events and, therefore, the ability to synchronize over all communication channels.

Our current implementation does support multi-threaded programs but only executes them using a single core, uniprocessor execution model. This decision works well for programs that employ threads primarily to improve UI responsiveness, but slows down programs that use multiple threads for task parallelization and precludes time-travel debugging some types of data race bugs. An alternative is to utilize techniques from *Octet* [9], a JVM that provides low overhead, deterministic replay for concurrent programs.

In this work, we do not snapshot or restore the data referenced by native pointers. Thus, these values are out-of-sync during replay in the TTD. As a result, we mark them as not inspectable in the debugger. This limits the ability of the developer to fully understand the affects of statements involving native pointers. However, the use of native invocation in managed languages is discouraged and best practice suggests limiting its use to small parts of the program. At the cost of, potentially much, higher overhead, these native structures and invocations could be reverted by using full tracing, such as iDNA [4, 32], to capture and reverse the sequences of events that occur in a native call.

The central goal of this paper is to establish the feasibility of our general approach to TDD, for the common case of debugging programs with small-to-moderate input data and live heap sizes. This result is a first step: we believe that TARDIS can scale to large heaps, 100+ MB to multi-GB, just as GC systems have done. In addition many of the snapshotting optimizations presented in this work rely on features of a generational collector to achieve a low overhead which may rule out the use of another kind of collector that could give better performance for a particular application. Thus, further study to better understand how the object demographics, *e.g.* allocation rates, live heap size, and survival rates, of a program interact with the engineering choices made in the GC/snapshot system, *e.g.* collector behavior, write-barrier algorithm, COW support, is needed (See [5–7]). Finally, we observe that leveraging ideas from the recent GC literature, such as parallel/concurrent operation [41], organizing the heap to reduce the cost of managing old, large, or immutable objects [8], and possibly using dedicated hardware to assist in heap management [13], may be key to ensuring low overheads for a TTD system when dealing with very large live heap sizes.

7. Related Work

Time-traveling debuggers are an attractive means for improving debugging. Work on them has a long history and a wide variety of methods for implementing them have been proposed. In addition, many of the challenges encountered in their construction must also be overcome when building systems for other applications such as deterministic-replay, automatic fault-recovery, transparent system/application migration, or program invariant checking.

Tracing TTD The most straight-forward mechanism for implementing a TTD system is simple trace collection with the ability to revert the program state based on the logged data [4, 15, 21, 23, 35, 45]. Fundamentally, a trace-based approach eagerly records all the data needed to revert any event in the program (regardless of if it will be needed later). Even with substantial effort to reduce the time/space cost of this logging [4, 15, 45], using a purely trace-based approach to TTD results in average runtime overheads of over $10\times$ and potential logging data rates of 100's of MB per second. These high overhead rates make them infeasible for use as the default debugger.

Record-Replay TTD During debugging only a small portion of the entire execution is likely to be of interest. Thus, an alternative to eagerly tracing the entire program execution is to intermittently take snapshots of the entire program state and then, on demand, restore, and replay [10, 20, 30, 51, 53–55]. Early work in this area focused on the use of memory protection features to support incremental and asynchronous checkpoints using *copy-on-write* [10, 20, 42]. A similar approach using hardware virtualization via a hypervisor has been explored [30]. However, the need to take snapshots at the coarse grained address space (or full system) level requires large intervals between snapshots to achieve low overheads "Overhead is low [16-33$\%$] even for very short checkpoint intervals of 10 seconds" [30]. Thus, these hypervisor based approaches frequently suffer from multi-second latency when initiating reverse execution, latencies that trigger users to look for other solutions.

A major source of the snapshot overhead is the *semantic gap* between memory writes at a hardware level and the live data at the level of the executing program. This problem is particularly severe for GC based languages which generally write to large ranges of memory, only a small portion of which is live, on a regular basis. As a result, the snapshot process spends large amounts of time serializing irrelevant data. This observation was made by Wilson and Moher [54] who proposed coordinating the snapshot extraction with the GC in a LISP system to record only live objects, instead of the full process address space. However, the system was not implemented and only *performance projections* of 10% with a shapshot interval of 2.5 seconds were made as ballpark overhead estimates. In addition, their work focused solely on memory state and did not explicitly identify, generalize to

other aspects of the computation, or exploit the more general idea that a managed runtime has access to the full range of information needed to construct an optimized TTD system.

Persistent Data-Structure TTD Persistent data structures efficiently version a data structure when it is modified such that the current or any previous version of the structure can be efficiently accessed using a timestamp key [19]. The idea of lifting the program state into a persistent data structure has been used to produce TTD systems [36, 43]. Reverse execution in these systems is simply reading the values from the appropriate version of the persistent structure. Although efficient specialized implementations of persistent trees [19] exist, transforming arbitrary (heap) graphs into persistent structures has a larger overhead. Consequently, these approaches incur runtime overheads on the order of $3\times$ and, as the persistent structure grows $O(1)$ in size per write, the amount of memory devoted to the execution history can quickly exceed the memory devoted to the program.

Commerical TTD A testament to the appeal of TTD is the number of industrial offerings in the last decade [15, 26, 47, 51]. Of these products, UndoDB and Chronon are published under EULAs that prohibit the publication of evaluation results, so we can only use their marketing material or other public statements to compare them to TARDIS. According to their product webpage, UndoDB, a record-replay TTD for C/C++ on Linux, the "Average slowdown is $2-4\times$, though in pathological cases slowdown can be up to 10x" [52]. Chronon is a trace-based TTD for Java [15] whose webpage states "The Chronon Recorder puts minimal overhead on your applications" [16] and whose CTO Prashant Deva said "We dont [*sic*] give out performance numbers" [17]. VMware Workstation 7 included whole system replay debugging [47], which was discontinued in 2011 when Workstation 8 was released. Anecdotally, VMware's replay debugging imposed a 2–$3\times$ slowdown on forward execution. Microsoft offers Intellitrace [26], which logs only function argument values, providing limited time-travel functionality.

Related Areas of Research A number of research areas use technologies that are relevant to the implementation of a time-traveling debugger. The ability to *deterministically replay* the execution of a program is critical to implementing a TTD. Of particular interest is recent work on deterministic replay systems in managed runtimes such as JavaScript [11, 38]. The Mozilla *rr* [48] tool has explored leveraging hardware performance counters to reduce the cost of record/replay with good initial results. Also relevant is recent work on *Octet*, a JVM which provides a mechanism for performing low overhead deterministic replay with concurrent programs [9]. Checkpoint (snapshot) extraction is another topic that has been studied in a number of contexts including *fault-recovery* [50] and *process migration* [18, 37, 44]. The idea of piggybacking on the GC to perform a task (at low cost) has also been used for efficient heap invariant checking [46].

Execution History Analysis A number of research areas depend on, or would benefit from the availability of, a low cost method for obtaining the execution history for a program. Work on interrogative debugging [1, 31, 36] utilizes the execution history to construct a *program slice* [2] of the statements that explain why and how a particular value was computed. Automated error detection and root cause analysis tools can use execution history information to automatically identify and localize a variety of software defects [12, 22, 27, 29, 34]. The cost of producing the history, at the needed level of detail, is a limiting factor in the applicability of all these techniques. Thus, the ability to efficiently replay snippets of a program's execution, and track exactly the required information, is an enabler for further work in these areas.

8. Conclusion

Managed runtimes, such as the .Net CLR or a JavaScript engine, have already paid the price to support the core features — type safety, memory management, and virtual IO — that can be reused to implement a low cost time-traveling debugger. Using this insight, this paper has introduced TARDIS, an affordable time-traveling debugger for managed languages. Our experimental results show that TARDIS provides time-travel functionality while incurring an average overhead of only 7% during normal execution, a rate of 0.6 MB/s of history logging, and is able to begin time-travel with a worst-case $0.68s$ start latency on our benchmark applications.

These low runtime overheads and data write rates, combined with its ability to debug optimized code, make it feasible to envision the use of a time-traveling debugger as the default debugger during day-to-day development. TARDIS's lightweight collection of detailed execution history information represents an enabling technology for related research areas including interrogative debugging, automated fault localization, automatic fault-recovery and reporting. Thus, TARDIS represents a key step on the path to bringing time-traveling debuggers into the mainstream and transforming the practice of debugging.

Acknowledgments

We would like to thank the OOPSLA reviewers, Ben Livshits, Kathryn McKinley, James Mickens, Todd Mytkowicz, and Ben Zorn for their feedback, comments, and support for this work.

References

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[3] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and

M. Mergen. Implementing Jalapeño in Java. In *OOPSLA*, 1999.

[4] 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*, 2006.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS*, 2004.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis (2006-mr2). In *OOPSLA*, 2006.

[7] S. M. Blackburn and A. L. Hosking. Barriers: Friend or foe? In *ISMM*, 2004.

[8] S. M. Blackburn and K. S. McKinley. Immix:A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, 2008.

[9] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. F. Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *OOPSLA*, 2013.

[10] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI*, 2000.

[11] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *UIST*, 2013.

[12] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[13] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA*, 2012.

[14] Common compiler infrastructure. http://ccimetadata.codeplex.com.

[15] Chronon v3.10. http://chrononsystems.com.

[16] Chronon performance, Visited June 4, 2014. http://www.chrononsystems.com/what-is-chronon/performance.

[17] Announcing Chronon "DVR for Java", Visited June 4, 2014. http://www.theserverside.com/discussions/thread.tss?thread_id=62697.

[18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[19] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 1989.

[20] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. In *Parallel and Distributed Debugging*, 1988.

[21] GDB v7. http://www.gnu.org/software/gdb/news/reversible.html.

[22] Z. Gu, E. T. Barr, D. Schleck, and Z. Su. Reusing debugging knowledge via trace-based bug search. In *OOPSLA*, 2012.

[23] C. Head, G. Lefebvre, M. Spear, N. Taylor, and A. Warfield. Debugging through time with the Tralfamadore debugger. In *RESoLVE*, 2012.

[24] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI*, 1992.

[25] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS*, 2007.

[26] Intellitrace. http://msdn.microsoft.com/en-us/library/vstudio/dd264915.aspx.

[27] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*, 2007.

[28] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman & Hall/CRC, 2012.

[29] Y. P. Khoo, J. S. Foster, and M. Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *ICSE*, 2013.

[30] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[31] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *ICSE*, 2008.

[32] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: Portable mixed-environment debugging. In *OOPSLA*, 2009.

[33] H. Lee, D. von Dincklage, A. Diwan, and J. E. B. Moss. Understanding the behavior of compiler optimizations. *Software Practice and Experience*, 2006.

[34] D. Lessa, B. Jayaraman, and J. Chomicki. Temporal data model for program debugging. In *DBPL*, 2011.

[35] B. Lewis. Debugging backwards in time. In *AADEBUG*, 2003.

[36] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *ECOOP*, 2008.

[37] J. T. K. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime migration of browser sessions for JavaScript web applications. In *WWW*, 2013.

[38] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for Javascript applications. In *NSDI*, 2010.

[39] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[40] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, 1993.

[41] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *PLDI*, 2010.

[42] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *TCON*, 1995.

[43] F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: Efficient in-memory object graph versioning. In *OOPSLA*, 2009.

[44] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *ASPLOS*, 2011.

[45] G. Pothier, É. Tanter, and J. M. Piquer. Scalable omniscient debugging. In *OOPSLA*, 2007.

[46] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *OOPSLA*, 2010.

[47] Retrace. http://www.replaydebugging.com/.

[48] Mozilla rr tool. http://rr-project.org/.

[49] J. B. Sartor, M. Hirzel, and K. S. McKinley. No bit left behind: The limits of heap data compression. In *ISMM*, 2008.

[50] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS*, 2008.

[51] UndoDB v3.5. http://undo-software.com.

[52] UndoDB performance, Visited June 4, 2014. http://undo-software.com/content/faqs.

[53] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. URDB: A universal reversible debugger based on decomposing debugging histories. In *PLOS*, 2011.

[54] P. R. Wilson and T. G. Moher. Demonic memories for process histories. In *PLDI*, 1989.

[55] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MoBS*, 2007.