The RoadRunner Dynamic Analysis Framework for Concurrent Programs

Cormac Flanagan
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Abstract

RoadRunner is a dynamic analysis framework designed to facilitate rapid prototyping and experimentation with dynamic analyses for concurrent Java programs. It provides a clean API for communicating an event stream to back-end analyses, where each event describes some operation of interest performed by the target program, such as accessing memory, synchronizing on a lock, forking a new thread, and so on. This API enables the developer to focus on the essential algorithmic issues of the dynamic analysis, rather than on orthogonal infrastructure complexities.

Each back-end analysis tool is expressed as a filter over the event stream, allowing easy composition of analyses into *tool chains*. This tool-chain architecture permits complex analyses to be described and implemented as a sequence of more simple, modular steps, and it facilitates experimentation with different tool compositions. Moreover, the ability to insert various monitoring tools into the tool chain facilitates debugging and performance tuning.

Despite RoadRunner's flexibility, careful implementation and optimization choices enable RoadRunner-based analyses to offer comparable performance to traditional, monolithic analysis prototypes, while being up to an order of magnitude smaller in code size. We have used RoadRunner to develop several dozen tools and have successfully applied them to programs as large as the Eclipse programming environment.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging–*monitors, testing tools*; D.2.4 [*Software Engineering*]: Software/Program Verification–*reliability*

General Terms Languages, Algorithms, Reliability

Keywords concurrency, dynamic analysis

1. Introduction

Concurrent programs are notoriously prone to defects caused by interference between threads. These are difficult errors to detect via traditional testing alone because scheduling nondeterminism leads to exponentially-many possible thread interleavings, each of which may induce an error. The recent advent of multi-core processors only exacerbates this problem by exposing greater concurrency.

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.

The difficulty of validating multithreaded software via traditional testing has motivated much research on dynamic analysis tools for detecting a variety of errors, including race conditions [5, 6, 8, 10, 21, 24, 27, 31], deadlocks [1, 16], and violations of desired atomicity [9, 13, 28, 29] and determinism properties [4, 23]. The overhead of developing such tools, however, is rather large. For example, the Atomizer dynamic atomicity checker [9] contained over 8,500 lines of code, which was split between extensions to an existing Java front end and a run-time library.

To facilitate research on dynamic analysis tools for concurrent programs, we have developed a robust and flexible framework called RoadRunner that substantially reduces the overhead of implementing dynamic analyses. RoadRunner manages the messy, low-level details of dynamic analysis and provides a clean API for communicating an event stream to back-end analysis tools. Each event describes some operation of interest performed by the target program, such as accessing memory, acquiring a lock, forking a new thread, etc. This separation of concerns allows the developer to focus on the essential algorithmic issues of a particular analysis, rather than on orthogonal infrastructure complexities.

RoadRunner is written entirely in Java, with no modifications to the underlying Java Virtual Machine. It adds instrumentation code to the target program's bytecode at load time, avoiding any need to re-compile or otherwise modify the target. The system is roughly 20,000 lines of code in size, and is available from `www.cs.williams.edu/~freund/rr`. In the remainder of this section, we outline the RoadRunner architecture and its benefits.

RoadRunner facilitates writing dynamic analyses. Writing a RoadRunner back-end analysis tool only requires defining methods to handle various events of interest. Each event handler takes as an argument an event object describing the operation being performed by the target program. The RoadRunner framework provides support for associating analysis (or *shadow*) state with memory locations, locks, and threads; for reporting error messages; for identifying the source location for a particular event; and so on.

In our experience, RoadRunner-based analyses are substantially simpler to implement than traditional, monolithic analyses. As one example, re-implementing the Atomizer [9] on top of RoadRunner required only 245 lines of new code, versus the 8,500 lines of code for the original implementation.

To date, we have used RoadRunner to develop several dozen dynamic analysis tools, many of which are summarized in Figure 1. Their number and modest size (for even the most complex) provides evidence of RoadRunner's flexibility and utility. In addition, due to careful implementation and optimization choices, the performance of RoadRunner-based analyses is competitive with other specialized research prototypes built to study individual analyses in this domain (e.g., [9, 21, 24]).

Tool Name	Size (lines)	Description
Utility Tools		
EMPTY	78	Processes all events as no-ops
COUNT	105	Counts the number of each type of event
TIMER	120	Profiles the time taken by each event handler
PRINT	177	Prints the event stream in a human-readable form
CONTENTION	62	Identifies where threads block due to lock contention
FAULTINJECTION	109	Randomly filters synchronization operations out of event stream to induce errors
Analysis Tools		
THREADLOCAL	48	Filters accesses to thread-local data
READONLY	56	Filters accesses to read-only data
PROTECTINGLOCK	69	Filters lock operations for locks protected by other locks
LOCKSET [24]	327	Detects races using the LockSet algorithm
ERASERWITHBARRIER [22, 24]	457	Detects races using LockSet + a barrier analysis
HAPPENSBEFORE [19]	486	Detects races using VectorClocks
DJIT+ [22]	582	Detects races using an optimized VectorClock algorithm
MULTIRACE [22]	923	Detects races using a hybrid LockSet/VectorClock analysis
GOLDILOCKS [8]	1,416	Detects races using an extended LockSet algorithm
FASTTRACK [10]	758	Detects races using an Epoch/VectorClock analysis
VELODROME [12]	1,088	Detects serializability errors
ATOMIZER [9]	245	Detects atomicity violations using Lipton's theory of reduction
SINGLETRACK [23]	1,655	Detects determinism errors
JUMBLE [11]	1,326	Adversarial memory implementation to utilize relaxed memory model nondeterminism
SIDETRACK [30]	500	Detects generalized serializability errors

Figure 1. Representative tools implemented for ROADRUNNER. Size is measured in lines of code, excluding blank lines and comments.

ROADRUNNER facilitates composing dynamic analyses. ROADRUNNER tools can be composed (via command-line configuration) into a *tool chain*, where events are dynamically dispatched down this tool chain until they are handled by the appropriate tool. Thus, each dynamic analysis is essentially a filter over event streams, and ROADRUNNER makes it easy to create new analyses via the composition of simpler analyses.

As an illustration, the Eraser algorithm [24] can be expressed in terms of three simpler analyses: a `ThreadLocal` analysis, which filters out accesses to thread-local data; a `ReadOnly` analysis, which filters out accesses to read-only data; and a `LockSet` analysis, which tracks the set of locks consistently used to protect each variable and reports a potential race if one of these sets becomes empty. The following command line configuration composes these three tools and applies them to the target program `Target.class`:

```
rrrun -tool=ThreadLocal:ReadOnly:LockSet Target
```

Each of these tools can be independently re-used. For example, we can significantly improve the performance of a relatively slow `HappensBefore` race detector [19] by composing it with the above `ThreadLocal` and `ReadOnly` filters, so that `HappensBefore` analyzes only shared, mutable memory locations:

```
rrrun -tool=ThreadLocal:ReadOnly:HappensBefore Target
```

As another example, the Atomizer serializability checker must reason about locations with potential race conditions, but may ignore race-free locations. Tool composition enables us to write the `Atomizer` tool without worrying about race conditions. We simply prefix `Atomizer` in the tool chain with any race detector, which allows us to easily experiment with analysis configurations, as in:

```
rrrun -tool=ThreadLocal:ReadOnly:LockSet:Atomizer Target
rrrun -tool=FastTrack:Atomizer Target
```

ROADRUNNER facilitates debugging dynamic analyses. ROADRUNNER's pipe-and-filter architecture enables one to insert additional diagnostic tools into the tool chain to facilitate debugging. The following utility tools have proven particularly useful:

1. PRINT: print out the event stream at various places in the tool chain (for correctness debugging);
2. TIME: time the overhead of various tools in the chain (for performance debugging);
3. COUNT: count the distribution of events at various points in the chain (to identify which event handlers to optimize); and
4. RECORD: record an event stream for later replay, to circumvent scheduling non-determinism during debugging.

Note that these tools, once written, can be used in combination with any others, thereby avoiding the cross-cutting clutter of inserting code to perform these operations in the analysis event handlers.

ROADRUNNER facilitates comparing dynamic analyses. Performing controlled comparisons between different dynamic analyses is quite difficult. Analyses written by different authors are often implemented and evaluated for different languages, computing platforms, or virtual machines. Some also change the run time's virtual machine, whereas others run on an unmodified version.

ROADRUNNER addresses this problem by making it easier to reimplement a collection of published algorithms in a uniform framework, thereby obtaining clear precision and performance comparisons. For example, ROADRUNNER has been used to implement and experimentally compare six different dynamic race detection algorithms [10], resulting in a deeper understanding of their relative strengths. Such controlled experiments are an essential part of the scientific process.

2. ROADRUNNER API

This section outlines the ROADRUNNER API and describes how to write tools that monitor the stream of events generated by the target program while maintaining instrumentation state (that is, tool-specific shadow information about each thread, lock, and memory location used by the target). We first focus on implementing a single tool using this API and will then extend our presentation to include tool composition in Section 2.3 below.

Figure 2 summarizes the core classes in the ROADRUNNER API. Each analysis tool is defined by extending the `Tool` class. The `Tool` class defines methods to handle each type of event that may be generated by the target program, including:

- non-volatile memory (field and array) accesses;
- volatile memory (field and array) accesses;
- lock acquire and release;
- thread creation and `start`, `join`, and `interrupt` operations;
- `wait` and `join` operations; and
- method entry and exit.

We show four representative event handlers (lines 4–7) for thread creation, lock acquires and releases, and accesses to non-volatile memory. Each event handler method takes a specific subclass of `Event`. These event classes are also described in Figure 2. Before discussing events and event handlers in more detail, we first introduce several auxiliary ROADRUNNER data structures.

Thread and Lock Shadows. The Java Virtual Machine allocates a `Thread` object for each thread in the target program. ROADRUNNER associates a `ShadowThread` object with each `Thread` object. Each `ShadowThread` object (see Figure 2, line 36) contains a reference to the underlying `Thread`, a unique integer identifier `tid`, and a reference to the `ShadowThread` for the parent (or forking) thread, if any. Similarly, ROADRUNNER allocates a corresponding `ShadowLock` object (line 45) for every Java object used as a lock.

Tools often need to associate tool-specific information with each thread and lock. We could just add new fields to the `ShadowThread` and `ShadowLock` classes as necessary, but this clearly leads to poorly defined abstractions and pollution of our core APIs. Letting tools subclass these classes also leads to problems, since efficiently implementing tool composition becomes difficult.

To permit extensibility while avoiding these problems, we introduce the `Decoration` class (line 57). Essentially, the `Decoration` class provides a generic map from `Key`s to `Value`s. This map can be implemented as a hash table, but that would not provide adequate performance. Instead, we require the `Key` to be a subclass of `Decoratable`. The `Decoration` map can then store the `Value` for a `Key` in the `Key` itself. Specifically, each `Decoration` has a unique small integer, which is used to index into an `Object` array declared in the `Decoratable` superclass. This pattern provides a fast implementation of the `get` and `set` methods in `Decoration`.

The `ShadowThread` and `ShadowLock` classes extend the class `Decoratable` and provide a generic `makeDecoration` method whereby tools can add `Decoration`s for any desired type `T` to those structures. We illustrate the use of these `Decoration` fields in the `LockSet` example below.

Variable Shadows. Tools also need to record information about each memory location used by the target program (that is, each static field, instance field, and array element). While `Decoration`s work well for threads and locks, they are ill-suited for memory locations. There are orders of magnitude more memory locations than locks or threads, and these memory locations are accessed orders of magnitude more frequently.

Instead, ROADRUNNER maintains a *shadow location* corresponding to each memory location, and this shadow location stores a `ShadowVar` object reference. When a location is first accessed, the tool's `makeShadowVar` method is called to compute the shadow location's initial value. `ShadowVar` is an empty interface that simply serves to annotate types intended to be stored in these shadow locations. The class `ShadowThread` implements the `ShadowVar` interface because some tools (such as the `ThreadLocal` tool described below) store `ShadowThread`s in shadow locations.

Events. The `Event` superclass (line 13) stores a `ShadowThread` object for the thread performing the operation. Subclasses of `Event` then contain additional information describing that particular kind event. For example, `AcquireEvent` (line 19) contains the source location of the acquire (in the `info` field) and a `ShadowLock` object describing the object whose lock is being acquired.

Figure 2: Core ROADRUNNER API

```
1  Tool Abstract Class
2
3  abstract class Tool {
4    void create  (NewThreadEvent e) { }
5    void acquire (AcquireEvent e)   { }
6    void release (ReleaseEvent e)   { }
7    void access  (AccessEvent e)    { }
8    abstract ShadowVar makeShadowVar(AccessEvent e);
9  }
10
11 Events
12
13 class Event {
14   ShadowThread thread;
15 }
16
17 class NewThreadEvent extends Event { ... }
18
19 class AcquireEvent extends Event {
20   AcquireInfo info;     // loc
21   ShadowLock lock;
22 }
23
24 class AccessEvent extends Event {
25   ShadowVar shadow;
26   boolean putShadow(ShadowVar newShadow)
27 }
28
29 class FieldAccessEvent extends AccessEvent {
30   FieldAccessInfo info;  // loc, class/field desc.
31   Object target;         // reciever
32 }
33
34 Thread, Lock, and Variable Shadows
35
36 class ShadowThread extends Decoratable
37                    implements ShadowVar {
38   Thread thread;
39   int tid;
40   ShadowThread parent;
41
42   static <T> Decoration<ShadowThread,T> makeDec(T init)
43 }
44
45 class ShadowLock extends Decoratable {
46   Object lock;
47
48   static <T> Decoration<ShadowLock,T> makeDec(T init)
49 }
50
51 interface ShadowVar { }
52
53 Decorations
54
55 class Decoratable { ... }
56
57 class Decoration<Key extends Decoratable,Value> {
58   Decoration(Value initial)
59   Value get(Key k)
60   void set(Key k, Value v)
61 }
```

For each memory access, ROADRUNNER calls the tool's `access` method, passing an `AccessEvent` that contains the current value of the shadow location in the `shadow` field, as well as a method `putShadow` for updating that shadow location (lines 24–27). Additional subclasses of `AccessEvent`, such as `FieldAccessEvent` (line 29), record details about different types of accesses.

Figure 3: `LockSet` **Tool**

```
61 class Set implements ShadowVar {
62   static Set empty()
63   Set add(ShadowLock lock)
64   Set remove(ShadowLock lock)
65   Set intersect(Set other)
66   boolean isEmpty()
67 }
68
69 class LockSet extends Tool {
70
71   static Decoration<ShadowThread,Set> locksHeld =
72     ShadowThread.makeDec(Lockset.empty());
73
74   void acquire(AcquireEvent e) {
75     Set ls = locksHeld.get(e.thread);
76     locksHeld.set(e.thread, ls.add(e.lock));
77   }
78
79   void release(ReleaseEvent e) {
80     Set ls = locksHeld.get(e.thread);
81     locksHeld.set(e.thread, ls.remove(e.lock));
82   }
83
84   ShadowVar makeShadowVar(AccessEvent e) {
85     return locksHeld.get(e.thread);
86   }
87
88   void access(AccessEvent e) {
89     Set ls = (Set)e.shadow;
90     Set held = locksHeld.get(e.thread);
91     Set newLs = ls.intersect(held);
92     e.putShadow(newLs);
93     if (newLs.isEmpty()) error();
94   }
95 }
```

2.1 LockSet Tool Example

To illustrate the RoadRunner API in more detail, we show how to implement Eraser's LockSet analysis on top of RoadRunner. This analysis tracks the locks currently held by each thread and the set of locks consistently held on all accesses to each memory location.

The `Set` class in Figure 3 represents a set of `ShadowLock` objects. That class provides methods to create an empty `Set`, to add and remove specific `ShadowLocks` from a `Set`, to intersect two `Sets`, and to check for emptiness. Our implementation uses a functional representation of these sets.

The `LockSet` class creates a decoration `locksHeld` to store the locks held by each thread (line 71). This set is initially empty and is updated by the `acquire` and `release` event handlers. (Java locks are reentrant, but to simplify tool implementations, RoadRunner does not generate events for re-entrant lock acquires and releases, since they are no-ops.)

The `LockSet` class stores a `Set` in each shadow location. When a location is first accessed, the method `makeShadowVar` initializes the shadow location with the set of locks held by the accessing thread. At each memory access, the `access` event handler retrieves the current lock set `ls` from the shadow location and the set `held` of locks held by the current thread, and computes the intersection of these two sets. The result is stored back in the shadow location and, if that set is empty, `LockSet` reports an error.

2.2 RoadRunner Synchronization Models

All threads in the target program may generate events. Since the thread triggering an event is also the thread that executes the tool's event handler code, it is possible that multiple event handlers may be running concurrently. Thus, it is important to design tools to avoid concurrency errors in the event handlers, such as race conditions on a tool's data structures. Different mechanisms may be used to ensure thread-safety:

1. *Event Serialization*: RoadRunner can be configured to serialize events so that only a single handler is active at a time. While useful for preliminary design and exploration, as well as debugging, this approach incurs a major performance penalty since it essentially removes all concurrency from the target program.
2. *Tool-Internal Concurrency Control*: A programmer may add synchronization to the internal implementation of a tool to provide finer-grained concurrency control.
3. *Optimistic Concurrency Control for* `ShadowVar`*s*: Tool-internal concurrency control can lead to performance bottlenecks if synchronization is performed on "hot paths" through event handlers. For example, acquiring locks to guard against races on shadow locations in the `access` handler is very expensive, since that handler is called for every memory access.

   To avoid this scenario, RoadRunner provides an optimistic concurrency control mechanism for updating shadow locations. The `putShadow` method of an `AccessEvent e` performs an atomic-compare-and-swap operation: if the current value in the shadow location is the same as `e.shadow`, then this method replaces the shadow location with its argument and returns true. However, if the shadow location is not the same as `e.shadow` (meaning that it was changed by another thread after the event object `e` was initialized), then the method fails and returns false. In this case, RoadRunner also updates `e.shadow` with the current shadow value to facilitate a subsequent retry.

   Thus, if two access event handlers attempt to concurrently modify the same shadow location, only one will succeed, and the other can recognize the conflict and reprocess the event. Using this feature, the `access` method in `LockSet` can be optimized as follows:

```
void access(AccessEvent e) {
  Set held = locksHeld.get(e.thread);
  Set newLs;
  do {
    Set ls = (Set)e.shadow;
    newLs = ls.intersect(held);
  } while (!e.putShadow(newLs));
  if (newLs.isEmpty()) error();
}
```

2.3 Tool Composition

We now extend the core RoadRunner API to support the tool composition model, which was designed to be easy-to-use and to provide reasonable efficiency. (In many cases, composing small tools yields analyses that are comparable in performance to a single large tool.) Our implementation reflects the following design choices:

1. We have designed RoadRunner to make sequential, linear tool composition as efficient as possible, since we have found linear compositions, or *tool chains*, to be most common. [1]
2. Programs often manipulate millions of memory locations. The overhead of maintaining a separate shadow memory location *for each tool* is prohibitively expensive. Thus, RoadRunner only provides a single "shadow location" for each memory

[1] Parallel composition (specified on the command-line as '`-tool=A|B`') is also supported, but it has not been optimized and is more useful for exploratory work than for high-performance implementations.

location, and each tool should modify that shadow location only if it is the location's "current owner." The first tool in the chain is the initial owner, and ownership passes down the tool chain when the owning tool explicitly notifies the next tool in the chain that it wishes to relinquish ownership.

3. Tool chains end with a `LastTool` to terminate event dispatch. All other tools may assume that there is a next tool in line.

Figure 4 extends our initial presentation of the `Tool` class with elements to support sequential composition. In particular, each tool contains a pointer `next` to the next tool in the tool chain.

RoadRunner requires each subclass of `Tool` to conform to the following design guidelines:

1. Event handlers must invoke the same handler on the `next` tool, unless the tool is filtering out that particular event. Figure 5 illustrates this idiom for `acquire` and `release`.
2. To identify which tool owns a shadow location, each tool must have an associated *shadow type* $T <:$ `ShadowVar`, and must only store references of that shadow type in shadow locations. The shadow types of all tools in the tool chain must be distinct.
3. The `makeShadowVar()` method must return an object of this shadow type T.[2]
4. For an access event `e` where the tool owns the shadow location (that is, the shadow value has type T), the handler may either:
   - Retain ownership of the memory location by continuing to store a T object in the location's shadow. In this case, the event should not be passed to the `next` tool.
   - Relinquish ownership by invoking `this.advance(e)`, which updates the shadow location with a value created by the next tool in the chain and then calls that tool to handle the event. Once ownership is relinquished, the shadow location will never again contain a T object, and this tool must not modify it on subsequent accesses.
5. For an access event `e` where the shadow value does not have type T, the handler should dispatch to `next.access(e)`.

The `LockSet` tool defined in Figure 5 conforms to these requirements. Its shadow type is `Set`, its `access` method processes events only when the shadow value is a `Set`, and it relinquishes ownership of a location once a race condition has been observed.

For the `ThreadLocal` tool in Figure 6, the shadow type is `ShadowThread`, and this tool initializes each shadow location with the `ShadowThread` object for the first thread to access that location. Subsequent accesses by that thread are filtered out of the event stream until a second thread accesses the location. At that point, the `ThreadLocal` tool passes ownership to the next analysis. Thus, the composed analysis "`ThreadLocal:LockSet`" applies the `ThreadLocal` tool to filter out thread-local accesses, and then computes the lock set for all thread-shared locations using `LockSet`. In this case, if a race is seen on a location, `LockSet` reports the race and passes ownership to `LastTool`, which sets the shadow location to a type not processed by any user-defined tool.

3. RoadRunner Implementation Details

RoadRunner is written entirely in Java, with no modifications to the underlying Java Virtual Machine, and is roughly 20,000 lines of code in size.

RoadRunner uses a specialized class loader to instrument the classes loaded by the target program at load time. The instrumentor, which is built using ASM [3], augments classes with additional fields and methods to generate the RoadRunner event stream and to store shadow locations. The RoadRunner class loader also gathers the class file metadata (such as type names, field names, source locations, etc.) for later use by tools. We describe the basic instrumentation process for field and array accesses in detail; the instrumentation process for other events is similar.

Figure 7 shows a simple class `A` before and after instrumentation. The shadow locations for each object's fields are stored in the object itself as additional fields of type `ShadowVar`.[3] For the field `int x` in `A`, RoadRunner adds the field `$rr_x` to hold the

[2] Utility tools that never own locations, such as Print or Count, may invoke `next.makeShadowVar()` to pass ownership down the tool chain.

Figure 4: `Tool` **Class with Tool Chain Support**

```
103 abstract class Tool {
104
105   final Tool next;
106
107   void create  (NewThreadEvent e) { next.create(e);  }
108   void acquire (AcquireEvent e)   { next.acquire(e); }
109   void release (ReleaseEvent e)   { next.release(e); }
110   void access  (AccessEvent e)    { next.access(e);  }
111
112   abstract ShadowVar makeShadowVar(AccessEvent e);
113
114   final void advance(AccessEvent e) { ... }
115 }
```

Figure 5: `LockSet` **Tool with Tool Chain Support**

```
115 class LockSet extends Tool {
116   ...
117   void acquire(AcquireEvent e) { ... next.acquire(e); }
118   void release(ReleaseEvent e) { ... next.release(e); }
119
120   void access(AccessEvent e) {
121     if (e.shadow instanceof Set) {
122       Set ls = (Set)e.shadow;
123       Set held = locksHeld.get(e.thread);
124       ls = ls.intersect(held);
125       e.putShadow(ls);
126       if (ls.isEmpty()) {
127         error(); advance(e);
128       }
129     } else {
130       next.access(e);
131     }
132   }
133 }
```

Figure 6: `ThreadLocal` **Tool with Tool Chain Support**

```
133 class ThreadLocal extends Tool {
134
135   ShadowVar makeShadowVar(AccessEvent e) {
136     return e.thread;
137   }
138
139   void access(AccessEvent e) {
140     if (e.shadow instanceof ShadowThread) {
141       if (e.shadow != e.thread) advance(e);
142     } else {
143       next.access(e);
144     }
145   }
146 }
```

[3] RoadRunner also supports a less precise but more space-efficient object-grained instrumentation in which a single `ShadowVar` is used for all object fields (or all elements in an array) [31].

Figure 7: Instrumentation Example

Source Class (before instrumentation)

```
146 class A {
147   int x;
148   void f(int a[]) {
149     x = x + 1;
150     a[11] = 3;
151   }
152 }
```

Resulting Class (after instrumentation)

```
152 class A {
153   int x;
154   ShadowVar $rr_x;
155
156   int $rr_get_x(int accessID, ShadowThread ts) {
157     RR.read(this, accessID, ts);
158     return x;
159   }
160
161   int $rr_put_x(int xv, int accessID, ShadowThread ts) {
162     RR.write(this, accessID, ts);
163     return x = xv;
164   }
165
166   void f(int a[]) {
167     ShadowThread $rr_ts = RR.currentThread();
168     $rr_put_x($rr_get_x(101, $rr_ts)+1, 102, $rr_ts);
169     RR.writeArray(realToShadow(a), 11, 103, $rr_ts);
170     a[11] = 3;
171   }
172 }
```

shadow state for `x` (line 154). Thus, each object roughly doubles in size under RoadRunner, due to these shadow fields.

All reads and writes to the field `x` are replaced by calls the RoadRunner-inserted methods `$rr_get_x` and `$rr_put_x` (e.g., line 149 becomes line 168). In addition to reading or writing `x`, these two methods also call the `RR.read` and `RR.write` methods, which in turn extract the shadow for `x` and dispatch the appropriate event to the tool chain. The accessor method `$rr_get_x` takes the currently executing thread and an `accessID` as parameters. The `accessID` is an integer generated by RoadRunner to uniquely identify each syntactic access in the program source, and it enables the `RR.read` and `RR.write` methods to recover source-level information when generating events. For example, RoadRunner maps the identifier 101 (line 168) to a `FieldAccessInfo` object describing an access to `A.x` at line 149.

The `ShadowThread` for the current thread is passed to every `RR` entry point. To avoid recomputing the current thread for each event, RoadRunner inserts a single call to `RR.currentThread()` at the start of each instrumented method to obtain and then store the current `ShadowThread` in a local variable `$rr_ts`.

RoadRunner uses a different strategy to maintain shadow locations for array elements since it cannot extend the JVM's representation for arrays to directly include shadow data. For each *real* array of size n allocated by the target program, RoadRunner allocates a *shadow* array of type `ShadowVar[]`, also of size n. RoadRunner maintains an internal *real-to-shadow map*, and the *realToShadow* operation (see line 169) uses this map to find the shadow array for the real array being accessed. We describe our optimized map structure and lookup procedure below.

4. Optimizations

RoadRunner relies heavily on the JVM's JIT compiler to optimize the instrumented code and tool chain dispatches. Given that most event handlers are relatively small and the tool chain is fixed at system startup, aggressive method inlining by JITs, such as HotSpot, actually makes most tool chain dispatches and event handling operations run no slower than hand-optimized checkers built from scratch. For example, the version of Atomizer built without RoadRunner was actually a little slower than the version designed as a composition of simpler tools. Achieving this level of performance required designing the RoadRunner data structures and instrumentation strategy to avoid a number of significant bottlenecks. We described some of these below.

Event Reuse. To avoid unnecessary allocation of event objects, RoadRunner creates, for each thread, exactly one object of each `Event` type, and reuses that object for all events of that type generated by that thread.

Access Fast Path Inlining. Despite our best efforts to optimize the event dispatch mechanism, the sheer number of memory accesses causes the overhead of setting up and dispatching `AccessEvent`s to be a bottleneck. Early experience with RoadRunner provided a key insight for greatly reducing this cost: most of the time, typical analyses need only examine the `ShadowThread` and `ShadowVar` to properly handle an access, and do little or no work before returning.

RoadRunner provides an "Access Fast Path" idiom to enable the direct inlining of these fast paths into the target code. Full `AccessEvent`s are then only generated when the analysis falls off the fast path. The following method in the `LockSet` class defines a fast path for reads that succeeds in the common case where the lock set for the accessed location is exactly the (non-empty) set of locks held by the current thread.

```
static boolean readFP(ShadowVar v, ShadowThread ts) {
  return v == locksHeld.get(ts) && !((Set)v).isEmpty();
}
```

The instrumentor generates the following variant of `$rr_get_x` for class `A` that avoids calling the slow path `RR.read` in the common case where the fast path succeeds and returns true:

```
int $rr_get_x(int accessID, ShadowThread ts) {
  if (!LockSet.readFP(this.$rr_x, ts))
    RR.read(this, accessID, ts);
  return x;
}
```

The JIT optimizer typically inlines calls to small methods such as `$rr_get_x`, `readFP`, `locksHeld.get`, and `isEmpty`, resulting in very little overhead for processing most accesses. Write fast paths are specified similarly.

For tool chains with multiple tools, the fast paths are called in the tool chain order and tested sequentially until either (1) they all return false, at which point the slow path `RR.read` is called; or (2) one fast path returns true, meaning it successfully handled the event, in which case no slow path processing is necessary.

Several issues, such as the exact JIT inlining policy and potential interference between threads executing the same fast path, require careful attention and a degree of manual tuning, but judicious use of fast paths can substantially improve performance. If each tool's fast path returns true only in situations where the shadow value has that tool's shadow type and where the tool's access handler would process the event in the same way without passing it along, then this fast path optimization should have no effect other than improved performance.

Real-to-Shadow Map. As mentioned above, RoadRunner uses the *realToShadow* operation to map arrays to their shadow state. Implementing this lookup in an efficient manner is surprisingly tricky, due to it being an extremely heavily used data structure

Tool Chain	Slowdown (x Base Time)				
	Basic	+Event Reuse	+Fast Path Inlining	+Array Lookup	+Decoration Inlining
EMPTY	52.6	8.2	7.2	5.8	5.6
ERASER	52.8	11.8	10.5	10.1	9.4
ERASER:PROTECTINGLOCK:ATOMIZER	54.9	14.8	12.5	10.5	9.8
FASTTRACK	54.1	13.6	9.5	8.2	7.3
FASTTRACK:VELODROME	55.8	15.6	10.9	9.4	8.1
Average	54.0	12.8	10.1	8.8	8.0

Figure 8. ROADRUNNER performance on the JavaGrande benchmarks. ERASER abbreviates THREADLOCAL:READONLY:LOCKSET.

for array-intensive programs, and also due to garbage collection concerns. In particular, we wish to garbage collect real arrays (and their shadows) once the target program no longer references them. ROADRUNNER maintains a three-tier lookup table:

1. The first tier is a per-thread inline cache for each syntactic array access, which contains the eight (real-array/shadow-array) pairs most recently looked up by each thread at that syntactic access.
2. The second tier is a shared `ConcurrentHashMap`, called *Recent*, which also stores (real-array/shadow-array) pairs.
3. The final tier is a shared `WeakHashMap`, called *Attic*. At fixed intervals, all entries stored in the *Recent* table are moved into the *Attic*, to permit garbage collection of real and shadow arrays. (Real arrays referenced only from the *Attic*, where they are stored as weak references, can be reclaimed.)

The *realToShadow* procedure first looks in the per-thread inline cache, then in the *Recent* table, and finally in the *Attic* table, stopping as soon as the array is found (and inserting it into the cache and *Recent* tables if not already there). To further reduce overhead, the ROADRUNNER instrumentor performs a simple data-flow analysis to eliminate redundant lookups.

There are many alternative designs for maintaining the shadow array map, but this three-level strategy has proven straightforward to implement, efficient, and predictable in behavior.

ShadowThread Decoration Inlining. Decorations are convenient, modular, and efficient, but they do introduce extra levels of indirection that can impact performance for heavily-used `ShadowThread` decorations. To reduce this overhead, ROADRUNNER provides a load-time bytecode rewriting mechanism that can dynamically add additional per-tool fields into the `ShadowThread` class. The following example code illustrates how the `LockSet` tool can use this feature to replace the `locksHeld` decoration. The tool simply defines "dummy" accessor and mutator methods (`ts_get_locksHeld` and `ts_set_locksHeld`) of the appropriate types and calls these methods from event handlers:

```
class LockSet extends Tool {

  static Set ts_get_locksHeld(ShadowThread ts) {
    return null;
  }

  static void ts_set_locksHeld(ShadowThread ts, Set s) {
  }

  void acquire(AcquireEvent e) {
    Set held = ts_get_locksHeld(e.thread);
    ...
  }
}
```

ROADRUNNER recognizes the special name prefixes (`ts_get_` and `ts_set_`) at tool load time and inserts a corresponding field `$rr_locksHeld` into the `ShadowThread` class. It then rewrites the dummy accessor and mutator methods to manipulate this new field. While this lightweight mechanism has been sufficient for our purposes, aspects [17] provide more general extension mechanisms.

Performance. We have found ROADRUNNER to provide acceptable performance in practice. Figure 8 shows the average slowdown of five ROADRUNNER analysis configurations when checking the programs from the JavaGrande benchmark suite [14] on a dual quad-core machine running Mac OSX 10.6 and HotSpot 1.6. The second column describes the slowdown of the basic ROADRUNNER framework, with all of the optimizations from this section turned off. The subsequent columns then show the benefits as each additional optimization is turned on. Cumulatively, the optimizations enable these analyses to have slowdowns of less than 10x. This is quite competitive with specialized research prototypes built to study individual analyses (e.g., [9, 21, 24]). As reported in earlier papers [10, 11], ROADRUNNER checkers can scale to systems as large as the Eclipse development environment [7].

5. Related Work

ROADRUNNER uses the ASM [3] library to perform rewriting and instrumentation of the target program's `.class` files at load time. A number of other systems provide features comparable to ASM, e.g., BCEL [2]. The SOOT framework [26] is similar, but the rewriting is performed on a somewhat higher-level intermediate language of the kind traditionally used for compiler optimizations. ROADRUNNER provides a higher-level and more focused API than these more general rewriting engines, since ROADRUNNER analyses are specified in terms of event stream handling, and not in terms of rewriting operations on target program code. Indeed, much of ROADRUNNER's code base is essentially focused on lifting the bytecode-based virtual machine abstraction to this event-based abstraction level.

Sofya [18] is a dynamic program analysis framework that has similar goals to ROADRUNNER. A key difference is that Sofya runs the target program in its own JVM, which prevents event processing from interfering with program behavior, but introduces additional overheads for inter-JVM communication. Sofya provides an Event Description Language to specify which events are of interest to the analysis, whereas ROADRUNNER in general communicates all events to the analysis back-end, since most analyses (e.g., race detectors) will typically be interested in most or all events. Thus, ROADRUNNER is designed to support a high-bandwidth event stream, which in turn requires an architecture where the target program and the analysis run on the same JVM.

CalFuzzer [15] is a framework for "active testing" of concurrent programs. It performs bytecode rewriting using SOOT and runs in the same memory space as the target program. Each analysis provides callback functions to handle memory accesses, synchronization operations, etc. CalFuzzer appears not to directly support shadow locations for fields and for array elements. Instead, the CalFuzzer framework passes integer identifiers to the callback functions to identify memory locations and locks. Analyses can then allocate their own arrays indexed by these identifiers, but under this approach shadow state is likely not garbage-collected, which

may be problematic for large benchmarks. CalFuzzer does include explicit support for perturbing the scheduler for active testing.

ATOM [25] provides a framework for instrumenting native code Alpha programs with callbacks to analysis tools. The instrumentation process permits a flexible choice of which events to instrument, but does not provide explicit support for shadow state. Some ATOM tools, most notably Eraser [24], implement this feature on top of ATOM.

The Valgrind framework [20] supports heavyweight instrumentation of binary code programs, where every register has an associated *shadow register* that contains meta-data about the register's value. Valgrind works via *disassembly-and-resynthesis*, performing the instrumentation on an higher-level intermediate representation, somewhat like the SOOT framework discussed above. The Valgrind distribution provides a number of analysis tools, including some that focus on race conditions. In comparison, RoadRunner uses a more lightweight *copy-and-annotate* approach to instrumentation, which is sufficient for our purposes.

6. RoadRunner in Practice

RoadRunner has proven invaluable for building and evaluating analyses. Implementing a first prototype of a new analysis often takes a few days rather than the weeks or months required to build a system from scratch. Much of this improvement comes from the RoadRunner event stream model, which matches the level of abstraction with which we formulate analyses.

In addition, initial versions of RoadRunner analyses can often be applied to large-scale systems such as Eclipse, because RoadRunner deals with and hides many of the complexities involved in scaling to such systems. This scalability and easy experimentation is critical for evaluating and refining new analyses.

Tool composition in RoadRunner has also played a major role in how we design, implement, test, and validate dynamic analyses. It enables us to express complex algorithms as the composition of simpler, modular ones; to reuse analysis components unchanged; and to insert monitoring and debugging filters into the tool chain without cluttering the analysis code.

Acknowledgments

This work was supported in part by the NSF under Grants 0341179, 0341387, 0644130, and 0707885 and by a Sloan Foundation Fellowship. Ben Wood helped implement array tracking and several analyses. Jaeheon Yi, Caitlin Sadowski, and Catalin Iordan helped test and validate the RoadRunner framework.

References

[1] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD*, pages 51–60, 2006.

[2] Byte Code Engineering Library. `http://jakarta.apache.org/bcel/`, 2007.

[3] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[4] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, pages 3–12, 2009.

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[6] M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, pages 761–770, 2001.

[7] The Eclipse programming environment, version 3.4.0. Available at `http://www.eclipse.org`, 2009.

[8] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

[9] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.

[10] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[11] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *PLDI*, 2010.

[12] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, pages 293–303, 2008.

[13] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, pages 175–190, 2004.

[14] Java Grande Forum. Java Grande benchmark suite. Available at `http://www.javagrande.org/`, 2008.

[15] P. Joshi, M. Naik, C.-S. Park, and K. Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV*, pages 675–681, 2009.

[16] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120, 2009.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[18] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. *ICSE Companion*, pages 51–52, 2007.

[19] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1988.

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, June 2007.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, 2003.

[22] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[23] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, pages 394–409, 2009.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15(4):391–411, 1997.

[25] A. Srivastava and A. Eustace. ATOM : A system for building customized program analysis tools. In *PLDI*, pages 196–205, 1994.

[26] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction*, pages 18–34, 2000.

[27] C. von Praun and T. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

[28] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[29] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, pages 1–14, 2005.

[30] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *PADTAD*, 2009.

[31] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.