A Case Study on Executing Instrumented Code in Java PathFinder

Karl Palmskog, Farah Hariri, and Darko Marinov
University of Illinois at Urbana-Champaign
{palmskog, hariri2, marinov}@illinois.edu

ABSTRACT

Dynamic program analysis is widely used for detecting faults in software systems. Dynamic analysis tools conceptually either use a modified execution environment or inject instrumentation code into the system under test (SUT). Tools based on a modified environment, such as Java PathFinder (JPF), have full access to the state of the SUT but at the cost of a higher runtime overhead. Instrumentation-based tools have lower overhead but at the cost of less convenient control of runtime such as thread schedules. Combinations of these two approaches are largely unexplored, and to our knowledge, have never been done in JPF. We present a case study of adapting an existing instrumentation-based tool to run inside JPF. To keep the instrumentation unchanged, we limited our changes to the code invoked by the instrumentation. Ultimately, the required changes were few and essentially reduce to properly dividing the analysis-related state and logic between the JPF and host JVM levels. Others can benefit from our experience to quicker adapt their instrumentation-based tools to run in JPF.

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

Keywords: dynamic analysis, instrumentation, Java PathFinder

1. INTRODUCTION

Dynamic program analysis is a popular technique for quality assurance of software systems. It can be used both for opportunistic and exhaustive fault detection, and for comprehension of semantic properties of the system under test (SUT). In particular, concurrent programs are a priority target for dynamic analysis, partly due to their unique susceptibility to faults that often cannot be reliably reproduced, such as atomicity violations and deadlocks.

Tools and techniques for dynamic analysis can be divided into those that use a modified execution environment and those that do not. Both approaches have their advantages and disadvantages. If the execution environment is not modified, e.g., if a Java program runs in a regular JVM, some form of code instrumentation must be performed by the tool. This involves injecting additional code into the SUT at locations of interest; when executed, this code generates *events* that are intercepted and processed by the tool, while preserving the functional behavior of the SUT. Leaving the environment in pristine condition allows taking full advantage of optimizations such as JIT compilation, and lets users explicitly control the boundary between the SUT and other artifacts such as libraries, at the cost of diminished control over runtime properties and less convenient access to the SUT state. In comparison, executing a SUT inside a modified environment, such as Java PathFinder (JPF) [8], allows full access to the SUT state and control of runtime properties such as thread schedules, but at a higher runtime cost. Running instrumented SUT code generated by a dynamic analysis tool inside a modified execution environment can capture key advantages of both approaches, but is largely unexplored in the literature [1, p. 55]. To the best of our knowledge, no combination of these approaches has been reported previously in the context of JPF.

We present a case study of adapting an instrumentation-based tool to run inside JPF. The original tool instruments Java bytecode at locations of interest to concurrency, such as thread creations, methods calls, and field accesses [3]; Section 2 shows a simple example. Our main motivation was to gain a convenient control of thread schedules; we use JPF as we are familiar with it. Because the original tool only considers single traces of the SUT, we wanted to use JPF as a simulator, i.e., for execution without backtracking—we do not explore schedules within one JPF run but externally invoke JPF multiple times with different parameters to obtain various schedules.

To keep the development effort of our adaptation low, a key requirement was to *keep the instrumentation of SUTs unchanged*. Thus, instrumented SUTs can still be analyzed outside of JPF, and we did not have to touch the tool's fairly complex bytecode-injection logic. Instead, we limited our changes to the *event-processing logic*, i.e., the code called by the operations injected into the SUT. While one could, in principle, reimplement the event processing using JPF listeners to execute uninstrumented SUTs, it irrevocably ties the tool to JPF, and in our case would have forced us to reimplement the complex code-injection logic.

Section 3 presents in detail the changes we performed; overall, they were not extensive. Section 5 summarizes the tasks required for these changes, which essentially reduce to dividing state and logic related to event processing between the JPF and the host JVM. Following these tasks for an instrumentation-based analysis tool can increase the scope and accuracy of its analysis without the need for extensive refactoring. We believe our approach could be automated to a large extent, with users configuring division of logic and state via configuration files and annotations. Moreover, developers of new analyses can consider implementing them first in JPF (for easier debugging and experimentation) but *using instrumentation rather than JPF listeners*, to enable easier deployment directly on the JVM (for faster and more scalable analysis, with a potentially more complex implementation).

2. ORIGINAL DYNAMIC ANALYSIS TOOL

The purpose of the original dynamic analysis tool [3] is to infer atomicity properties of concurrent programs; for lack of space, we do not describe it in detail. Instead, we give an overview of the key points via a simple example. Our example instrumented SUT is a multithreaded download manager that spawns a configurable number of worker threads to download URLs from a shared list. Fig. 1 shows the program source, with code in-

```
 1 public class Downloader {
 2   public static void main(String[] args)
 3    throws InterruptedException, MalformedURLException {
 4     ThreadData _td = ThreadData.current();
 5     Updater.methodEntry("main", _td);
 6     final URLManager manager = new URLManager();
 7     String[] pages = {"wiki", "timeline", "roadmap"};
 8     for (String p : pages) {
 9       manager.addURL
10        (new URL("http://...nasa.gov/trac/jpf/" + p));
11     }
12     int numWorkers = Integer.parseInt(args[0]);
13     Thread[] threads = new Thread[numWorkers];
14     for (int i = 0; i < numWorkers; i++) {
15       threads[i] = new Thread() {
16         public void run() {
17           ThreadData _td = ThreadData.current();
18           Updater.methodEntry("run", _td);
19           URL url;
20           while ((url = manager.getNextURL()) != null)
21             download(url);
22           Updater.methodExit(_td);
23         }
24         private void download(URL url) {
25           ThreadData _td = ThreadData.current();
26           Updater.methodEntry("download", _td);
27           /* ... call to downloading library ... */
28           _td.counter++;
29           Updater.methodExit(_td);
30         }
31       };
32       threads[i].start();
33       Updater.fork(threads[i], _td);
34     }
35     for (Thread t : threads) {
36       t.join();
37       Updater.join(t, _td);
38     }
39     Updater.methodExit(_td);
40   }
41 }
42 class URLManager {
43   private List<URL> urls = new ArrayList<URL>();
44   synchronized URL getNextURL() {
45     ThreadData _td = ThreadData.current();
46     Updater.methodEntry("getNextURL", _td);
47     URL url = urls.isEmpty() ? null : urls.remove(0);
48     Updater.fieldRefGet(this, urls, "urls", _td);
49     Updater.methodExit(_td);
50     return url;
51   }
52   synchronized void addURL(URL url) {
53     ThreadData _td = ThreadData.current();
54     Updater.methodEntry("addURL", _td);
55     urls.add(url);
56     Updater.fieldRefGet(this, urls, "urls", _td);
57     Updater.methodExit(_td);
58   }
59 }
```

Figure 1: SUT with instrumentation shown in bold.

jected by the instrumentation in bold. The main method creates a URLManager instance (manager) and adds some URLs to the shared list (lines 7–11). Then, it creates the specified number of threads (numWorkers), each of which enters a loop that retrieves and downloads URLs (lines 20–21). Finally, after all URLs have been downloaded, all threads are joined (lines 35–38).

The code injected into the SUT calls methods in two classes, Updater and ThreadData, whose simplified definitions are listed in Fig. 2. The Updater class shows methods for five events; the real tool processes many more events. The instrumentation adds calls to methodEntry and methodExit at the start and end, respectively, of each SUT method (e.g., lines 18 and 22 in Fig. 1). It also adds calls to fork and join where a thread is started and joined, respectively (e.g., lines 33 and 37 in Fig. 1). Because the original instrumentation is performed at the bytecode level without analyzing the class hierarchy, fork and join can be called for objects that are not instances of Thread. Hence, the event-processing code performs runtime checks (line 6 of Updater in Fig. 2). Finally, the instrumentation adds calls to fieldGetRef after a field is read (lines 48 and 56 in Fig. 1).

ThreadData is a bookkeeping class used to track data related to SUT threads. The field threadId introduces a logical numbering of SUT threads. The field counter counts the number of calls to external libraries, which amounts here to calls to download a URL (lines 27–28 in Fig. 1). ThreadData also holds observation data produced when events are processed. The forThread method in ThreadData looks up, for a given Thread instance, its corresponding ThreadData instance, creating it if required. The methods in Updater contain the logic that updates the observation data and additionally maintains metadata for SUT objects using the map objectsData. At the end of the SUT execution, all observation data is summarized and written to file.

Two design decisions on event processing are important to point out. First, most methods analyze only the identity of the objects passed in, e.g., to track which thread accesses which object. The only exception are the fork and join methods that also check the type of the objects and perform map lookups via forThread. Second, the instrumentation mostly calls methods on Updater and ThreadData to represent events, but it directly accesses the field counter (rather than calling some method, say, incrementCounter), presumably for performance reasons. We are not aware of the rationale behind these decisions for the original tool. However, our goal was to *make the instrumented code work in JPF, without any changes to the instrumentation.*

3. INSTRUMENTED CODE IN JPF

Simply running an instrumented SUT in JPF proved to be problematic, not least because the observation data summarizer calls methods in java.io classes that are currently unavailable inside JPF. In addition, mixing SUT code with extensive code added by instrumentation inside JPF runs the risk of invalidating the analysis output, because JPF would be unable to distinguish between the real SUT and instrumentation at runtime. Finally, unnecessarily executing code at the JPF level is expensive in terms of both time and memory.

In light of these concerns, we decided to migrate instrumentation code to the JVM level by using the JPF's Model Java Interface (MJI). We introduced custom JVM equivalents of Updater and ThreadData, named JVMUpdater and JVMThreadData, respectively, to hold the analysis-related state and logic. Because the instrumentation code directly manipulates counter in ThreadData, that field could not simply be moved to JVMThreadData. Instead, we had to duplicate the field at both levels and to keep the levels consistent when mapping ThreadData instances to JVMThreadData instances in our peer classes. Handling instrumentation-injected calls to methods that are not passed SUT objects directly, e.g., methodEntry and methodExit, was straightforward. After removing the method bodies and adding the **native** keyword to signatures in Updater, moving the logic to JVMUpdater, and modifying the MJI peer JPF_dyn_Updater to proxy calls, no other changes are necessary. Fig. 3 illustrates the conceptual division between JPF and the host JVM.

In contrast, the methods fork and join required splitting the event processing logic between the stripped-down Updater class and the JVMUpdater class. The reason for these splits is that the lookup performed by forThread is difficult to perform at the JVM level, since the map between Thread and ThreadData lives

```
public class Updater {
 private static Map<Object,ObjectData> objectsData =
   new WeakHashMap<Object,ObjectData>();
 public static synchronized void fork
  (Object childThread, ThreadData parentThreadData) {
   if (!(childThread instanceof Thread)) return;
   ThreadData childThreadData =
     ThreadData.forThread((Thread) childThread);
   /* ... logic updating observationData ... */
 }
 public static synchronized void join
  (Object thread, ThreadData waitingThreadData) {
   /* ... similar to fork .. */
   /* ... logic updating observationData ... */
 }
 public static void methodEntry
  (String methodName, ThreadData threadData) {
   /* ... logic updating observationData ... */
 }
 public static void methodExit
  (ThreadData threadData) {
   /* ... logic updating observationData ... */
 }
 public static void fieldRefGet(Object owner,
  Object val, String fname, ThreadData threadData) {
   /* ... logic updating observationData and objectsData ... */
 }
}
```

```
public class ThreadData {
 public int counter;
 private int threadId;
 private ObservationData observationData;
 public ThreadData(int threadId) {
   this.threadId = threadId;
   this.counter = 0;
   this.observationData = new ObservationData();
 }
 public ObservationData getObservationData() {
   return observationData;
 }
 public int getThreadId() {
   return threadId;
 }
 public static ThreadData current() {
   return localData.get();
 }
 public static synchronized ThreadData forThread
  (Thread thread) {
   ThreadData td = threadData.get(thread);
   if (td == null) {
     td = new ThreadData(nextThreadId++);
     threadData.put(thread, td);
   }
   return td;
 }
 private static int nextThreadId = 0;
 private static Map<Thread, ThreadData> threadData =
   new WeakHashMap<Thread, ThreadData>();
 private static ThreadLocal<ThreadData> localData =
   new ThreadLocal<ThreadData>() {
     protected ThreadData initialValue() {
       ThreadData td =
         forThread(Thread.currentThread());
       ThreadDataSummarizer.register(td);
       return td;
     }
   };
}
```

Figure 2: Classes used by instrumentation-injected code.

at the JPF level. To avoid this complication, we introduced the methods `forkNative` and `joinNative`, which are invoked after the respective map lookups. These calls are then proxied via the MJI peer to `JVMUpdater`.

The definitions of the classes resulting from the change of `Updater` are listed in Fig. 4. As is emphasized in the code, the `objectsData` field has moved from `Updater` to `JVMUpdater`. The corresponding classes for `ThreadData` are listed in Fig. 5. The `observationData`

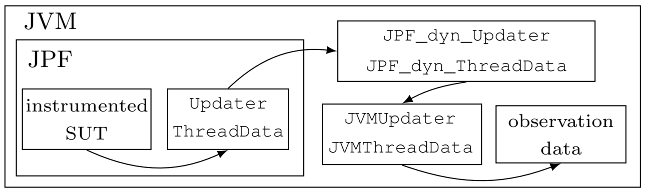

Figure 3: Architecture of the changed tool.

field has moved from `ThreadData` to `JVMThreadData`, and, to make sure that `JVMThreadData` objects are created for each new thread, the method `initialValueNative` is called at the JPF level. In the peer class `JPF_dyn_ThreadData`, the corresponding method creates a new `JVMThreadData` instance and keeps track of the mapping between it and the associated `ThreadData` object.

The signature of the method `fieldRefGet` changes in the transition to the JVM level; the parameters `owner` and `value` change their types from `Object` to **`int`**. Because only the identities of these objects are used for map lookups in the event-processing logic, and the map `objectsData` can use **`int`** lookups via autoboxing, no further changes are required.

After applying the outlined changes, we managed to analyze 18 out of 19 instrumented small test programs unchanged inside JPF; the remaining program had to be adapted slightly. We further successfully analyzed five out of six of the larger benchmark programs [3], with the final program failing due to extensive calls to `java.io` methods currently unavailable in JPF. We measured the runtime of the five working benchmark programs on an Intel Core2 Quad 2.33 GHz computer with 8 GB of memory. Table 1 shows the user execution time, i.e., the user-mode CPU time for each benchmark program, uninstrumented directly on the JVM (*JVM*), instrumented directly on the JVM (*JVM+i*), uninstrumented in JPF (*JPF*), and instrumented in JPF (*JPF+i*). With all benchmarks being multithreaded, user execution time has the advantage of eliminating the real time differences that arise as the JVM leverages multiple processor cores, which the single-threaded JPF is unable to do. *tsp2* runs exceptionally slowly in JPF, most likely due to its extensive use of array operations. The execution times are the average of 10 runs, except for *tsp2* in JPF, which are for single runs due to their long duration. The time overhead (of using instrumentation and JPF) is the ratio between the execution time and the corresponding uninstrumented JVM execution time. The last three columns show the memory usage, with and without instrumentation (plus the computed overhead), as reported in the JPF output statistics summary.

As anecdotal evidence of the benefits of JPF, while running *collections* in JPF, we encountered a thread schedule that is unlikely to be exercised by the JVM but that revealed a fault in the random fuzzer driving the SUT execution. Specifically, the fuzzer called the method `nextInt` in `java.util.Random` on the result of an expression that performed a subtraction; in the unlikely schedule, the result of the expression was negative. This shows that running instrumented code inside JPF can offer immediate benefits.

4. RELATED WORK

The dynamic binary instrumentation tool Valgrind [5] combines instrumentation and a modified execution environment, in a way conceptually similar to what we do. In Valgrind, machine code is translated to an intermediate format, instrumented, and recompiled to run in a controlled environment with a simulated CPU. However, in our approach, the modified environment is unaware

```java
public class Updater {
  public static synchronized void fork
   (Object childThread, ThreadData parentThreadData) {
    if (!(childThread instanceof Thread)) return;
    ThreadData childThreadData =
     ThreadData.forThread((Thread) childThread);
    forkNative(childThreadData, parentThreadData);
  }
  native public static void forkNative
   (ThreadData childThreadData,
    ThreadData parentThreadData);
  public static synchronized void join
   (Object thread, ThreadData waitingThreadData) {
    /* ... similar to fork ... */
    joinNative(threadData, waitingThreadData);
  }
  native public static void joinNative
   (ThreadData threadData,
    ThreadData waitingThreadData);
  native public static void methodEntry
   (String methodName, ThreadData threadData);
  native public static void methodExit
   (ThreadData threadData);
  native public static void fieldRefGet(Object owner,
   Object val, String fn, ThreadData threadData);
}
```

```java
public class JPF_dyn_Updater extends NativePeer {
  @MJI public static void forkNative
   (MJIEnv env, int cref, int ctref, int ptref) {
    JVMUpdater.fork
     (JPF_dyn_ThreadData.remap(env, ctref),
      JPF_dyn_ThreadData.remap(env, ptref));
  }
  @MJI public static void joinNative
   (MJIEnv env, int cref, int tref, int wtref) {
    /* ... similar to forkNative ... */
  }
  @MJI public static void methodEntry
   (MJIEnv env, int cref, int mref, int tref) {
    JVMUpdater.methodEntry
     (env.getStringObject(mref),
      JPF_dyn_ThreadData.remap(env, tref));
  }
  @MJI public static void methodExit
   (MJIEnv env, int cref, int tref) {
    JVMUpdater.methodExit
     (JPF_dyn_ThreadData.remap(env, tref));
  }
  @MJI public static void fieldRefGet(MJIEnv env,
   int cref, int oref, int vref, int fref, int tref) {
    JVMUpdater.fieldRefGet
     (oref, vref, env.getStringObject(fref),
      JPF_dyn_ThreadData.remap(env, tref));
  }
}
```

```java
public class JVMUpdater {
  private static Map<Object,ObjectData> objectsData =
    new WeakHashMap<Object,ObjectData>();
  public static void fork
   (JVMThreadData childThreadData,
    JVMThreadData parentThreadData) {
    /* ... logic updating observationData ... */
  }
  public static void join(JVMThreadData threadData,
   JVMThreadData waitingThreadData) {
    /* ... logic updating observationData ... */
  }
  public static void methodEntry
   (String methodName, JVMThreadData threadData) {
    /* ... logic updating observationData ... */
  }
  public static void methodExit
   (JVMThreadData threadData) {
    /* ... logic updating observationData ... */
  }
  public static void fieldRefGet(int owner, int val,
   String fname, JVMThreadData threadData) {
    /* ... logic updating observationData and objectsData ... */
  }
}
```

Figure 4: Changed classes for `Updater`.

```java
public class ThreadData {
  public int counter;
  private int threadId;
  public ThreadData(int threadId) {
    this.threadId = threadId;
    this.counter = 0;
  }
  public static ThreadData current() {
    return localData.get();
  }
  public static synchronized ThreadData forThread
   (Thread thread) {
    ThreadData td = threadData.get(thread);
    if (td == null) {
      td = new ThreadData(nextThreadId++);
      threadData.put(thread, td);
      initialValueNative(td);
    }
    return td;
  }
  native public static void initialValueNative
   (ThreadData td);
  private static int nextThreadId = 0;
  private static Map<Thread, ThreadData> threadData =
    new WeakHashMap<Thread, ThreadData>();
  private static ThreadLocal<ThreadData> localData =
    new ThreadLocal<ThreadData>() {
      protected ThreadData initialValue() {
        return forThread(Thread.currentThread());
      }
    };
}
```

```java
public class JPF_dyn_ThreadData extends NativePeer {
  private static Map<Integer, JVMThreadData> map =
    new HashMap<Integer, JVMThreadData>();
  public static JVMThreadData remap
   (MJIEnv env, int tdref) {
    int tdId = env.getIntField(tdref, "threadId");
    JVMThreadData td = map.get(tdId);
    td.counter = env.getIntField(tdref, "counter");
    return td;
  }
  @MJI public static void initialValueNative
   (MJIEnv env, int rcls, int tdref) {
    int threadId = env.getIntField(tdref, "threadId");
    JVMThreadData td = new JVMThreadData(threadId);
    map.put(threadId, td);
    ThreadDataSummarizer.register(td);
  }
}
```

```java
public class JVMThreadData {
  public int counter;
  private int threadId;
  private ObservationData observationData;
  public JVMThreadData(int threadId) {
    this.threadId = threadId;
    this.counter = 0;
    this.observationData = new ObservationData();
  }
  public ObservationData getObservationData() {
    return observationData;
  }
  public int getThreadId() {
    return threadId;
  }
}
```

Figure 5: Changed classes for `ThreadData`.

of the instrumentation, and our key point is to keep the code-injection logic unchanged.

The JPF core is a virtual machine running on a host JVM. By default, JPF explores all choice points in a SUT state space, but other uses can be configured. Havelund and Roşu [4] first suggested that a stateless model checker could generate traces for programs running inside runtime monitoring environments, which is similar to our use of JPF for controlling thread schedules.

Program	**Time (s)**				**Time overhead vs.** ***JVM***			**Memory (MB)**		**Mem. ovh.**
	JVM	*JVM+i*	*JPF*	*JPF+i*	*JVM+i*	*JPF*	*JPF+i*	*JPF*	*JPF+i*	*JPF+i*
collections	0.58	3.55	24.98	135.76	6.18	43.43	236.06	212.00	831.10	3.92
elevator	0.10	1.07	8.07	12.34	11.23	84.81	129.78	150.00	150.00	1.00
jcurzez1	0.20	3.57	155.94	244.91	17.86	780.07	1225.18	224.60	290.00	1.29
jcurzez2	0.19	3.75	161.45	258.28	20.23	870.84	1393.11	237.40	343.50	1.45
tsp2	0.23	9.29	32807.90	92943.21	39.96	141109.24	399755.76	4323.00	6905.00	1.60

Table 1: Time and memory overhead of instrumented code on JVM and JPF for five original benchmarks [3].

The jpf-nhandler [6] extension to JPF allows configuring selected SUT methods to be executed at the JVM level. By default, jpf-nhandler re-instantiates at the JVM level all objects passed as parameters to the selected methods, which can be costly and fails for some objects, e.g., `Thread` instances. However, jpf-nhandler can be configured to maintain looser correspondences between JPF and JVM objects, and to persist objects created at the JVM level. Hence, our division of analysis-related state and logic between JPF and JVM could also be implemented through jpf-nhandler. A previous effort related to jpf-nhandler was mixed execution of code at the JPF and JVM levels [2], but no previous work reports executing (unchanged) instrumented code in JPF.

Tools that enable control of thread schedules by instrumenting Java bytecode go back over a decade [7]. We did not pursue this approach as we wanted to avoid integrating instrumentation for controlling threads with the existing instrumented code.

5. LESSONS LEARNED AND FUTURE WORK

Using *both* instrumentation *and* JPF may initially be perceived as an "overkill" because one can implement an analysis using only instrumentation or only JPF. However, our goal was to enable control of thread schedules for an existing, instrumentation-based tool. Overall, the changes we made to enable running instrumented SUTs inside JPF were few but subtle. We believe our approach is applicable to similar instrumentation-based tools and offers a way to increase their sophistication (e.g., to control non-deterministic choices such as thread schedules) with little effort. The main obstacle is to decouple the logic that queries SUT objects from the logic that updates the analysis-related state.

As it turned out, the logic updating the SUT metadata in the tool we considered depended almost exclusively on the type of event and a few pieces of event context, e.g., the thread in which the event took place and the identity of the callee object. This allowed method calls from instrumented code to be straightforwardly proxied via MJI to the JVM level. Event-processing logic that is entangled with sophisticated queries of SUT state would have required more effort to port. Observation data, i.e., the SUT metadata updated by the logic, is ideally kept completely at the JVM level for reasons of performance and noninterference with JPF-based analysis. Because the instrumentation produces code that directly accesses a field (namely, `counter`), we could not fully migrate all SUT metadata outside the reach of JPF. This is a strong argument for only injecting method calls into a SUT.

In summary, we carried out the following tasks to run instrumented SUT code in JPF: (1) Introduce JVM-level versions of classes called by injected code. (2) Transfer as much state as possible from JPF-level classes to the JVM-level classes. (3) Split up the event-processing logic that accesses SUT state and the logic that updates metadata; place the former at the JPF level and the latter at the JVM level. (4) Mark key JPF-level method signatures with **`native`** and create MJI peer classes that proxy calls to the JVM, synchronizing state where necessary. We believe most of these tasks could be automated if key parameters are specified by the user as annotations and options in property files. Hence, future work includes developing a JPF extension, possibly on top of jpf-nhandler, to assist in running instrumented code inside JPF.

Other future work is possible. First, our *JPF+i* tool stores nearly all observation data at the JVM level, which precludes the use of backtracking in JPF, and thus exploration of several thread schedules for a single run. By checkpointing observation data at JPF choice points, it would be possible to enable backtracking. Second, minimizing memory requirements of *JPF+i* is another important issue. For example, migrating the `WeakHashMap` field `objectsData` from `Updater` to `JVMUpdater` means that map entries for garbage-collected SUT objects are no longer dropped automatically and leak memory. They could be dropped by adding custom JPF listeners to remove corresponding JVM objects when their SUT objects are garbage-collected by JPF. Third, there is a substantial duplication of functionality in the analysis arising from the instrumented code and the analysis available inside JPF. For example, the instrumentation in *JPF+i* includes its own data-race analysis, while the same functionality is already available in JPF.

The complete executable code of our running example is at `http://mir.cs.illinois.edu/farah/artifacts/jpf_example.zip`.

Acknowledgments

We thank Peter Dinges, Marcelo d'Amorim, Milos Gligoric, and the reviewers for their comments. This work was partially supported by NSF Grant Nos. CCF-1012759 and CCF-1438982.

6. REFERENCES

[1] C. Artho. *Combining Static and Dynamic Analysis to Find Multithreading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.

[2] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM*, 2006.

[3] P. Dinges, M. Charalambides, and G. Agha. Automated inference of atomic sets for safe concurrent execution. In *PASTE*, 2013.

[4] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, Mar. 2004.

[5] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[6] N. Shafiei and F. van Breugel. Automatic handling of native methods in Java PathFinder. In *SPIN*, 2014.

[7] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. *ENTCS*, 70(4):142–157, 2002.

[8] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.