JMVX: Fast Multi-threaded Multi-version Execution and Record-Replay for Managed Languages

DAVID SCHWARTZ, University of Illinois at Chicago, USA
ANKITH KOWSHIK, University of Illinois at Chicago, USA
LUÍS PINA, University of Illinois at Chicago, USA

Multi-version execution (MVX) is a technique that deploys many equivalent versions of the same program — variants — as a single program, with direct applications in important fields such as: security, reliability, analysis, and availability. MVX can be seen as "online Record/Replay (RR)", as RR captures a program's execution as a log stored on disk that can later be replayed to observe the same execution. Unfortunately, current MVX techniques target programs written in C/C++ and do not support programs written in managed languages, which are the vast majority of code written nowadays.

This paper presents the design, implementation, and evaluation of JMVX— a novel system for performing MVX and RR on programs written in managed languages. JMVX supports programs written in Java by intercepting automatically identified non-deterministic methods, via a novel dynamic analysis technique, and ensuring that all variants execute the same methods and obtain the same data. JMVX supports multi-threaded programs, by capturing synchronization operations in one variant, and ensuring all other variants follow the same ordering. We validated that JMVX supports MVX and RR by applying it to a suite of benchmarks representative of programs written in Java. Internally, JMVX uses a circular buffer located in shared memory between JVMs to enable fast communication between all variants, averaging 5% |47% performance overhead when performing MVX with multithreading support disabled|enabled, 8% |25% when recording, and 13% |73% when replaying.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Software reliability*; *Software maintenance tools.*

Additional Key Words and Phrases: Multi-version execution, record replay, deterministic replay, reproducible debugging

ACM Reference Format:
David Schwartz, Ankith Kowshik, and Luís Pina. 2024. JMVX: Fast Multi-threaded Multi-version Execution and Record-Replay for Managed Languages. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 329 (October 2024), 29 pages. https://doi.org/10.1145/3689769

1 Introduction

Multi-version execution (MVX) is a technique that allows developers to deploy many equivalent versions of the same program — variants — as a single program. MVX has direct applications in many fields, such as: software security [15, 23, 29, 46, 47], in which each variant votes on security-sensitive actions before performing them; software reliability [1, 8, 20, 27, 39], which can survive crashes in one variant by allowing other variants to immediately resume service; software analysis [32, 34, 49], in which expensive and incompatible analyses run behind the scenes in separate variants; and

Authors' Contact Information: David Schwartz, dschwa23@uic.edu, University of Illinois at Chicago, Chicago, Illinois, USA; Ankith Kowshik, akowsh2@uic.edu, University of Illinois at Chicago, Chicago, Illinois, USA; Luís Pina, luispina@uic.edu, University of Illinois at Chicago, Chicago, Illinois, USA.

ACM 2475-1421/2024/10-ART329
https://doi.org/10.1145/3689769

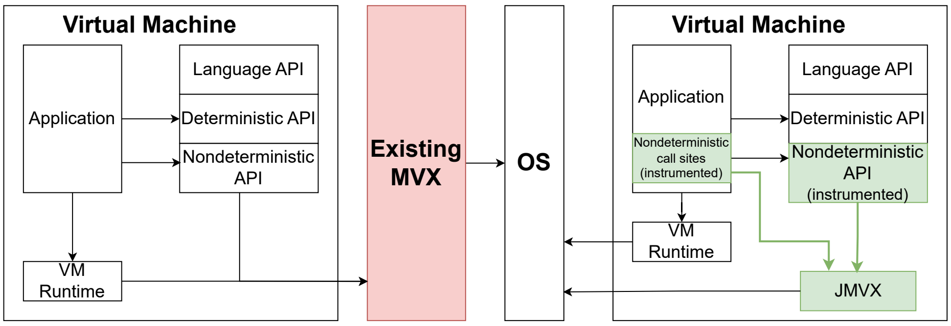

Fig. 1. Overview of JMVX and how it differs from existing MVX approaches. JMVX instruments relevant non-deterministic APIs (*e.g.,* `FileInputStream.read`), or their call-sites within the application; and redirects execution via JMVX's runtime (green arrows). JMVX allows VM runtime management calls to continue without interception. Typical MVX intercepts all interactions with the underlying OS, which causes abrupt termination of programs due to benign differences (*e.g.,* different order when loading classes).

software availability [19, 33, 37, 38], which performs reliable software updates with zero downtime in variants in the background. MVX is related to Record-Replay (RR), as it can be seen as "online RR". RR makes two processes behave exactly the same by running one process first — the recorder — which creates a log saved to disk, and another process later — the replayer — which uses the log to replicate the recorder's execution. RR is an important tool for debugging because it can record a bug made in production and replay it deterministically in development, and has been adopted in practice [7, 24, 30].

Most effort on supporting MVX and RR has been devoted to programs written in C/C++ and compiled to native binaries [1, 8, 15, 19, 20, 23, 24, 27, 29, 30, 32–34, 37, 39, 46, 47, 49]. MVX support for programs written in managed languages, such as Java, remains elusive. Unfortunately, current MVX approaches do not support running high-level language virtual machines (HLLVMs), such as the Java Virtual Machine (JVM) because of two fundamental issues. First, HLLVMs typically exercise edge cases that MVX solutions do not support, such as self-modifying code (*e.g.,* Just-In-Time compilers typically found in high performant HLLVMs). Second, inherent *benign divergences*, differences in a program's execution that result in the same output, cause existing MVX techniques to fail on HLLVMs [20, 32], even for deterministic programs. For instance, the same deterministic Java program can load different classes in a different order (Section 3.4), which causes MVX to terminate as it detects variants to have diverged. The same is true with different compiler optimizations and different Garbage Collection cycles in different runs of the same deterministic program.

This paper presents JMVX, a novel technique for both MVX and RR of programs written in a managed language — Java. Figure 1 shows JMVX's novel approach. Instead of capturing interactions between the program and the underlying Operating System (OS), as existing MVX techniques do; JMVX instruments key non-deterministic methods, identified via an automated approach (Section 3.2). The methods reside close to the boundary between Java and native VM code. Staying close to this boundary allows objects to replicate as much internal state as possible. JMVX's approach naturally ignores all benign non-determinism due to the HLLVM operation, and only captures non-determinism directly initiated by the target application. This allows JMVX to record and replay class loading, which is an area other java based RR systems [11] struggle with. We found that instrumenting only 71 methods belonging to the Java API is enough to support MVX and RR on the DaCapo benchmark suite [9], which is representative of the Java language (Section 4).

Capturing non-deterministic methods is not enough to enable MVX and RR of multi-threaded programs, because it does not capture the order in which threads synchronize (*e.g.,* which threads gets which lock and when). Our empirical results show that ignoring multi-threading causes benchmarks to fail on MVX or on replay as the variants experience a different thread scheduling (Section 4.4). Jmvx supports multi-threading for MVX and RR with a novel approach: capture the scheduling observed in one variant/recording, and ensure that all other variants/replaying follows the same scheduling. Other MVX systems for C/C++ follow a similar approach [20, 23, 32]; but no RR system does so. RR systems that support multi-threading typically limit programs to run on a single thread [30], which results in higher recording costs and lower fidelity. RR systems that specifically target managed languages do not support multi-threading [7, 38]. Jmvx is also the first system flexible enough to support both MVX and RR without requiring a fundamental reimplementation of each, and with competitive performance.

Besides describing the design and implementation of Jmvx (Section 3), this paper also presents an extensive evaluation of Jmvx's runtime and memory costs (Section 4). We show that Jmvx imposes an average of 5% |47% performance overhead when performing MVX, depending on whether multi-threading support is disabled|enabled, which is competitive with similar MVX systems that target C/C++ [1, 20, 32]; an average 8% performance overhead when recording single threaded programs, lower than the 333% reported in a generic RR system [30] and the 14% reported in a specialist RR system for Java [7]; 25% performance overhead to record multi-threaded programs; and an average 13% |73% performance overhead to replay programs with multi-threaded support disabled|enabled. We also show that the costs are dominated by the amount of non-deterministic methods invoked, and by the amount of thread synchronization present during program execution.

Jmvx's limitations and assumptions, described in detail in Section 3.8, include: Jmvx does not support shared memory synchronization (*e.g.,* volatile read/writes), Jmvx assumes the underlying program is free of data-races, Jmvx does not support finalizers, does not capture GC behavior (*e.g.,* weak references), and does not support custom class-loading that uses multi-threading (*e.g.,* loading a class adds a task to a thread-pool). The current prototype targets Java 8 (due to the module system introduced in Java 9+, as explained in Section 3.9)does not support shutdown hooks, does not intercept `java.util.concurrent.Lock`, and does not capture environment variables or file locks.

We have released a replication package that includes the prototype implementation of Jmvx, the scripts that automate all the experiments, the dataset we obtained when evaluating Jmvx, and detailed instructions on how to use Jmvx and replicate our evaluation [40], together with a repository containing all of the source code [41].

In short, this paper makes the following contributions:

- Presents the design and implementation of Jmvx, a novel MVX system for managed languages based on intercepting a modest number of manually-identified non-deterministic methods, with a flexible architecture that can also act as an efficient and competitive RR system that supports multi-threaded programs;
- Reports on an extensive evaluation of Jmvx's performance and memory overhead when executing a benchmark suite representative of the Java language;
- Makes available Jmvx's prototype, the dataset reported in this paper, and a replication package that reproduces all experiments reported in this paper.

2 The Problem with Multi-version Execution for Managed Languages

MVX works by capturing sources of non-determinism in all variants, and ensuring that they match or are equivalent. Typical sources of non-determinism include: file manipulation, network operations, accessing time and date, and random number generation. For instance, if a variant reads 100 bytes from a network socket, MVX has to ensure that all variants will read the same 100 bytes from the same socket at the same point in the program. A common solution is to employ a *leader-follower* architecture [20, 23, 32], in which one variant actually accesses non-determinism (the leader) and sends the results to the other variants (the followers).

Programs access non-determinism through the underlying Operating System (OS), via *system calls*. Existing support for MVX focuses on programs written in C/C++ that are compiled to native binaries, where the interaction with the underlying OS is directly observable [1, 8, 15, 19, 20, 23, 24, 27, 29, 30, 32–34, 37, 39, 46, 47, 49]. Such MVX solutions intercept all system calls and ensure that they behave in the same way on all variants. Even though this is an elegant solution, it does not work with programs written in managed languages that execute inside a High Level Language Virtual Machine (HLLVM).

One problem is *self-modifying code*, which is a well known limitation of MVX. Existing HLLVMs typically use a Just-In-Time compiler (JIT), which profiles code as it is interpreted and compiles/optimizes the code where the program spends most of its time. Existing MVX solutions do not support programs that generate code and run it, instead they may use a special loader/linker to instrument all code that a variant accesses. If a variant generates code besides loading/linking it, such code escapes the control of the MVX system.

Perhaps it is possible to modify existing MVX systems to support code generation, by using memory protection to detect when executable code pages are modified. However, such an approach comes at great costs. First, modern OSes implement memory protection via signals, which are slow (require many context switches to be delivered) and coarse (only detect changes at the granularity of a whole page). Second, finding CISC code inside a page requires valid pointers to where code starts [36, 43], which in turn requires modifying the HLLVM. Third, intercepting code generation requires that all variants JIT the same code at the same time. Note that JIT is based on low latency timing sources, such as the instruction `RDTSC` [14] to approximate how many cycles elapsed in a portion of code. Instrumenting such sources increases their latency to an extent where they are not accurate anymore, which hides the real hot-spots in a program and leads the JIT to optimize different code, leading to overall slower code.

Another problem is the inherent non-determinism inside HLLVMs, to the extent that two executions of the same deterministic program results in issuing different sets of system calls. Besides the JIT issue described above, we have to consider Garbage Collection (GC), code loading, and other internal HLLVM mechanisms involved in runtime management. For instance, the Java Virtual Machine (JVM) does not guarantee any ordering on class loading. A complex class hierarchy, with classes inheriting multiple interfaces, may be resolved/loaded in many valid orders. Each execution of the same program may load the same classes in different orders (we observed this to be true in practice). Existing MVX solutions applied to programs executing inside an HLLVM would observe variants behaving differently — a *divergence* — and terminate execution. Tolerating such divergences (*i.e.,* supporting programs written in managed languages) requires a fundamental reimplementation of the MVX technique. In its current form Jmvx does not capture or enforce data races in shared memory. Some caches that use weak references to be GC'ed based on memory pressure had to be disabled. In practice, JMVX is still able to perform MVX and RR on the DaCapo benchmarks with minimal tweaks to the JVM.

3 A Novel Multi-version Execution System for Managed Languages

Instead of expanding existing Multi-version execution (MVX) systems to work around the issues explained in Section 2, we propose Jmvx— a new MVX system designed carefully to support programs written in managed languages. Jmvx's architecture (Section 3.1) intercepts only application related non-determinism, avoiding all non-determinism related with runtime management, by

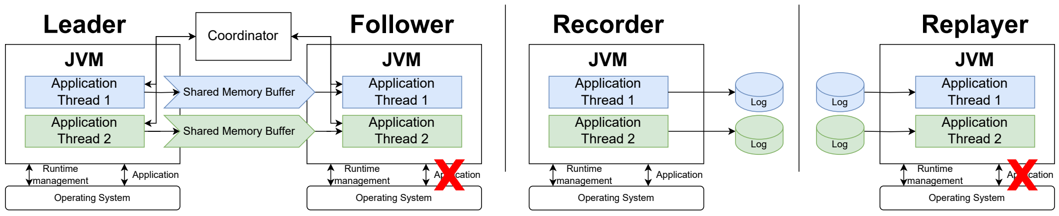

Fig. 2. Architecture of Jmvx showing the two supported models of execution (MVX and RR), and the four possible roles (leader, follower, recorder, and replayer). Leader and follower execute at the same time. The recorder executes first, generating a log that can be replayed after the recorder exits.

```
01: Passthrough.read(SocketInputStream s) { return s.$JMVX$read(); }
02:   Leader.read(SocketInputStream s) { int r = s.$JMVX$read(); follower.send(new SocketReadI(r)); return r;  } }
03: Follower.read(SocketInputStream s) { Object o = leader.recv(); assert(o instanceof SocketReadI); return o.i; }
}
04: Recorder.read(SocketInputStream s) { int r = s.$JMVX$read(); log.write(new SocketReadI(r)); return r;  } }
05: Replayer.read(SocketInputStream s) { Object o = log.read(); assert(o instanceof SocketReadI); return o.i; } }
06:
07:   Leader.read(SocketInputStream s, byte[] data) {
08:       int r = s.$JMVX$read(data); follower.send(new SocketReadIB(r,data)); return r;  } }
09: Follower.read(SocketInputStream s, byte[] data) {
10:     Object o = leader.recv(); assert(o instanceof SocketReadIB);
11:     System.arraycopy(o.b, 0, data, 0, o.i); return o.i; } }
12:
13: class SocketReadI { int i; }
14: class SocketReadIB { int i; byte[] b; }
```

Fig. 3. Java pseudo-code showing how Jmvx behaves when intercepting the non-deterministic method `SocketInputStream.read`. Jmvx can change the implementation at run time, which allows to turn a follower into the leader (promotion) and the leader into a follower (demotion). Note that method `$JMVX$read` denotes calling the original method (*i.e.,* `SocketInputStream.read`). Lines 7–11 show how Jmvx handles side-effects (*e.g.,* filling an array with read data).

instrumenting automatically identified non-deterministic methods (Section 3.2). Due to its flexible architecture, Jmvx also supports Record/Replay (RR); which records an execution to a log stored on disk, and can later replay that execution from the log with high fidelity (Section 3.3). Jmvx supports multi-threading by capturing the order of thread synchronization operations in one variant and enforcing it on another (Section 3.5) and tolerates benign divergences due to dynamic code loading (Section 3.4). Key to Jmvx's MVX performance is a circular buffer located in memory shared among all process that allows very fast communication between variants (Section 3.6).

3.1 Architecture

Jmvx supports 2 models of execution: Multi-version execution (MVX) and Record-Replay (RR), shown in Figure 2. Jmvx's MVX model follows a leader-follower architecture [1, 20, 23, 32], which uses possible roles: leader and follower. The RR model allows users to launch the application in two possible modes: recorder and replayer.

3.1.1 Multi-version Execution. Jmvx follows a leader-follower architecture, common on high-performance MVX systems [1, 20, 23, 32], which requires 3 processes: (1) the leader, (2) the follower, and (3) the coordinator. Both leader and follower run equivalent versions of the same application instrumented to support MVX (Section 3.3) as variants. Each variant can be a different version or release of the same program, which allows MVX to tolerate bugs present in the older release or new bugs introduced in the newer version [19]. JMVX's current prototype supports only one follower, but this is not a fundamental limitation of our approach.

The **leader** variant is responsible for issuing requests to the underlying OS (*e.g.,* reading from a file, writing to the network, checking the time), and sending the results to the follower. Such requests are sources of non-determinism, and JMVX intercepts methods that can access such non-determinism (Section 3.2). For instance, Figure 3 shows how JMVX intercepts method `SocketInputStream.read`. The leader, on Line 2, first issues the request, then sends it to the follower, and finally returns the results to the application. Communication with the follower is made either via the coordinator (described below) or a highly-performant shared memory buffer (Section 3.6). The **follower** variant does not issue any requests. Instead, it receives method invocation data from the leader, and returns the same non-deterministic data to the application. Following our example, on Line 3, the follower receives data from the leader, ensures that the data matches the same method (depicted via an assertion), and returns the same data to the application. If the follower attempts to call a different method from the leader, we say that a **divergence** took place (described below).

The **coordinator** allows for communication between the leader and follower without requiring the leader to know the identity of the follower (or number of followers). The coordinator allows all variants to synchronize as needed, and also allows the leader to send open file-descriptors that represent files on disk and active sockets.

3.1.2 Promotion/Demotion. The coordinator can start a promotion/demotion event that demotes the leader to become a follower and promotes a follower to become the new leader. Such an event can be triggered manually by the user, or automatically when the coordinator detects that the leader crashed. JMVX ensures that the follower has all the resources (*e.g.,* file-descriptors) needed to take over, and that the follower's state is equivalent to the leader's. More information on how JMVX handles file descriptors is in Section 3.3.3. Implementing promotions and demotions is relatively straightforward, as JMVX allows the roles of each variant to change at run time.

3.1.3 Record-Replay (RR). RR can be seen as offline MVX. Instead of sending data between running processes, the RR **recorder** saves that data as a log file on disk. Later, after the recorder has finished execution, a **replayer** process can replicate the same execution by replaying the log file. RR is a popular technique to capture bugs in production, and replicate them in development [7, 30]. Figure 3 shows how the recorder and replayer handle method `SocketInputStream.read`, which is similar to leader/follower except that the data is always written to/read from a file on disk.

3.1.4 Divergences. Both follower and replayer check if the method currently being called matches the next method called by the leader or logged by the recorder. Figure 3 shows such checks as assertions on Lines 3 and 5. When a check fails, the two executions have **diverged** and the follower/replayer terminates after printing a detailed message denoting what went wrong (containing the stack trace of where the divergence took place). The divergence may be real or benign. We experience real divergences during development before intercepting all methods that can access non-determinism. Benign divergences happen when the two variants execute the same logic via different actions. Tolerating benign divergences is common in MVX [19, 20, 32, 34], and JMVX provides mechanisms to tolerate benign divergences (Section 3.7). JMVX is the first RR system that can tolerate benign divergences using custom handlers between the recording and replay.

3.2 Methods Intercepted

We claim that it is possible to support MVX and RR on managed languages by intercepting a reasonable number of methods, which can be identified automatically. The methods that need to be intercepted are *non-deterministic* as their behavior depends on the environment in which they run.

```
07: // Original code
08: nativeMethod();
09:
10: // Instrumented code
11: JMVX.beforeNative();
12: nativeMethod();
13: JMVX.afterNative();
```

```
14: Lock lock;
15: volatile boolean syscall;
16:
17: static void beforeNative() {
18:  lock.lock();
19:  syscall = false;
20: }
21: static void afterNative() {
22:  if (syscall)
23:   logStackTrace();
24:  lock.unlock();
25: }
```

```
26: void signal(int sig) {
27:  if (sig == USR2)
28:   syscalls = true;
29: }
30:
31: void logStackTrace() {
32:  StackTraceElement[] st;
33:  Exception e = new Exception();
34:  st = e.getStackTrace();
35:  log(st);
36: }
```

Fig. 4. Mechanism to detect native methods during a sample run of a target program. This process runs with `strace`, set to deliver signal USR2 at every system call.

For instance, reading/writing to/from a file on disk or a network socket are examples of such non-determinism. Such non-deterministic methods involve a *system call* to the underlying OS to access the environment. In Java, methods that issue system calls do so using a C/C++ implementation, and are thus declared as `native`. For instance, Java programs can read from disk through the public method `FileInputStream.read`, which is implemented in Java and calls the private method `FileInputStream.read0`, which in turn is declared as native. However, not all native methods issue system calls. For instance, the native method `System.arrayCopy` is just a wrapper to native code that moves memory efficiently, used widely by all Java programs when performing string processing.

To find which native methods result in non-determinism via system calls, we employ a dynamic analysis that combines instrumenting native methods and running the instrumented program with the utility `strace` [26] configured to trace the Java process and deliver it a signal (USR2) on every system call. Figure 4 shows the native method instrumentation on Lines 7–13, which surrounds native calls with the methods `beforeNative` and `afterNative`. The method `beforeNative` acquires a lock on Line 18 to ensure that there is only one native method being executing in the whole process at any given time, and then resets a flag on Line 19. Then, the Java program executes the original native call. If the native method results in a system call, then `strace` delivers signal USR2, which results in executing the signal handler in Lines 26–29, which sets the flag on Line 28. Then, the method `afterNative` logs the current stack trace if the flag was set (Line 23), indicating that the native method resulted in a system call. Lines 31–36 show a straightforward way of capturing the current stack trace, and logging it for later analysis. Native methods that execute without setting the flag did not involve any system call and do not need to be instrumented.

After obtaining stack traces as described above, we process them offline through an automated process. The first step is to cluster the traces by only looking at the top N frames (excluding our instrumentation). We found that N=8 works well in practice. Then, we inspect each cluster manually to decide what method to intercept. Typically, the top frame is a Java wrapper that just calls the private method; in which case we can intercept the Java wrapper. In our manual analysis, we attempt to remain as system agnostic as possible. For instance, to copy files efficiently, the method on top of the stack is `UnixFileSystemProvider.copy`, which then calls native private method `UnixFileSystemProvider.copy0`. However, the Java method is only ever called from method `java.nio.Files.copy`, which we intercept. We expect different operating systems have different file system providers, but those are still called from the method we intercept. The exceptions to this rule are `FileSystemProvider.checkAccess` and `UnixFileAttributes.get`. These methods are called from many other methods (*e.g.*, `Files.isReadable`, `Files.isWritable`, etc). In this

```
37: // Original
38: class SocketInputStream extends InputStream {
39:  int read() { /* read a byte */ } }
```

```
40: // Instrumented
41: class SocketInputStream extends InputStream {
42:  int $JMVX$read() { /* read a byte */ }
43:  int read() { return JMVX.r.get().read(this); } }
44:
45: class JMVX { static ThreadLocal<Role> r; }
```

Fig. 5. Example of how Jmvx intercepts method `SocketInputStream.read`. Figure 3 shows possible implementations of `Role` with how Jmvx handles the intercepted method (*i.e.,* what code gets called on Line 43).

case, by intercepting the lower level methods, we drastically reduce the total number of methods intercepted.

Table 1 shows the methods we identified when running all benchmarks in the DaCapo suite [9] versions Bach and Chopin. The methods listed are an extensive set of methods required, but not exhaustive. This is not a fundamental limitation of Jmvx, as it can support more non-deterministic methods as needed by different workloads, and we provide an automated way to find which those methods are. Jmvx in conjunction with *strace* can be used to trace a target application and identify wrappers to java methods that lead to system calls. If the methods returned differ from those in included in Table 1, then additional support (for new methods) is needed to run the program. A more comprehensive approach would be to trace the JVM's test suite to identify all ways in which Java can access the underlying system; albeit the program should still be traced to find calls through a custom native interface.

Besides non-deterministic methods, we also instrument code to support class-loading and multi-threading; which we discuss in detail in Sections 3.4 and 3.5, respectively.

Blocking System Calls. Our technique uses a global lock to ensure that the program under analysis only calls one native method at a time. Without this lock, our results would contain false-positives when two different threads call different native methods and one results in a system call. In this case, the stack of the second thread would also be captured. With the lock as described, our technique does not lead to any false-positives.

However, our technique results in deadlocks for blocking system calls. For instance, waiting for a client to connect via `accept` or `epoll` is blocking. A server program may use such a blocking system call in one thread, while another thread performs unrelated tasks (*e.g.,* receiving/sending data to a client already connected). As the thread performing `accept` holds the global lock, the thread handling the client cannot make progress; which in practice results in a deadlock.

There are three possible solutions to such deadlocks. First, we can simply disable the lock (Lines 18 and 24 in Figure 4) and deal with the false-positives when performing the manual analysis of the traces. Second, we can list which native methods result in blocking system calls, skip instrumenting those methods, and rerun the trace collection. In practice, we only observed one benchmark to hang (h2-server). We disabled the lock and filtered the results manually by only using methods found in many executions of the benchmark. Third, we could force the blocking thread to give up the lock and suspend temporarily and then give it the lock back when it is rescheduled.

3.3 Instrumentation

Jmvx instruments relevant methods that interact with the underlying system to capture non-determinism, as shown in Figure 1. Figure 5 shows an example of instrumenting code `SocketInputStream.read`, which reads one byte from a network socket. First, Jmvx renames the original method (Line 39) by adding the prefix `$JMVX$` to it (Line 42). Then, Jmvx adds a method with the original name that delegates to the current `Role` (Line 43).

Table 1. Methods that Jmvx identifies and intercepts. N denotes how many benchmarks use the method, and M denotes that the method was manually identified even though it does not issue system calls (*e.g.*, to support class-loading and multi-threading). Jmvx intercepts 71 methods distributed among 28 classes in 10 packages. Methods marked with ^{opt} represent roughly 99% of all calls, and are optimized manually.

Package	Class	Name	N
java.util.zip	ZipFile	getEntryCSize	12
		getEntry	12
		getEntrySize	12
		getEntryBytes	12
		freeEntry	12
		getEntryMethod	12
		open	12
		getEntryFlag	9
		getEntryCrc	8
		getEntryTime	7
		close	3
		getTotal	2
		startsWithLOC	2
		getNextEntry	1
	ZipInputStream	read^{opt}	10
	Adler32	updateBytes	1
	CRC32	updateBytes	4
	ZipOutputStream	write^{opt}	1
java.util.jar	JarFile	getMetaInfEntry	5
sun.nio.ch	EPollArrayWrapper	epollCtl	1
		epollCreate	1
		sizeofEPollEvent	1
	FileDispatcherImpl	read	8
		write	1
		seek	3
		size	3
		close	8
java.util	TimeZone	getSystemTimeZone	4
java.lang	System	currentTimeMillis^{opt}	12
		nanoTime^{opt}	12
	ClassLoader	loadClass	M
	Object	wait^{opt}	M
	Thread	run	M
java.util.concurrent	ThreadPoolExecutor	getTask	M
	QueueingFuture	done	M
	ConcurrentLinkedQueue	poll	M
synchronized^{opt}			M
java.io	RandomAccessFile	open	4
		close	2
		read^{opt}	4
		write^{opt}	2
		seek^{opt}	4
	FileInputStream	open	12
		available^{opt}	12
		read^{opt}	12
		close	12
	FileOutputStream	open	12
		write^{opt}	12
		close	8
	File	getCanonicalPath	12
		delete	8
java.nio	Files	createTempFile	1
sun.nio.fs.spi	FileSystemProvider	checkAccess	2
		copy	1
sun.nio.fs	UnixFileAttributes	get	1
	UnixNativeDispatcher	opendir	8
		fdopendir	1
		closedir	8
		readdir	2
		mkdir	2
		open	8
		dup	1
		stat	3
		lstat	8
		access	3
		realpath	3
java.net	ServerSocket	bind	1
		accept	1
	Socket	connect	1
	SocketInputStream	read^{opt}	1
	SocketOutputStream	write^{opt}	1

There are 5 possible roles: pass-through, leader, follower, recorder, and replayer. Figure 3 shows how each handles method `SocketInputStream.read`. *Pass-through* (Line 1) simply calls the original method. *Leader* (Line 2) calls the original method, sends information about the method called and the data received to the follower, and returns the read data. *Follower* (Line 3) does not call the original method, as doing so results in an incorrect behavior that reads from the same socket two times. Instead, the follower waits for the leader to send a method call, checks that execution has not diverged, and returns the data that the leader sent. *Recorder* (Line 4) behaves as Leader except that it saves the data in a log file, and *Replayer* (Line 5) behaves as Follower excepts that it reads data from a log file.

Some of the native methods identified are public and called directly from client code. There is no method wrapper that Jmvx can instrument, so Jmvx intercepts such native methods at the call site, redirecting them to a static method. For instance, Jmvx redirect calls to native method `System.nanoTime` to call instead `JMVXRuntime.nanoTime`, which in turn delegates to the current role (similarly to Line 43).

3.3.1 Serialization, and Primitive Data Types. For each method, we created a respective serializable class, similar to `SocketReadI` shown in Line 6 of Figure 3. By default, the leader instantiates the respective class, sets the relevant data, and sends that instance to the follower using Java serialization (Line 2). Then, the follower receives a deserialized object, checks that it is of the expected type, and finally accesses the relevant data (Line 3). Note that all the methods that Jmvx instruments take/return serializable data types (primitives and strings), so Jmvx can simply use Java serialization.

3.3.2 Optimizations. Java serialization is well known to be slow, which is not a problem for most of the intercepted methods as applications call them a small number of times. However, we determined empirically that a few methods are called very frequently and require special support. We optimized a small set of methods that represents roughly 99% of all calls in the DaCapo benchmark suite:

- Methods that read data (*e.g.,* `InputStream.read`) send a byte array of data read
- Methods that write data (*e.g.,* `OutputStream.write`) compute a fast checksum on the leader, and send that checksum for the follower to ensure that it is writing the same data
- Methods that access timing data (*e.g.,* `System.currentTimeMillis`) send a `long` that represents the time returned to the leader

3.3.3 File Descriptors. Some of the methods instrumented deal with file descriptors, either representing files on disk (*e.g.,* `FileInputStream.open`) or active network connections (*e.g.,* `Socket.connect`). One of the main goals of MVX is to survive failures (*e.g.,* the leader fails, so the follower becomes the new leader), so Jmvx needs to ensure that the follower has access to the same file descriptors as the leader.

For files on disk, this is relatively easy: the follower can open the same file as the leader. Instead, Jmvx relies on Unix domain sockets [22] to send open file descriptors from the leader to the follower as ancillary data. As the leader does not know the identify of the follower, it sends open network connections to the coordinator, which then forwards it to the follower.

Finally, as the follower does not actually read from files on disk, Jmvx ensures all matching `read` operations move the file offset by the same amount via a `seek` of the same length as the read. Later, if the follower is promoted to leader, all file descriptors' offsets match the leader.

3.4 Code Loading

The Java Virtual Machine loads code using class-loaders. Typically, loading a class involves looking for a `.class` file following a hierarchy of folders and jar files (the *classpath*), by listing contents of folders, and opening/reading relevant files. For instance, a Java program may first search the local classpath in the order it was provided for the application, stopping when it finds the first `.class` file with the expected name and package. If it fails, it may then revert to the bootstrap classpath, which checks for `.class` files internally in the JVM itself (*e.g.,* all classes belonging to packages starting `java`).

The process of loading classes as described above naturally results in calling methods that Jmvx intercepts (*e.g.,* reading bytecode from a `.class` file, listing a folder, opening a `jar` file). Intercepting class-loading has two important consequences. **First**, it results in (benign) divergences because the order in which the JVM loads the same set of classes is undefined. For instance, if a class C implements two interfaces I_1 and I_2, the JVM may load the interfaces in any order. If both interfaces are in different folders or jar files, different variants attempting to them in a different order experience divergences: One variant loading I_1 finds it in the first jar file it checks, another variant loading I_2 needs to open a second jar file; resulting in a divergence. This issue is pressing enough that existing solutions for MVX/RR in Java ignore class-loading [7, 21] and assume all variants have access to the same code sources; or require all code to be loaded before starting [11]. Furthermore, attempting to use a whole-process MVX/RR solution for Java also results in the same divergences between different, otherwise equivalent, runs. **Second**, intercepting class-loading prevents MVX from loading different code in different variants, which is a key feature of MVX [20, 32, 34].

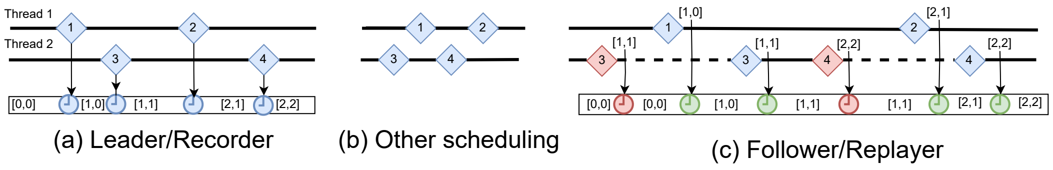

Fig. 6. Lamport clock used to synchronize threads.

JMVX supports class-loading by intercepting method `ClassLoader.loadClass`. When executing MVX, both leader and follower simply revert back to the `pass-through` role (Line 1 in Figure 3) while a class is loading, which does not capture methods invoked by each variant. Ignoring class-loading trivially tolerates benign divergences and allows different variants to load different, but equivalent, code; which is a key feature of MVX that allows leader and follower to run different versions [20, 32, 33, 38].

When performing RR, however, JMVX ensures the replay loads the same code as the recording, even if the code is not available in the system replaying the log. JMVX does so by associating all methods intercepted while loading a class with the class being loaded (and classloader used). For instance, continuing our example, when loading interface I_1, JMVX captures the intercepted methods that find the respective `.class` file in the first jar file, and associates that log with I_1. Then, JMVX does the same for I_2, but this time it captures not finding the `.class` file on the first jar and then opening a second jar. When replaying, if the JVM now loads I_2 first, then JMVX can match the actions of the replayer with the respective log of intercepted methods (that open two jar files). Our technique also supports actions performed from class initializers (initializers of `static` fields or code in `static` blocks), which we observed to open and process other files.

3.5 Multi-threading

One of the driving goals behind JMVX is to support multi-threading with high fidelity and minimal overhead for the leader/recorder process. To achieve that goal, JMVX starts by using separate communication channels per thread. As shown in Figure 2, each pair of leader/follower and recorder/replayer threads use a separate communication channel: a dedicated coordinator thread and shared memory buffer/log file per thread for MVX/RR.

Keeping separate communication channels ensures threads make progress in parallel, but it is not sufficient to capture all non-determinism observed by the leader and recorder. Consider the example shown in Figure 6. In one execution (a), the leader/recorder observes events by Threads 1 and 2 in the order: 1, 3, 2, 4. However, in another execution (b), it is possible to observe the same events in a different order: 3, 1, 4, 2. The two different execution orders may result in different program behavior. For instance, consider that the two threads are attempting to commit two parallel database transactions under a global lock, (a) may result in Thread 1 succeeding and (b) in Thread 2 succeeding, leading to different database contents.

3.5.1 Instrumentation. JMVX intercepts uses of the `synchronized` keyword, and calls to `Object.wait`, as shown in Figure 7. We note that JMVX's implementation can be extended to intercept other synchronization methods, such as implementations of interface `java.util.concurrent.Lock`'s `lock`, `unlock`, and `await` methods.

Every object in Java has an implicit monitor [18], which can be used to synchronize threads via the `synchronized` keyword. A method marked as `synchronized` (Lines 46–50 of Figure 7) acquires the respective monitor at the start and releases at the end, even when throwing exceptions. Instance

```
46: public synchronized void m() {
47:
48:     /* original body*/
49:
50: }
51:
52:
53:
54: synchronized (object) {
55:     /* original block */
56: }
57:
58: o.wait();
```

```
59: public void m() {
60:     JMVX.monitorEnter(this);
61:     try { $JMVX$m(); }
62:     finally { JMVX.monitorExit(this); }
63: }
64:
65: private void $JMVX$m() { /* original
body */ }
66:
67: JMVX.monitorEnter(object);
68: try { /* original block */ }
69: finally { JMVX.monitorExit(object); }
70:
71: JMVX.monitorWait(o);
```

Fig. 7. Synchronization operations instrumented by JMVX: synchronized methods (Lines 16–22 and 29–35), synchronized blocks (Lines 24–26 and 37–39), and invocations of `Object.wait` (Lines 28 and 41).

methods (*i.e.,* non-static) acquire the monitor of the receiver object (*i.e.,* `this`), class methods (*i.e.,* static) acquire the monitor of the class object. JMVX instruments `synchronized` methods as shown in Lines 59–65. First, it drops the `synchronized` modifier and renames the method, in our example from `m` to `$JMVX$m` (Line 65). Then, it generates a new method with the same name (Line 59) that surrounds a call the renamed method with a `try-finally` block that calls JMVX's runtime to acquire/release the relevant monitor (Lines 60–62).

Synchronized blocks (Lines 54–56) are easier to instrument, as the Java compiler already produces an equivalent `try-finally` structure using bytecodes `monitorenter` and `monitorexit` to acquire and release the monitor, respectively. JMVX simply redirects those bytecodes to the equivalent methods in JMVX's runtime (Lines 67–69).

Internally, JMVX uses `sun.misc.Unsafe` to acquire/release monitors at will. The leader/recorder **first** acquires the monitor, and **second** increments and copies a global Lamport clock, explained in Section 3.5.2. Then, the leader sends the copy to the follower, and the recorder writes the copy to the log. The follower/replayer **first** matches the clock copy with the global clock, which may result in waiting. Once matched, the follower/replayer **second** acquires the monitor and then increment the global clock. Doing so ensures the same monitors are acquired in the same order by all variants.

Finally, `wait` (Line 58) allows threads to wait until another thread calls `notify` on the same object (or a timeout occurs). Method `wait` must be called while holding the respective monitor, and ensures the thread is holding that monitor when `wait` returns. JMVX simply redirects calls to `wait` (Line 71). JMVX implements `wait` for the leader/recorder in the same way as acquiring a lock, by **first** calling `wait` and **second** incrementing the global clock and obtaining a copy, which it sends to the follower or writes to the log. The follower/replayer never calls `wait`. Instead, it starts by releasing the monitor (which is done implicitly when calling `wait`). Then, the follower/replayer **first** matches the received clock against the global clock, which may result in waiting; and **second** acquires the monitor and increments the global clock.

3.5.2 Logical Clock. JMVX captures the ordering of events observed by the leader/recorder process using a global Lamport logical clock [25], as used in other MVX systems [20, 32, 46]. JMVX's logical clock is a vector of numbers that starts at zero, with one entry per thread. In Figure 6, the first entry refers to Thread 1, and the second to Thread 2. The initial vector is thus $[0, 0]$. When a thread performs a captured event, it atomically increments its position on the vector and obtains a copy. Event 1 thus obtains vector $[1, 0]$, 3 vector $[1, 1]$, 2 vector $[2, 1]$, and 4 vector $[2, 2]$.

To ensure re-execution of thread ordering events in the same order as originally captured, Jmvx saves such events in their respective clock. The leader process sends the clock to the follower, and the recorder saves the clock in the log. Later, the follower/replayer aligns the clock of each event with its own global clock, as shown in Figure 6 (c). In this re-execution, the follower/replayer's Thread 2 reached Event 3 first. Jmvx attempts to match the event's clock $[1, 1]$ with the current global clock $[0, 0]$. Two vectors match if they differ by exactly 1 for the thread doing the match, which means *"this thread is next"*. Such matching fails for Event 3, so Jmvx makes Thread 2 wait.

Later in our example, Thread 1 reaches Event 1 and attempts to match the event's clock $[1, 0]$ with the global clock $[0, 0]$. The clocks match, so Jmvx updates the global clock to $[1, 0]$ and allows Thread 1 to proceed. Shortly after, Jmvx is able to match Event 3's clock, updating the global clock to $[1, 1]$ and allowing Thread 2 to proceed. The same process repeats for Events 3 and 4.

3.5.3 Performance and Back-off Optimization. We note that Jmvx's approach is biased towards low performance overhead on the leader/recorder. A wrong scheduling in the follower/replayer causes that thread to wait, which is costly. Jmvx's current implementation uses a global lock for the leader/recorder to increment/copy the clock atomically. Given that the follower/replayer does need to increment and copy the vector in one atomic step, Jmvx uses volatile memory operations instead, as defined by the Java Memory Model [28]. Each follower/replayer thread uses busy-waiting on volatile reads to wait for its turn, and a volatile write to increment the global clock on the thread's position. Note that each thread can update only their respective position in the global clock (*e.g.,* Thread 1 can only update the first number in the clock).

We measured that naive busy-waiting resulted in at least a 2 orders of magnitude slowdown in the follower/replayer. Note that a waiting thread slows the whole system down via cache-coherence traffic, which Jmvx minimizes via a back-off array with adequate padding to avoid false-sharing. To illustrate the back-off optimization, consider the following example. Thread 1 receives an event with clock $[1001, 2000, 3000]$, and the current clock is $[1000, 0, 0]$. First, each waiting thread computes how far it is from each other thread, and picks the most distant thread. In this case, Thread 1 knows it is 3000 increments away from Thread 3. If the distance is short, the thread simply busy-waits on the global clock itself. We found that 1000 works well in practice.

When a waiting thread is far from the next event, it registers the event in the appropriate position of the back-off vector. In our example, Thread 1 registers 3000 in position 3 of the back-off vector, which means *"please notify me when Thread 3 reaches Event 3000"*. Each thread sets its position in the back-off vector to zero when it increments the global clock to the specified value. In our example, eventually Thread 3 increments its position in the global clock to 3000 and, at that point, sets position 3 in the back-off vector to zero. Thread 1 now notices the change, and checks the global clock again.

Note that busy-waiting on the back-off vector minimizes cache-coherence traffic. When Thread 1 is waiting for 3000 on position 3, only Threads 1 and 3 have that position in their cache. Thread 1 is comparing the contents against 0, Thread 3 is comparing the contents against the most recent clock increment; both are reading operations so both threads can have the line in their cache without conflict. There is cache-coherence traffic only when Thread 3 reaches the requested number (3000) and updates the back-off vector position to 0, which in turn causes Thread 1 to read fresh memory and break out of the back-off loop.

3.5.4 Thread Pools. Besides uses of monitors, our current prototype also handles a more advanced concurrent feature in `java.util.concurrent`: thread pools via the `ExecutorService` API, which

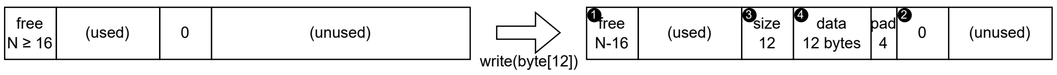

Fig. 8. Structure of the circular buffer, and example of writing some data.

out empirical evaluation found extensive use of. Users submit tasks, that are executed in a non-deterministic order when the executor has free threads. To support accurate MVX and RR, Jmvx needs to ensure that all executors have the same number of threads, and that the same threads execute the same tasks in the same order in all variants. As such, Jmvx instruments relevant methods by capturing their order in the leader/recorder via Lamport clocks (described in Section 3.5.2) and enforcing the same order in the follower/replayer. The current implementation captures the order by treating the following methods as synchronized: `ThreadPoolExecutor.getTask`, and `ConcurrentLinkedQueue.poll`. Jmvx also needs to ensure tasks finish in the same order, so it also captures the order of execution of method `QueueingFuture.done`.

3.6 Circular Buffer

To retain good performance when performing MVX, Jmvx requires an efficient means of communication so that the leader can send data to the follower. Other MVX [1, 20, 32, 46] systems use a C implementation of the Disruptor pattern [44] in shared memory to allow for efficient communication between variants. However, we could not find a suitable implementation in shared memory written in Java. As such, we implemented a simpler circular buffer located in shared memory as an *off-heap byte buffer* manipulated via `sun.misc.Unsafe`, which allows unrestricted access to memory inside the JVM. Jmvx allocates the buffer in `/dev/shm` and then uses `Unsafe.mmap` to map the file to memory on both leader and follower. Our implementation is a general-purpose means of efficient communication between two Java processes that can be easily reused for other purposes.

The circular buffer is accessed by two processes, one as the *writer* (leader) and another as the *reader* (follower). Figure 8 shows the structure of the buffer. The first 8 bytes keep how many bytes in the buffer are `free` and can be written. Each process keeps their own pointer to where they are in the buffer, both pointers start right after `free` and move to the start of each entry after reading/writing. Each entry starts with 8 bytes that contain its `size`, followed by the `data` and `padding` to ensure 8-byte alignment, necessary to guarantee atomic reads/writes in most architectures [14, 42].

Operations over the circular buffer always keep the following two invariants: **(1)** There is always an entry with zero `size` before the unused portion of the buffer, and **(2)** header `free` contains a lower bound on the number of available bytes (there may be more available bytes, but it is not possible that there are less available bytes).

Writing. The two invariants trivially hold for an empty buffer: The `size` of the first entry is zero, and `free` contains the capacity of the buffer. First, the writer checks whether there are are enough available bytes for a new entry comprising `size`, the payload `data`, and any needed `padding` ❶. In our example, `size` is 8 bytes, `data` is 12, and `padding` is 4; totaling 16 bytes. The writer busy-waits until there are enough bytes, and then decrements `free`. Then, the writer sets the `size` of the next entry to zero ❷, copies the `data` ❸, and, finally sets the `size` of the current entry ❹. Finally, the writer updates its pointer to the `size` of the next entry (currently zero).

Reading. The reader waits until the `size` of the next entry is non-zero. Once the reader knows the size, it reads the respective `data`, applies any required `padding`, and moves its pointer to the start of the next entry. Finally, the reader updates `free` with the available bytes (16 in our example).

```
HandlerStatus<Integer> handleDivergence(List<StackTraceElement> trace, Metadata data)           {
  if (trace.size() < 18 || !(data.getLeader() instanceof WriteB))  return new HandlerStatus(FAIL);
  StackTraceElement frame = stack.get(17);
  if (!frame.getClassName().equals("org.dacapo.harness.Callback")) return new HandlerStatus(FAIL);
  if (!frame.getMethodName().equals("complete"))                    return new HandlerStatus(FAIL);

  data.getFollower().bytes = new String(data.getFollower().bytes) + " and divergence handled ";
  return new HandlerStatus(OK, data.getFollower().length);                                       }
```

Fig. 9. Sample of a custom divergence handler for Jmvx. This handler tolerates the follower/replayer reporting a different time than the leader/recorder upon benchmark completion, concatenating string *"and divergence handled"* to the timing report.

Wrapping around. There are two possible cases for wrapping around the end of the buffer. **One**, there is enough room at the end to hold `size` but not `data`. In this case, `size` remains at the end of the buffer, and `data` can be found at the start of the buffer (after `free`). **Two**, there is no room for `size`; so the whole entry moves to the start of the buffer. Note that both reader and writer can distinguish each case as the `size` is always 8 bytes.

Correctness. When writing, step ❷ ensures Invariant 1 holds after step ❹. When reading, updated `free` at the end ensures Invariant 2 holds as the bytes of the just read entry are now unused. Considering **multithreading correctness**, our algorithm is correct in the Java Memory Model [28]. Reading and updating `free` uses atomic operations: volatile read and compare-and-swap (CAS). Failing a CAS means that the other role just updated `free`, which causes the whole check to be retried. Writing the `size` of the current entry requires a volatile operation to ensure that all other writes are visible to the reader when `size` becomes non-zero. Reading the `size` of the next entry thus also requires a volatile operation.

3.7 Handling Benign Divergences

Jmvx terminates execution when it detects a divergence, for instance one variant calls `File.exists` while other calls `OutputStream.write`. Divergences mean that the two variants have diverged and are attempting to perform different actions. We found the example above common during development as the follower would throw exceptions due to unsupported methods. The leader continues executing, while the follower attempts to print the exception to the standard out. Terminating the follower on a divergence is a reasonable default, used by other MVX systems [1, 20, 23, 32, 46]. In this section, we explain how Jmvx handles different benign divergences, guided by our efforts to support the DaCapo benchmark suite.

We note that some benign divergences are tolerable. For instance, the identity hash code of the same object may differ between variants. As long as the different hash code is not exposed by the program behavior, Jmvx simply allows it. The exceptions are when the hash code is used to determine an iteration order over visible methods (*e.g.*, opening files), explained in Section 3.7.4; and when the hash code itself is used as part of an outputted value, explained in Section 3.7.5. Other benign divergences naturally tolerated involve different native pointers obtained by the JVM (*e.g.*, when opening a zip file), each variant will use their own native pointer that refers to their own memory space.

3.7.1 Divergence Handlers. A key feature of MVX is the ability to tolerate *benign divergences* between different variants [1, 19, 20, 32, 34]. Such divergences happen when two variants display equivalent behavior via different methods. Section 3.4 explained how to tolerate loading the same classes in a different order. However, benign divergences can happen at any time during execution.

Jmvx provides a novel way of dealing with benign divergences by allowing developers to write Java code to handle divergences. Figure 7 shows an example of a custom divergence handler, that allows DaCapo's follower to report a different completion time than the leader. Custom divergence handlers need to implement method `handleDivergence` (Line 42) which takes as input the stack **trace** of where the divergence happened, and **data** about the divergence which contains what the leader and follower attempted to do; and returns a **status** about what to do next. In this case, the handler checks that the divergence happened when both variants were attempting to write bytes with a call stack size of at least 18 (Line 43), and coming from method `org.dacapo.harness.Callback.complete` (Lines 45–46). The divergence happened because the leader and follower are trying to write a different sequence of bytes (*i.e.,* report different timing results for the benchmark). As a result, the handler keeps the follower's data and appends the string "and divergence handled" to indicate that the handler did in fact execute (Line 48). Finally, the handler returns a status indicating that the divergence was tolerated, and Jmvx should continue executing normally, returning the number of bytes written (Line 49).

We used the divergence handler described in this section to allow DaCapo to print different completion times for each variant, so that we can obtain separate timings for leader and follower; and for recorder and replayer.

3.7.2 *Lazy Initialization of Data.*

It is common for programs to initialize costly data structures lazily. Multi-threaded programs typically use a double-check locking pattern [2], in which each thread checks a (volatile) variable: a `null` value means that the data needs to be initialized, any other value is the initialized data. To initialize the data, the pattern now acquires a monitor to initialize the data under mutual exclusion. Which thread initializes the data is thus non-deterministic, as threads race for a (volatile) memory access, not instrumented by Jmvx. As a result, the double-check locking pattern results in benign divergences in which thread acquires which monitor.

In practice, we found the stack trace that Jmvx prints upon divergence to be enough to find such cases of lazy initialization of data. We fix them by adding the classes with such pattern to a deny-list, that Jmvx uses to skip instrumenting classes that contain benign divergences that do not need to be synchronized among variants. Our implementation denies whole classes, instead of individual methods, for ease of implementation; in practice, we found that each denied class in our evaluation would require many denied methods. We found double-check locking inside class `Geometry` in the benchmark sunflow, which then calls synchronized methods on classes `Timer` and `UI`; and inside class `OutputPropertiesFactory` in the benchmark xalan. We added all to Jmvx's deny list.

3.7.3 *Caches.*

Caching is an important feature of modern programs, which save data that is costly to create in a data-structure (typically a hash-map or hash-set) to be reused later. Such data-structures typically can be read without acquiring monitors (on cache hits), but can be modified only under mutual-exclusion (on cache misses). Java supports *weak references* to implement such caches, which allow the Garbage Collector (GC) to reclaim data inside caches when memory pressure increases. Of course, weak references are non-deterministic and Jmvx does not capture any GC behavior. As a result, which threads acquire which monitors due to cache misses leads to benign divergences.

As with lazy data initialization above, we found the stack trace that Jmvx prints upon divergence to be enough to find such non-deterministic caches; and we fix them by adding the non-deterministic classes to Jmvx's deny list. We added the following classes: `PropertyCache`, part of fop; `PyType` and `PyJavaType`, part of jython; `ObjectPool` and `IteratorPool`, part of xalan. Two classes, `JarFile` and `JarURL` belonging to the Java API, call other non-deterministic methods (*e.g.,* `File.open`), so adding them to the deny list still results in benign divergences. Instead, we disabled their caches (turning each lookup into a miss), which trades tolerating divergences by lower performance.

3.7.4 Unordered Sets. Java provides easy access to sets with an undefined order of iteration (*e.g.,* `java.util.HashSet`). Applications use these sets to collect resources on which to perform non-deterministic operations (*e.g.,* which files to open), which then causes divergences when different variants iterate over the same set in different orders. To avoid such divergences, Jmvx's instrumentation replaces sets of objects that define a natural order (*i.e.,* implement interface `Comparable`) with the ordered alternative `SortedSet`.

3.7.5 Identity Hash Codes. Jmvx allows identity hash code to differ between variants, which may cause divergences if the application passes such values to any intercepted method as different variants will have different identity hash codes for the same object. We observed such a divergence when running the `h2` server with a verbose flag (`--trace`) that outputs the hash code of each thread object as a prefix to each log line. We wrote a straightforward custom divergence handler (Section 3.7.1) to tolerate such a divergence.

3.8 Limitations

In this section, we discuss the main limitations of Jmvx.

3.8.1 Synchronization via Shared Memory. Jmvx instruments the leader/recorder to capture relevant thread synchronization events, and then uses a logical clock to ensure the same order in the follower/replayer. Therefore, Jmvx is limited by the events it captures: monitor and thread pool usage. Furthermore, Jmvx assumes that the programs are data-race free. Jmvx ensures the same threads obtain the same monitors and execute the same tasks in the same order. However, threads may communicate and synchronize via atomic operations on shared memory. Jmvx does not capture such events, and thus does not support programs based on inter-thread synchronization via shared-memory. Nevertheless, Jmvx supports sophisticated multi-threaded applications present in the DaCapo benchmark. We believe Jmvx can be combined with existing techniques that capture shared-memory manipulation [11, 21], and we leave this as exciting future work.

3.8.2 Finalizers. The Garbage Collector (GC) may be a source of benign divergences via finalizers that execute at non-deterministic times depending on when the GC finds that object to be collectable. In our empirical evaluation, we found common uses of finalizers in two ways that result in benign divergences: (1) closing an open file descriptor, and (2) acquiring a monitor. Case 1 results in the follower/replayer closing an invalid file descriptor, which fails silently. For Case 2, Jmvx executes instrumented methods invoked by GC-related threads `ReferenceHandler` and `FinalizerThread` with the Passthrough strategy (shown in Figure 3). As such, both cases do not change the behavior of the program.

3.8.3 Multi-threading and Custom Class-Loading. Loading classes typically involve opening a small number of `jar` files to look for a particular class, until the class is found. However, as we describe in Section 3.4, the order in which the JVM loads classes is non-deterministic; which leads to divergences as different variants attempt to load different classes. During MVX, Jmvx uses the passthrough strategy (Figure 3) to handle methods invoked while loading a class; which is essential to allow different variants to load different code. While recording, Jmvx captures the methods invoked in the context of loading a class, and then sets those methods when the replay attempts to load the same class.

Java allows to use custom class-loading logic via user-defined class loaders. Jmvx supports such custom class loaders as long as they do not start threads, or communicate with other threads. This limitation prevents Jmvx from supporting the eclipse benchmark, as it uses OSGi to load classes as *"bundles"* and manage the lifetime of loaded bundles. The OSGi class-loader starts threads and then offloads dependency injection to other threads: a thread pool started by the main application which also processes tasks unrelated to class-loading. Not all classes require the same amount of processing, and bundles are loaded in different orders between different executions; which leads to divergences that cause JMVX to terminate.

3.8.4 Implementation Issues. Currently, JMVX does not support *shutdown hooks*, which are methods registered with the JVM to run when the JVM exits; JMVX can be extended to instrument such hooks and capture their order of execution. JMVX does not ensure that all variants observe the same environment variables, which can be supported by instrumenting methods `System.getEnv`. JMVX does not intercept `Lock` objects, we believe JMVX can be extended to intercept them with the same logic used for `synchronized` blocks and `Object.wait`. JMVX's current implementation may result in a deadlock if the follower acquires a file lock before the leader; JMVX can be extended to intercept method `java.nio.channels.FileLock.lock` to prevent such deadlock. As explained, the limitations listed can be fixed by extending JMVX's prototype; therefore they are not fundamental.

3.9 Applying JMVX to Other Languages

The techniques presented in this document can be applied to any managed languages with the following requirements. To identify which functions/methods to intercept, it must be possible to: (1) identify which functions/methods may result in system-calls (typically because they are implemented natively in C/C++ code or as part of runtime itself); (2) install custom signal handlers; and (3) obtain a stack-trace at any point in the program's execution. JMVX achieves 1 by considering Java methods annotated with the `native` keyword, 2 via JVM support, and 3 via instantiating an exception to get the stack-trace.

With a suitable set of functions/methods identified for a particular language/runtime, it also needs to support the ability to send/receive open file descriptors between variants for MVX; which JMVX supports via UNIX domain sockets. To port the instrumentation to another language's bytecode (or even source-code), the runtime needs to support overriding system code (*e.g.*, the `java` package in the JVM). JMVX achieves that by rewriting file `rt.jar`, found inside the JVM, which contains all base classes. Such a task may require a custom/rebuilt runtime if the base code cannot be overridden from outside the runtime itself.

JMVX's current prototype implementation targets Java 8 because later releases introduce the module system, which removes file `rt.jar`. The basic classes are now part of the JVM and kept inside a proprietary file format. We opted to prototype JMVX on JDK 8 for simplicity of development. However, we expect JMVX to be applicable to JDK 9+ by using the same techniques described in this paper. Perhaps JMVX will intercept slightly different methods due to differences in internal classes and native method distribution inside different versions of the JVM.

4 Evaluation

We implemented a prototype for JMVX in Java that totals 13524 lines of Java code, and uses the ASM bytecode library [12] to intercept the methods described in Section 3.2. As JMVX targets bytecode, it does not require access to source code of Java applications.

We evaluate the performance of JMVX's prototype, and test its correctness, using the DaCapo benchmark suite versions 9.12 (bach) and 23.11 (chopin). All experiments were performed on a machine running Ubuntu 22.04.3 LTS and equipped with 4 sockets, each holding an Intel(R) Xeon(R) Gold 5318H CPU (18 physical cores) and 194GB of RAM. We used Java version 1.8.0_231 (Oracle Hotspot build 25.231-b11) and rr [30] version 5.5.0. Unless otherwise noted, all experiments were performed in a single socket using 9 threads (half the available cores to allow for unrestricted leader/follower execution), with a maximum heap size of 12G, using a shared memory buffer of 500MB per thread, and repeated 10 times. We measure performance overhead using the numbers reported by the DaCapo benchmark suite, and we measure memory overhead via the total size of the heap when each benchmark finishes (regardless of how much is used by then). The total heap size serves as proxy for a high-water mark of memory usage, and we report the maximum observed in 10 runs.

To benchmark client-server applications, we adapted a previous H2 benchmark [35] that splits the client from the server, and executed the server with JMVX— `h2 server` — instead of using DaCapo's client-server benchmarks that run both client and server together, which is not a realistic deployment of MVX or RR.

To avoid skewing `pmd`'s results by analyzing instrumented code, we manually identified which classes the workload analyzes and added them to JMVX's deny list. We did not include the eclipse benchmark for the reasons explained in Section 3.8. Our evaluation includes all benchmarks from chopin that support our Java version, and reverts to the same benchmark on bach otherwise. Table 2 lists all the benchmarks we use, and their respective version.

To provide a clear picture of JMVX's performance, we answer the research questions (RQs):

- **RQ1**: What is the performance overhead of JMVX's instrumentation?
- **RQ2**: What is JMVX's performance overhead on Record-Replay (RR)?
- **RQ3**: What is JMVX's performance overhead on Multi-version execution (MVX)?
- **RQ4**: What is JMVX's performance overhead for supporting multi-threading?
- **RQ5**: What impact does the shared memory buffer size have on JMVX's MVX performance?
- **RQ6**: What is JMVX's memory overhead for supporting RR and MVX?
- **RQ7**: How does JMVX compare to similar RR systems [7, 30]?

4.1 JMVX's Instrumentation

To measure the overhead of the bytecode instrumentation alone (Section 3.3), we designed an experiment in which we used a passthrough strategy to intercept all the relevant methods but simply redirect them straight to the underlying implementation. We repeated the experiment without intercepting any methods related with multi-threading (Section 3.5).

Answer to RQ1: The overhead imposed by JMVX's instrumentation averages 2% without multi-threaded support, and 5% with, which is negligible.

4.2 Record/Replay

To measure how JMVX performs when recording, we recorded the execution of each DaCapo benchmark with and without support for multithreading. Table 2 shows the results. We also compare JMVX with rr [30] and Chronicler [7], which are the closest RR system. rr limits each program to using a single core, which causes an increase in overhead for multi-threaded benchmarks. Furthermore, rr fails to record/replay avrora, jme, and jython. Avrora causes an internal assertion in rr to fail related with locking/thread scheduling. Jme and jython crash when trying to get information about the machine they are running on, which rr does not support.

For the benchmarks that ran for each system respectively, rr's overhead is 333% whereas JMVX, with synchronization, is 25%. Note that Chronicler does not support multithreading, JMVX performs better than Chronicler in the same setting (8% slowdown vs 14%). We were not able to run Chronicler on our machine as the original prototype requires an older version of OSX that is not available anymore, so we include the numbers reported in the original paper. For JMVX, full multi-threaded support incurs an average slowdown of 25%.

To measure how JMVX performs when replaying a recorded execution, we replayed the recording of each DaCapo benchmark with and without support for multithreading. As Table 3 shows, only 7 DaCapo benchmarks can be replayed without multithreading support, the other benchmarks crash

Table 2. Performance and memory overhead when recording using JMVX and similar RR systems [7, 30]. **Vanilla** refers to DaCapo without any RR system. **No Sync** refers to JMVX without recording multi-threading information (Section 3.5). The superscript notes the version of each benchmark: B for Bach, and C for Chopin. Each execution consists of a single benchmark run, we report the average of 10 executions. The table also shows the results reported by rr [30] and Chronicler [7].

Program	**Vanilla**	[30]	[7]	**JMVX**			
				Nosync		Sync	
	Time	*Perf*	*Perf*	*Perf*	*Mem*	*Perf*	*Mem*
avrora^C	12290.4 ± 57.6	—	1.01×	1.13×	1.00×	1.26×	1.00×
batik^C	8919.3 ± 58.8	2.20×	1.08×	1.04×	1.00×	1.08×	1.00×
fop^C	7650.4 ± 89.1	2.72×	1.36×	1.16×	1.02×	1.18×	1.02×
h2^B	11015.4 ± 254.8	2.07×	1.06×	1.01×	1.00×	1.62×	1.00×
h2 server	405821.0 ± 13954.5	1.36×	—	1.03×	1.00×	1.06×	1.00×
jme^C	9108.3 ± 22.3	—	—	1.05×	1.00×	1.05×	1.00×
jython^C	28924.4 ± 241.3	—	1.12×	1.06×	1.20×	1.87×	1.17×
luindex^B	2443.8 ± 14.7	2.99×	1.01×	1.25×	1.00×	1.35×	1.00×
lusearch^B	2424.1 ± 237.5	6.43×	1.39×	1.10×	1.17×	1.32×	1.00×
pmd^C	16228.9 ± 393.4	5.01×	1.11×	1.07×	1.18×	1.11×	1.13×
sunflow^C	31496.7 ± 1952.4	6.14×	1.01×	0.88×	3.33×	0.90×	3.33×
xalan^C	8216.4 ± 674.8	10.04×	1.21×	1.14×	1.02×	1.17×	1.02×
AVG		4.33×	1.14×	1.08×	1.24×	1.25×	1.22×

or diverge before completion. Chronicler[7] does not report the performance for replaying any DaCapo recording. JMVX is faster than rr, primarily due to JMVX's ability to replay with multiple threads, as JMVX's synchronized mode replay is on average faster than rr. JMVX imposes a 13% slowdown on replay without support for multi-threading, and 73% with. This experiment also shows the importance of capturing thread synchronization to ensure a high-fidelity replay that behaves equivalently to the original recording.

We note that the h2 server replay runs faster than vanilla. This is a well known effect observed when replaying server executions [24, 30] due to the absence of delay when accepting new client connections and due to replacing expensive network interactions (*e.g.*, receiving/sending data through a TCP socket) with fast interactions with the replay log on disk. H2 executes as a library, with network connections replaced by function calls, and thus does not observe such fast replays.

Sunflow seems to execute faster with JMVX. We note that the standard deviation is quite high (almost 2 seconds for a 31 second average execution), which we verified by running the benchmark manually. As such, we can conclude that the overhead introduced by JMVX on sunflow is undistinguishable from experimental error.

Answer to RQ2: The performance for the recorder averages 8% slowdown without synchronization and 25% with whereas the follower averages 13% slowdown without synchronization and 73% with.

Answer to RQ7: JMVX is competitive with similar RR systems. JMVX has similar performance to Chronicler when multithreading is turned off. With and without multithreading support, JMVX can out preform rr as it uses more of the available processors.

Table 3. Performance and memory overhead when replaying using Jmvx and similar RR systems [7, 30], together with the size of the recordings, compared against **Vanilla** on Table 2. **No Sync** refers to Jmvx without recording multi-threading information (Section 3.5). Recording sizes are reported uncompressed (*Raw*) and compressed (*Gzip*). The superscript notes the version of each benchmark: B for Bach, and C for Chopin. Each execution consists of a single benchmark run, we report the average of 10 executions. The table also shows the results reported by rr [30], which limits programs to running on a single core. Data for Chronicler [7] is not present as it cannot replay multithreaded programs (Section 5), nor were we able to rerun the system.

Program	[30]		**JMVX**							
			Nosync				Sync			
	Perf	*Size*	*Perf*	*Mem*	*Size*		*Perf*	*Mem*	*Size*	
		Gzip			*Raw*	*Gzip*			*Raw*	*Gzip*
avrora^C	—	—	1.22×	1.00×	121M	7M	1.28×	1.00×	329M	25M
batik^C	3.24×	156M	0.99×	1.00×	470M	153M	1.04×	1.00×	475M	153M
fop^C	2.91×	50M	—	—	48M	43M	1.17×	1.00×	49M	44M
h2^B	5.34×	36M	—	—	75M	16M	2.60×	1.02×	1.4G	68M
h2 server	0.34×	68M	—	—	11G	109M	0.47×	0.96×	12G	159M
jme^C	—	—	1.04×	1.00×	231M	219M	1.15×	1.00×	231M	219M
jython^C	—	—	1.05×	1.11×	506M	164M	2.14×	1.03×	873M	218M
luindex^B	3.49×	37M	1.48×	1.00×	61M	23M	1.65×	1.00×	64M	24M
lusearch^B	7.08×	60M	—	—	252M	96M	4.10×	1.00×	384M	119M
pmd^C	5.30×	39M	1.22×	1.03×	156M	30M	2.31×	0.96×	526M	44M
sunflow^C	6.33×	33M	0.89×	1.04×	12M	8.5M	0.93×	1.04×	12M	8.5M
xalan^C	8.24×	38M	—	—	330M	71M	1.95×	0.84×	351M	75M
AVG	4.70×		1.13×	1.03×			1.73×	0.99×		

4.3 Multi-version Execution

To measure how Jmvx performs on MVX, we ran each DaCapo benchmark in MVX mode with two variants, leader and follower. We disabled instrumenting `System.currentTimeMillis` inside DaCapo code and we used the divergence handler described in Section 3.7.1 to allow leader and follower to report different completion times. Table 4 shows the results.

As before, we repeated the experiment with and without support for multi-threading. We can see that the results for leader and follower track the results for recorder and replayer discussed in the previous section. Leader has an average overhead of 5% without synchronization and 47% with. The follower has an average overhead of 5% without synchronization and 80% with. The performance overhead increases for benchmarks that call more non-deterministic methods, which we can estimate by the *No sync* log sizes on Table 2, and for benchmarks that use more synchronization, which we can estimate by the *Sync* log sizes in the same table. Note that users experience the overhead on the leader, as the follower executes in the background. As such, and as we explain in Section 3.5.3, Jmvx is biased towards lower leader/recorder overhead.

Answer to RQ3: The performance overhead imposed by Jmvx averages 5% without synchronization and 47% with synchronization for performing MVX, as experienced by the user (*i.e.,* leader variant) The follower variant experiences on average a higher performance overhead of 5% without synchronization and 80% with.The performance costs can be predicted by the recording log sizes.

Table 4. Performance and memory overhead of Jmvx's MVX. **Vanilla** refers to DaCapo without Jmvx, **No Sync** refers to Jmvx without multi-threading support (Section 3.5). Each execution consists of 2 warm-ups (discarded); and a 3rd run (measured); we report the average of 10 executions.

Program	**Vanilla**	**Leader**				**Follower**			
		No Sync		**Sync**		**No Sync**		**Sync**	
	Time	*Perf*	*Mem*	*Perf*	*Mem*	*Perf*	*Mem*	*Perf*	*Mem*
avrora[C]	11510.2 ± 49.2	1.05×	1.00×	1.16×	1.00×	1.05×	1.00×	1.16×	1.00×
batik[C]	6281.3 ± 13.9	1.01×	1.00×	1.04×	1.00×	1.01×	1.00×	1.05×	1.00×
fop[C]	2360.6 ± 64.5	1.09×	1.00×	1.11×	1.00×	1.11×	1.00×	1.13×	1.00×
h2[B]	10320.7 ± 183.0	—	—	1.64×	1.00×	—	—	1.90×	1.39×
h2 server	716599.0 ± 52827.3	—	—	1.13×	1.00×	—	—	1.12×	0.71×
jme[C]	7506.2 ± 7.9	1.00×	1.00×	1.00×	1.00×	1.02×	1.00×	1.03×	1.00×
jython[C]	22697.9 ± 44.6	1.03×	1.38×	2.41×	1.22×	1.03×	1.29×	2.41×	1.49×
luindex[B]	1282.5 ± 27.6	1.25×	1.00×	1.38×	1.00×	1.25×	1.00×	1.37×	1.00×
lusearch[B]	825.5 ± 95.0	—	—	2.20×	1.00×	—	—	5.78×	0.63×
pmd[C]	7104.7 ± 282.4	—	—	2.22×	0.76×	—	—	2.23×	0.78×
sunflow[C]	30729.1 ± 3011.6	0.88×	1.05×	0.92×	1.02×	0.87×	1.07×	0.95×	1.02×
xalan[C]	3407.3 ± 58.9	—	—	1.46×	1.07×	—	—	1.47×	1.07×
AVG	—	1.05×	1.06×	1.47×	1.01×	1.05×	1.05×	1.80×	1.01×

4.4 Multi-threading

The previous sections discuss the costs related with supporting multi-threaded applications, by comparing the performance overhead when enabling/disabling Jmvx's instrumentation related with threads (explained in Section 3.5). Table 2 shows that the cost of recording increases as we add support for synchronization from 8% to 25%. Table 3 shows that the cost of replaying increases from 13% to 73%. Supporting multi-threading in RR also leads to an increase in the log sizes, proportional to how much synchronization is present in each benchmark. Some benchmarks use multiple threads with little synchronization. For instance, sunflow is a ray-tracer that builds a scene using a single-thread, and then uses multiple threads to render the scene that access the scene without any synchronization. Other benchmarks rely heavily on synchronization, in particular H2 and Jython. H2 is a SQL database that uses synchronization to enforce correct transaction ordering. Jython is a Python interpreter that uses synchronization to match Python's sequential consistent memory semantics. However, Jython's workload in DaCapo is single-threaded. Table 4 shows similar increases in performance overhead to support MVX with and without support for multi-threading.

We note that supporting multi-threading increases the overhead for the follower and replayer variants. As explained in Section 3.5.3, Jmvx is biased towards low overhead when recording by not interfering with thread scheduling and just capturing it. However, for the follower/replayer, Jmvx needs to ensure the same thread scheduling, which is costly as the wrong thread being scheduled keeps the correct thread from making progress for a short while. Such wrong thread schedulings happen non-deterministically and very frequently, which drastically increases the performance overhead on benchmarks in which threads synchronize frequently (*e.g.*, h2, jython, and lusearch). Tables 3 and 4 also shows the consequences of not supporting multi-threading: some benchmarks diverge when multi-threading support is disabled.

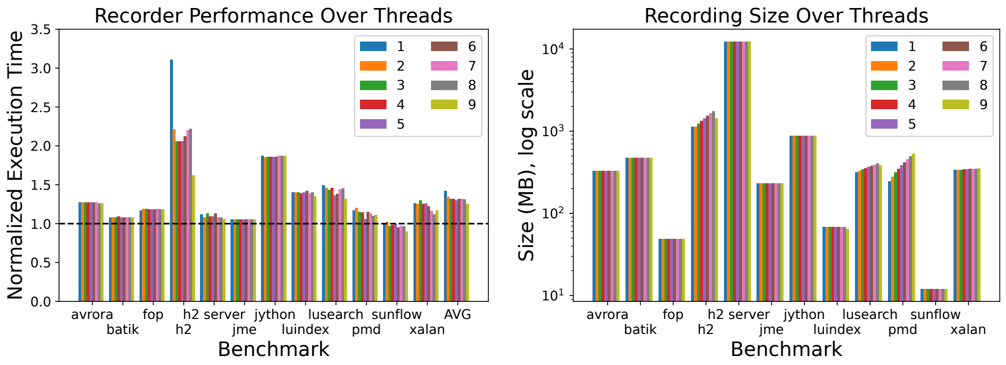

Fig. 10. Jmvx's RR performance overhead and recording size when varying the number of threads. Each bar is normalized to vanilla DaCapo running with the same number of threads (*e.g.,* the bar for 4 threads is normalized against vanilla running 4 threads). We omit the performance overhead for the replayer as the results follow the same trends as the recorder, but with higher overall numbers which match the results in Table 3.

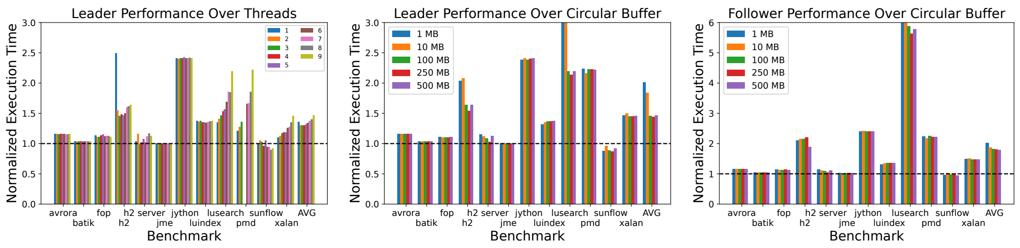

Fig. 11. Jmvx's MVX performance overhead when varying the number of threads and the size of the shared memory buffer. We omit the performance overhead for the follower with different threads as the results follow the same trends as the leader, but with higher overall numbers which match the results in Table 4.

We repeated all experiments by configuring the DaCapo benchmark suite with a varying number of threads, from 1 to 9. Figures 10 and 11 show the results for RR and MVX, respectively. We can see that Jmvx introduces a constant overhead regardless of thread number. Also, note that the recording sizes when varying the number of threads also remain constant. We can conclude that the same workload requires the same amount of synchronization regardless of the number of threads. H2 and lusearch are the two outliers. H2 shows a peak in overhead for one thread. We believe this is due to aggressive JVM optimizations for monitors used by a single thread [3], multiplied by the large amount of synchronization in h2 (note the log sizes in Table 2). Jython has high overhead (1.87x for recorder, 2.14x for replayer, and 2.41x for leader and follower) for this same reason. Jython is single-threaded but uses a large number of locks, which are otherwise biased and optimized when run without Jmvx.

The low synchronization cost multiplied by a large number of synchronization operations highlights Jmvx's costs of capturing logical clocks when running with a single thread. Lusearch is a highly concurrent short-lived benchmark (800ms to complete after warm-up, as measured in Table 4). Such low execution time highlights Jmvx's constant costs that are amortized over time in other benchmarks.

Answer to RQ4: Supporting multi-threaded programs increases Jmvx's performance overhead from 5% to 47% for MVX, from 8% to 25% for recording, and from 13% to 73% for replaying. Disabling support for multi-threading causes 4 benchmarks to diverge: h2, h2 server, lusearch, and xalan. Jmvx's overhead and log sizes are not related to the number of threads, but to the amount of synchronization present in the target program. The amount of synchronization can be predicted by the difference in log sizes when multi-threading support is enabled/disabled.

4.5 Shared Memory Buffer

We repeated the experiments for both MVX and RR using 9 threads and different buffer sizes: 1MB, 10MB, 100MB, 250MB, and 500MB. Figure 11 shows the result. We can see that the follower performance remains unchanged regardless of buffer size. Smaller buffers (*i.e.,* less than 10MB) slow down the leader, as the leader fills them up and then has to wait for the follower to make room before resuming execution. Buffers larger than 100MB do not make any difference in the results, as they provide enough room for the leader to make progress without waiting for the follower.

Answer to RQ5: The leader suffers a significant slowdown for buffers 10MB or smaller, as it has to wait for the follower to make room in the shared buffer. Buffers 100MB or larger have no impact on the follower's performance. The buffer size does not affect the follower's performance.

5 Related Work

There are many Multi-version execution (MVX) systems designed for programs written in C/C++, focusing on security [15, 23, 29, 46, 47], reliability [1, 8, 20, 27, 39], software analysis [32, 34, 49], and availability [19, 33, 37]. Similarly to some of these MVX systems, Jmvx uses a leader/follower architecture [20, 23, 32, 46, 47, 49] and supports multi-threading based on Lamport clocks [20, 32, 46].

The closest work related to Jmvx is Chronicler [7], a Record-Replay (R/R) system for Java programs. Similarly to Jmvx, Chronicler works by instrumenting Java bytecode. However, Chronicler detects non-deterministic methods via a static analysis that tracks the transitive closure of methods that call native methods. Chronicler may consider as non-deterministic methods belonging to libraries or to the application, and logs the whole execution of the method when recording. Later, replaying, simply skips the whole method. As Chronicler instruments methods that operate on complex data, it needs to detect which objects each instrumented method may modify, and record those objects as well. Jmvx, instead, intercepts low level methods manually found to be non-deterministic and that operate on primitive data (*e.g.,* bytes, integers, strings). Furthermore, Chronicler's replay does not ensure the same internal state of the application because complex methods may be skipped; which reduces Chronicler's fidelity during replay and makes it unsuitable for MVX as the follower (adapted from a replayer) does not keep enough internal state to become the leader. Jmvx ensures the internal state of all objects during replay is equivalent. Finally, Chronicler does not support multi-threaded applications; and it does not record class-loading, assuming the recorder and replayer have access to the same classpath. Octet [11] is a system for capturing read/write dependencies between threads, which can be used to implement an RR system for Java [10]. Octet does not support class loading (it requires all classes to be loaded before recording/replaying), requires a custom research JVM, does not support MVX, and introduces a higher performance overhead than Jmvx for recording: 49% (29% for replay). DejaVu [13] is an RR system for multi-threaded Java programs, implemented as a custom interpreter with deterministic scheduling on a single CPU, and does not support modern JIT compilers. LEAP [21] is a deterministic record/replay tool for Java that captures shared memory operations based on a static analysis, but does not capture any other source of non-determinism apart from thread scheduling (*i.e.,* LEAP does not capture networking, file I/O, time). We believe it is possible to combine LEAP's support of shared memory operations with Jmvx's support of all sources of non-determinism, and leave this as exciting future work.

There are many RR systems designed for programs written in C/C++ [5, 16, 17, 24, 30, 31, 45, 48]. Some [5, 30, 31] can handle the complexity of a Java program executing inside the JVM. For instance, rr [30] can record the execution of a whole internet browser (which includes support for Javascript) and replay it later.

Unfortunately, multithreading support remains a challenge for RR systems. To ensure replay fidelity, most RR systems record programs in a virtual single CPU [13, 16, 17, 30, 45, 48], forcing them to run in a single thread; so that the RR system can control which thread is scheduled at any moment, and record that decision. Later, the RR system can replay the same scheduling. Of course, limiting executions to a single thread limits the fidelity of the recording as it does not capture bugs related to race conditions (with monitors). Single threaded execution also severely limits the performance of a multithreaded system, which can result in dramatic slowdowns while recording. For instance, executing the DaCapo benchmark with 9 threads but using a single CPU yields an average slowdown of 3.71× and maximum of 12.65×. Jmvx does not limit multi-threading in this way, and results in much lower performance overheads when recording/replaying.

Ensuring RR fidelity in the presence of multi-threading requires capturing the order of events between threads during recording, and then enforcing it during replay Jmvx uses Lamport clocks for that purpose. PinPlay[31] is an RR system implemented via dynamic instrumentation of native code that logs memory accesses in user space via shadow memory. PinPlay logs all stores, and logs loads when the value on main memory differs from the value on shadow memory; which implies that main memory was changed externally (*e.g.,* by the OS during a system call). Then, PinPlay identifies data dependencies in the logs and creates a partial ordering. As a result, PinPlay enforces the order on data shared between threads, and allows threads to modify thread-private data unimpeded. PinPlay's recordings thus have higher fidelity than Jmvx, but at a much higher performance cost (performance overhead ranging between 36x–147x when recording and 10x–36x when replaying). Scribe[24], an RR system implemented as an OS module for Linux, logs system calls and supports multithreaded programs via rendezvous points and sync points. Rendezvous points are code locations where threads compete for a resource. Each resource is given a unique sequence number logged in the recording when acquiring that resource. During replay, Scribe uses the sequence number to order requests to that resource. Sync points are deterministic points where a program can switch context (*e.g.,* system calls). Scribe defers non-deterministic asynchronous events to the next sync point (*e.g.,* wait for the next system call to deliver a signal), thus recording a log that can be replayed deterministically. Jmvx does not support such non-deterministic events directly, instead it defers to capturing synchronized operations that react to such async events (*e.g.,* notifying a thread will cause that thread to return from `wait`, which Jmvx then captures).

Time Traveling Debugging systems (TTDs), which allow developers to step back in time after a breakpoint, share similarities with RR systems. The debugger records snapshots frequently, which log non-deterministically program actions. To travel backwards in time, the debugger reverts the state of the program to a prior snapshot and replays operations to reach the desired point. One such system is TARDIS[4], which supports programs written in a managed language (C#) and supports multi-threading, by runing one thread at a time and capturing data on context switches. Crochet [6] supports similar checkpoint/rollback by performing a whole-heap copy at the time of checkpoint, and restoring it on rollback. Crochet supports multiple threads without limitations, and achieves high efficiency when copying the whole heap through a Copy-on-Write policy on a per-object basis. TTDs and checkpoint/rollback systems have different goals from RR systems, as they must enable re-execution in the same process; instead of later on a separate process. For instance, they do not need to capture open network sockets, as these remain open during the program execution.

6 Conclusion

Performing Multi-version execution (MVX) for programs written in managed languages is challenging due to the inherent non-determinism of the underlying High Level Language Virtual Machine (HLLVM): two executions of the same program behave different in terms of Just-In-Time compilation, code loading, garbage collection, and so on. MVX executes many processes at the same time (variants), and such differences cause MVX to terminate due to variants diverging.

This paper presents the design, implementation, and evaluation of Jmvx— a novel MVX system for managed languages that automatically identifies and instruments 71 non-deterministic methods within the JVM and target program, and ensures that all variants obtain the same results from those methods. Jmvx uses a circular buffer located in shared memory for fast communication between variants, and supports multi-threaded applications by capturing the order of thread synchronization operations in one variant, and ensuring all other variants follow the same order. Furthermore, Jmvx also supports Record/Replay (RR) by writing all captured non-determinism to disk while recording, and reading it from disk while replaying. Our evaluation shows Jmvx supports MVX on a suite of Java benchmarks representative of typical Java applications, with an average performance overhead of 5% |47%, depending on whether multi-threading support is disabled|enabled. Jmvx supports RR over the same suite of benchmarks with an average performance overhead of 8% |25% to record; and 13% |73% to replay.

7 Acknowledgements

This work was funded in part by NSF CCF-2227183. We would like to thank: David Woloch, Linfeng Li, Meer Shah, Luis De Santos, and Jason Pereira for their work on the Jmvx prototype; John Fike for IT support during development and testing; and our anonymous reviewers for their useful feedback.

8 Data Availability Statement

We have released a replication package that includes the prototype implementation of Jmvx, the scripts that automate all the experiments, the dataset we obtained when evaluating Jmvx, and detailed instructions on how to use Jmvx and replicate our evaluation [40], together with a repository containing all of the source code [41]. The replication package and source code are freely available with permissive licenses that allow unrestricted reuse.

References

[1] Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. 2022. SaBRe: Load-Time Selective Binary Rewriting. *Int. J. Softw. Tools Technol. Transf.* 24, 2 (apr 2022), 205–223. https://doi.org/10.1007/s10009-021-00644-w

[2] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, JD Mitchell, Kelvin Nilsen, et al. 2000. The "double-checked locking is broken" declaration. https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html. Accessed: 2023-11-15.

[3] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. 1998. Thin Locks: Featherweight Synchronization for Java. *SIGPLAN Not.* 33, 5 (may 1998), 258–268. https://doi.org/10.1145/277652.277734

[4] Earl T. Barr and Mark Marron. 2014. Tardis: affordable time-travel debugging in managed runtimes. *ACM SIGPLAN Notices* 49, 10 (Oct. 2014), 67–82. https://doi.org/10.1145/2714064.2660209

[5] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. 2011. Karma: Scalable Deterministic Record-Replay. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. Association for Computing Machinery, New York, NY, USA, 359–368. https://doi.org/10.1145/1995896.1995950

[6] Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:31. https://doi.org/10.4230/LIPIcs.ECOOP.2018.17

[7] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 362–371. https://doi.org/10.1109/ICSE.2013.6606582

[8] Emery Berger and Benjamin Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. *Sigplan Notices - SIGPLAN* 41, 158–168. https://doi.org/10.1145/1133255.1134000

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[10] Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. 2015. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *Proceedings of the Principles and Practices of Programming on The Java Platform* (Melbourne, FL, USA) *(PPPJ '15)*. Association for Computing Machinery, New York, NY, USA, 90–101. https://doi.org/10.1145/2807426.2807434

[11] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. OCTET: Capturing and Controlling Cross-Thread Dependences Efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 693–712. https://doi.org/10.1145/2509136.2509519

[12] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

[13] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (Welches, Oregon, USA) *(SPDT '98)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/281035.281041

[14] Intel Corporation. 2007. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation.

[15] Benjamin Cox and David Evans. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity. In *15th USENIX Security Symposium (USENIX Security 06)*. USENIX Association, Vancouver, B.C. Canada. https://www.usenix.org/conference/15th-usenix-security-symposium/n-variant-systems-secretless-framework-security-through

[16] Pavel Dovgalyuk, Denis Dmitriev, and Vladimir Makarov. 2015. Don't Panic: Reverse Debugging of Kernel Drivers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 938–941. https://doi.org/10.1145/2786805.2803179

[17] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224. https://doi.org/10.1145/844128.844148 Publisher: ACM New York, NY, USA.

[18] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[19] Petr Hosek and Cristian Cadar. 2013. Safe Software Updates via Multi-Version Execution. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 612–621. https://doi.org/10.1109/ICSE.2013.6606607

[20] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-Version Execution Framework. *SIGPLAN Not.* 50, 4 (mar 2015), 339–353. https://doi.org/10.1145/2775054.2694390

[21] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 385–386. https://doi.org/10.1145/1882291.1882361

[22] Kohlschütter Search Intelligence. [n. d.]. junixsocket: Unix Domain Sockets in Java (AF_UNIX). https://github.com/kohlschutter/junixsocket. Accessed: 2023-11-15.

[23] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*. Institute of Electrical and Electronics Engineers, Inc., 431–442. https://doi.org/10.1109/DSN.2016.46

[24] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *ACM SIGMETRICS Performance Evaluation Review* 38, 1 (June 2010), 155–166. https://doi.org/10.1145/1811099.1811057

[25] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[26] Dmitry V. Levin et al. [n. d.]. strace - linux syscall tracer. https://strace.io. Accessed: 2023-11-15.

[27] Liming Chen and A. Avizienis. 1995. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.* 113–. https://doi.org/10.1109/FTCSH.1995.532621

[28] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[29] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem Execution for Efficient Live Patch Testing. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 43.

[30] Robert O'Callahan, Chris Jones, Albert Noll, Nathan Froyd, Nimrod Partush, and Kyle Huey. [n. d.]. Engineering Record And Replay For Deployability. ([n. d.]), 377–389.

[31] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs *(CGO '10)*. ACM, 2–11. https://doi.org/10.1145/1772954.1772958 Book Title: Proceedings of the 8th annual IEEE/ACM international symposium on code generation and optimization.

[32] Luís Pina, Anastasios Andronidis, and Cristian Cadar. 2018. FreeDA: Deploying Incompatible Stock Dynamic Analyses in Production via Multi-Version Execution. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '18)*. ACM. https://doi.org/10.1145/3203217.3203237

[33] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 573–585. https://doi.org/10.1145/3297858.3304063

[34] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. 2017. A DSL Approach to Reconcile Equivalent Divergent Program Executions. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX.

[35] Luís Pina, Luís Veiga, and Michael Hicks. 2014. Rubah: DSU for Java on a Stock JVM. In *Proceedings of the ACM 2014 International Conference on Object-Oriented Programming Languages, Systems, and Applications* (Portland, OR, USA) *(OOPSLA '14)*. ACM. https://doi.org/10.1145/2660193.2660220

[36] Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* 17, 7 (jul 1974), 412–421. https://doi.org/10.1145/361011.361073

[37] Weizhong Qiang, Feng Chen, Laurence T. Yang, and Hai Jin. 2017. MUC: Updating cloud applications dynamically via multi-version execution. *Future Generation Computer Systems* 74 (2017), 254–264. https://doi.org/10.1016/j.future.2015.12.003

[38] Ugnius Rumsevicius, Siddhanth Venkateshwaran, Ellen Kidane, and Luís Pina. 2023. Sinatra: Stateful Instantaneous Updates for Commercial Browsers Through Multi-Version eXecution. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2023.26

[39] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 33–46. https://doi.org/10.1145/1519065.1519071

[40] David Schwartz, Ankith Kowshik, and Luís Pina. 2024. Artifact for Jmvx: Fast Multi-threaded Multi-Version eXecution and Record-Replay for Managed Languages. https://doi.org/10.5281/zenodo.12637140

[41] David Schwartz, Ankith Kowshik, and Luís Pina. 2024. Source for JMVX prototype. https://github.com/bitslab/jmvx

[42] David Seal. 2000. *ARM Architecture Reference Manual* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[43] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[44] Martin D. Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. 2011. Disruptor : High performance alternative to bounded queues for exchanging data between concurrent threads. https://lmax-exchange.github.io/disruptor/disruptor.html

[45] undo.io. [n. d.]. UndoDB reversible debugging tool for Linux and Android. https://undo.io/resources/undodb-reversible-debugging-tool-linux-and-android. Accessed: 2023-11-15.

[46] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming Parallelism in a Multi-Variant Execution Environment. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 270–285. https:

//doi.org/10.1145/3064176.3064178

[47] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 167–179. https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert

[48] Min Xi, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. 2007. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*.

[49] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: Compositing Security Mechanisms through Diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 271–283. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xu-meng

Received 2024-04-05; accepted 2024-08-18