Deterministic Replay of Java Multithreaded Applications

Jong-Deok Choi and Harini Srinivasan
IBM T. J. Watson Research Center
Hawthorne, NY, USA, 10532
jdchoi@us.ibm.com harini@us.ibm.com

Abstract

Threads and concurrency constructs in Java introduce non-determinism to a program's execution, which makes it hard to understand and analyze the execution behavior. Non-determinism in execution behavior also makes it impossible to use execution replay for debugging, performance monitoring, or visualization. This paper discusses a record/replay tool for Java, DejaVu, that provides deterministic replay of a program's execution. In particular, this paper describes the idea of the *logical thread schedule*, which makes DejaVu efficient and independent of the underlying thread scheduler. The paper also discusses how to handle the various Java synchronization operations for record and replay. DejaVu has been implemented by modifying the Sun Microsystems' Java Virtual Machine.

1 Introduction

The ubiquity of the Java programming language in current software systems has made development of advanced programming environment tools for writing efficient and correct Java programs very important. Building such tools, however, is non-trivial because of non-determinism in Java, introduced by features such as multiple threads, windowing events, network events/messages and general input/output operations. For example, repeated execution of a program is common while debugging a program, and non-determinism may disallow a bug that appeared in one execution instance of the program from appearing in another execution instance of the same program [8, 13, 11, 10, 12, 5, 1, 2, 4].

A key missing element in current debuggers and monitoring tools for Java is the ability to provide a deterministic replay of a non-deterministic execution instance. In this paper, we present a record/replay tool for Java, called DejaVu, that enables deterministic replay of concurrent Java programs. In particular, we describe how DejaVu deterministically replays non-deterministic execution behavior due to threads and related concurrent constructs such as synchronization primitives. DejaVu, to our knowledge, is the first tool that handles all the Java synchronization operations in the context of deterministic replay of multithreaded Java applications.

Appeared in the 1998 ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT). August 1998. pp 48-59.

In addition to threads and concurrent constructs, windowing events/inputs and network events can also attribute to non-deterministic execution behavior. Although the current DejaVu implementation handles network and windowing events, they are the topic of another paper [3].

DejaVu, developed as an extension to the Sun Microsystems' Java Virtual Machine (JVM), runs in two modes: (1) The record mode, wherein, the tool records the *logical thread schedule information* of the execution while the Java program runs; and (2) the replay mode, wherein, the tool reproduces the execution behavior of the program by enforcing the recorded logical thread schedule.

DejaVu uses a portable approach and is independent of the underlying thread scheduler, be it an operating system scheduler or a user-level thread scheduler. Although described in the context of Java, the techniques employed by DejaVu apply to general multithreaded programming systems with similar synchronization primitives. Another advantage of DejaVu is that it can be used on a multiprocessor system as well, though higher overhead than on a uniprocessor system is expected.

The rest of the paper is organized as follows: Section 2 discusses the notion of logical thread schedule and how to identify a logical thread schedule in a program execution. Section 3 discusses our approach for recording logical thread schedule information. This section also describes how to handle the various synchronization operations of Java. Section 4 discusses how we replay a logical thread schedule. Section 5 discusses the DejaVu implementation and some performance results. Section 6 compares our approach to previous approaches. Finally, Section 7 concludes the paper.

2 Deterministic Replay

Replaying a multithreaded program on a uniprocessor system can be achieved by first capturing the thread schedule information during one execution of the program, and then enforcing the exact same schedule when replaying the execution [12]. A thread schedule of a program is essentially a sequence of time intervals (time slices). Each interval in this sequence contains execution events of a single thread. Thus, interval boundaries correspond to thread switch points. Capturing the actual thread schedule information is not always possible, in particular, with commercial operating systems. Rather than relying on the underlying physical thread scheduler (either an operating system or a user-level

```
class Test {
  static public volatile int f = 0;
      // shared variable
  static public volatile int g = 20;
      // shared variable
  static public void main(String argv[]) {
    int j; // local variable
    MyThread t1 = new MyThread();
    t1.start();
    j = 20;
    System.out.println(``f = `` + f
      + `` j = `` + j);
  }
}

class MyThread extends Thread {
  public void run() {
    int k; // local variable
    k = 5;
    Test.f = Test.f + k;
    Test.g = Test.g - k;
  }
}
```

Figure 1: Example Program

thread scheduler) for thread scheduling information, we capture the *logical thread schedule* information that can be computed without any help from the thread scheduler. We refer to the thread schedule information obtained from a thread scheduler as the *physical thread schedule* information, and each time interval in a physical thread schedule as a *physical schedule interval*.

To better understand the notion of logical thread schedule, consider a simple multithreaded Java program shown in Figure 1. Here, thread `main` starts a child thread, `t1`. Both `main` and `t1` can access the (shared) member variables, `f` and `g` – `main` reads `f` and `t1` reads and writes variables `f` and `g`.[1] Variables `k` and `j` are thread-local variables, while `f` and `g` are thread-shared variables.[2]

Figure 2 depicts a few execution instances (physical thread schedules) of the example program on a uniprocessor machine: time is marked in the vertical direction. In Figure 2(a), the shared variable `f` is incremented by `t1` before `main` can print the value of `f`. Hence, for this execution, the value printed is `5`. The difference between the execution instances (a) and (b) in Figure 2 is that, in the latter, variable `j` is updated before thread `t1` starts executing. This does not affect the execution behavior of the program because accessing a local variable is a local event of a thread. The value of `f` printed by thread `main` is still `5`. However, in Figure 2(c), `main` prints the value of `f` before `t1` updates it. Hence, the value printed is `0`. Likewise, in Figure 2(d), the value of `f` that gets printed by `main` is `0`. The difference between the thread schedules (c) and (d) in the figure lies in the order of local variable accesses. In (c), `k` is updated before the shared variable `f` is accessed by thread `main`; in (d), `k` is updated after `f` is accessed in `main`.

[1]Declaring the shared variables `volatile` forces each thread not to treat them as locals in the absence of any explicit synchronization operations.

[2]We use the term variable to denote the unique memory location associated with the variable at an execution point.

An execution behavior of a thread schedule can be different from that of another thread schedule, if the order of shared variable accesses is different in the two thread schedules. Thus, it is possible to classify physical thread schedules with the same order of shared variable accesses into equivalence classes. In our example, schedules (a) and (b) belong to the same equivalence class. Likewise, schedules (c) and (d) belong to one equivalence class. We collectively refer to all the physical thread schedules in an equivalence class as a *logical thread schedule*.

Synchronization events can potentially affect the order of shared variable accesses, and thus affect the possible logical thread schedules. Java provides several flavors of synchronization:

- `monitorenter`, `monitorexit` that mark the begin and end, respectively, of a critical section. The semantics of a critical section is that only one thread can execute the section of code at any given time. A different thread can enter the critical section only after the first has executed the `monitorexit` operation. However, threads *compete* to enter a critical section, and during different execution instances, threads may acquire access to the critical section in different orders. The Java synchronized methods and statement blocks can be implemented using `monitorenter` and `monitorexit` operations.
- `wait`, `notify/notifyAll` that can be used to coordinate the execution order of multiple threads. A thread that has executed a `wait` on an object must wait to be notified by a different thread executing a `notify` operation on the same object. The thread that executes the `notify` on an object wakes up an *arbitrary* thread waiting on the same object. `notifyAll` can be used to wake up all the corresponding waiting threads.
- `suspend` and `resume` are also used to coordinate the execution order. A thread can suspend another thread or *itself*. A suspended thread must be explicitly resumed by another thread.
- Finally, it is possible to also interrupt the execution of a thread at any point by a different thread using the `interrupt` operation.

All of the above synchronization operations affect the execution order of threads, which in turn can affect the order of shared variable accesses and hence the logical thread schedules. In addition, the interactions between synchronization operations in the user application are themselves part of the application and have to be reproduced for the user to correctly understand their program behavior. It is therefore imperative, in a record/replay tool, to capture all these synchronization events and the shared variable accesses in order to reproduce the exact same execution behavior of the program. We collectively refer to the synchronization events and shared variable accesses as *critical events*. A logical thread schedule is a sequence of intervals of critical events, wherein each interval corresponds to the critical and non-critical events executing consecutively in a specific thread. (Note that events of a single thread are well ordered according to their temporal order during the execution of the thread.)

Note that it is not necessary to trace each critical event individually – in particular, for a particular logical thread schedule, if multiple critical events occur in succession and are not separated by a thread switch, it suffices to trace

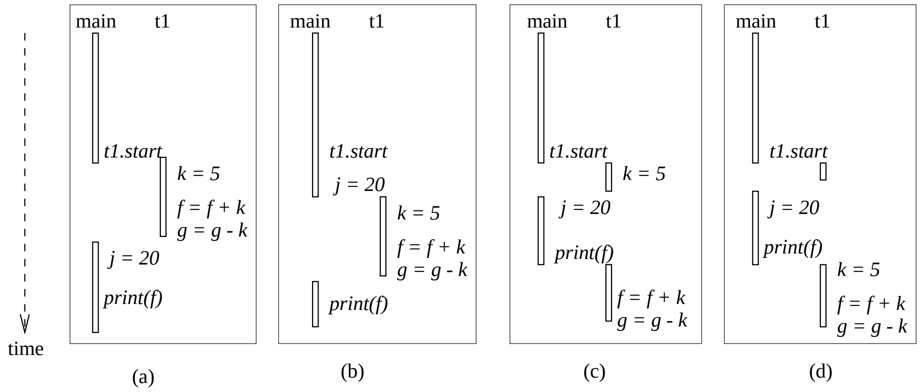


Figure 2: Possible Physical Thread Schedules for Example Program. Object Updates are indicated by a variable name followed by an equals sign.

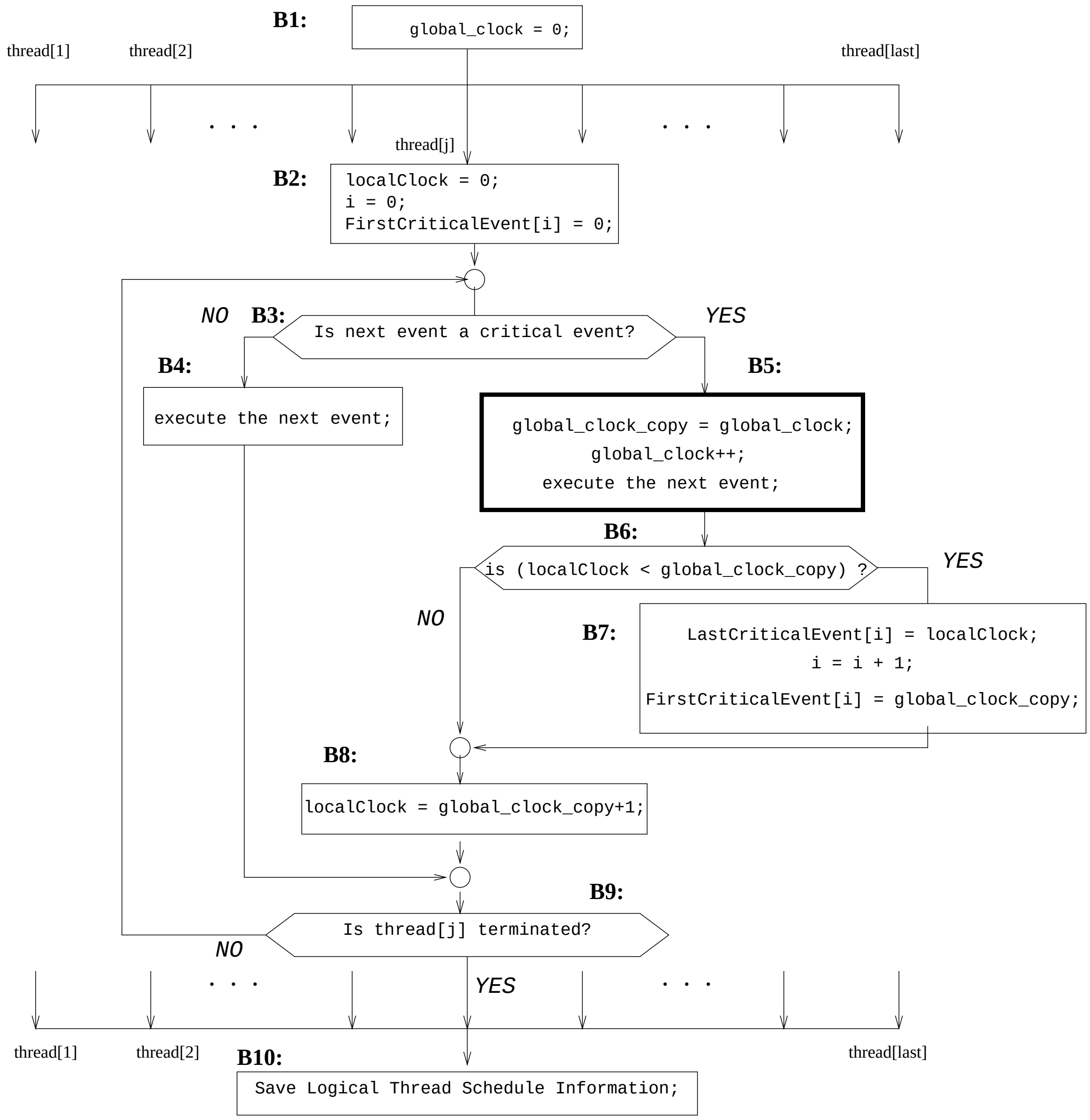


Figure 3: Flowchart to Capture the Logical Thread Schedule

the critical event *interval*, *i.e.*, the first and the last critical event. (In this paper, we use the term *tracing events* to refer to monitoring events and recording the monitored events in main memory or a secondary storage place such as disks.) Tracing the critical event interval rather than every critical event reduces the space requirements for the traces, and consequently improves the performance of the replay tool. For example, in both the logical schedules in Figure 2, the shared variables `f` and `g` are accessed in thread `t1`, uninterrupted by thread switches, in the order `f`; `f`; `g`; `g` (corresponding to the reads and writes of `f` and `g` respectively). Rather than trace each of these critical events, we simply trace the first access to `f`, which is a read, and the last access to `g`, which is a write, in `t1`. Before proceeding to discussing record and replay of critical events, the rest of this section formalizes the notion of logical thread schedule and discusses how such schedules can be identified.

2.1 Capturing Logical Thread Schedules

The *logical thread schedule*, captured by DejaVu, contains enough thread schedule information to reproduce the execution behavior of the program during a replay. The approach to capture logical thread schedule information is based on a global clock (i.e., time stamp) for the entire application and one local clock for each thread.

2.1.1 Logical Schedule Intervals

The logical thread schedule of an execution instance on a uniprocessor system is an ordered set of critical event intervals, called *logical schedule intervals*. Each logical schedule interval is a set of maximally consecutive critical events of a thread. Formally, a logical schedule interval is a (temporally well ordered) non-empty set of critical events with the following properties:

1. all critical events of the logical schedule interval belong to the same thread;
2. given any two critical events C_i and C_j of the logical schedule interval, all critical events of the thread that happened between C_i and C_j also belong to this logical schedule interval; and
3. no two adjacent intervals belong to the same thread.

The implication of 3 above is as follows: If there are two physical schedule intervals P_i and P_j of a thread such that no physical schedule intervals between P_i and P_j have critical events, then all the consecutive physical schedule intervals of this thread from P_i to P_j are merged into a single logical schedule interval. The two physical schedule intervals of thread `main` in Figure 2(c) and Figure 2(d) are examples of such intervals to be merged into a single logical schedule interval. Such a merge is possible because the first physical schedule interval of `t1` in these figures does not contain a critical event. We need only two logical schedule intervals to capture the four physical schedule intervals in these figures: one logical schedule interval for thread `main` consisting of `t1.start` and `print(f)`, and the other for thread `t1` consisting of the assignments to `f` and `g`. Assignment to `k` in thread `t1` can be done independent of its timing with respect to thread `main` without affecting the execution behavior of the program.

With the above properties, an execution instance of an application on a uniprocessor system has a unique logical thread schedule. We capture this unique logical thread schedule during the record phase, and enforce it during the replay phases to reproduce the same execution behavior. The manner in which non-critical events between two logical schedule intervals are scheduled during a replay does not affect the execution behavior of the program. In the rest of the paper, we will refer to the logical schedule interval as the schedule interval when the meaning is clear.

Each schedule interval, LSI_i, is an ordered set of critical events, and can be represented by its first and last critical events as follows:

$LSI_i =< FirstCriticalEvent_i, LastCriticalEvent_i >$.

We use a global clock that ticks at each execution of a critical event to uniquely identify each critical event. Therefore, $FirstCriticalEvent_i$ and $LastCriticalEvent_i$ can be represented by their corresponding global clock values. While running, each thread captures the $FirstCriticalEvent$ and $LastCriticalEvent$ pair of each of its schedule intervals. The following section provides a detailed description of how we capture $FirstCriticalEvent$ and $LastCriticalEvent$.

The idea of capturing and tracing schedule interval information, and not tracing the exhaustive information on each critical event, as is done in previous approaches [8, 13, 11, 10], is crucial for the efficiency of our replay mechanism. In the trace file generated by the system, we have found it quite general for a schedule interval to consist of thousands of critical events, all of which can be efficiently encoded by two, not thousands of, word-long clock values.

2.1.2 Identifying Schedule Intervals

Although the global clock and a thread's local clock start with the same time value, the local clock stays behind the global clock when a different thread executes a critical event: when the thread is scheduled out, the global clock continues to tick for each critical event executed by other threads while the local clock stays still. We use this observation in capturing the schedule interval for each thread as follows:

- At the beginning, all the local clocks and the global clock have the same value (say, zero).
- When a critical event is executed by a thread, the thread compares its local clock with the global clock.
- If the two clock values are different, the thread has just detected the end of the previous schedule interval and also the start of a new schedule interval.
- The thread executes the critical event, and increments the global clock, both as one atomic operation. (Details of this will be provided below.) The thread then synchronizes the local clock with the global clock.

At the beginning, global_clock is initialized to 0.[3] Figure 3 shows the steps for thread[i] in detail as a representative case. The statements in B5 must be executed atomically. Block B7 corresponds to the case when a new thread schedule interval is detected. Note that, according to the algorithm, all the threads except the main thread will have an initial interval $< 0, -1 >$, which should be ignored. (The initial interval of the main thread could also be $< 0, 0 >$, and should not be ignored. This could happen if the main thread is scheduled out after only one critical event. We assume, however, we can always distinguish the main thread from the other threads.)

Figure 4 shows the timing diagram with four threads: $T1$, $T2$, $T3$, and $T4$. The vertical lines indicate time. The

[3] global_clock is shared by all threads in the program. Other variables in the flowchart are local to each thread.

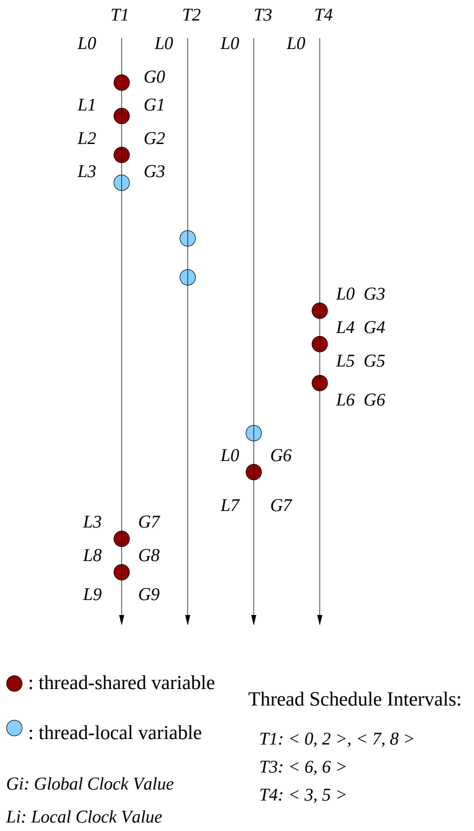


Figure 4: Identifying Thread Schedules

figure shows that $T1$ is the first thread scheduled by the thread scheduler. $T1$ executed three accesses to shared variables followed by one access to a local variable. Then, a thread switch happened, and $T2$ was scheduled in. After accessing two local variables, $T2$ was scheduled out, and $T4$ was scheduled in, and so on. The figure also shows the thread schedule intervals for threads $T1$, $T3$, and $T4$. Note that $T2$'s execution does not contain any critical event and hence, nothing has to be traced for $T2$. Consider $T1$: there are five critical events in $T1$ (in this case all of them correspond to shared variable accesses). The first three critical events occur in succession, and then, there are four thread switches before the next two critical events. The first three critical events of $T1$ correspond to global clock values 0, 1 and 2 respectively. The 4th and 5th critical events correspond to global clock values 7 and 8, respectively. Hence, our algorithm computes two logical schedule intervals for $T1$, $< 0, 2 >$ and $< 7, 8 >$.

3 Tracing Critical Events

In the previous section, we described how to detect thread schedule intervals using a set of local clocks, one for each thread, and a global clock. The global clock is used to order the critical events, by assigning a unique, increasing value to each critical event. Since multiple threads execute critical events and update the same global clock, the following three events must be executed as a single atomic action during the record phase:

1. *AssignGlobalClock* – assigning the global clock value to the critical event;

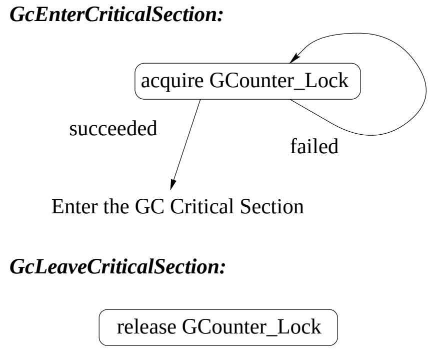


Figure 5: Acquiring and Releasing locks for general critical events

```
GcCriticalSection () {
  GcEnterCriticalSection;
  global_clock_copy = global_clock;
  global_clock++;
  execute the critical event;
  GcLeaveCriticalSection;
}
```

Figure 6: GC-critical section for general critical events

2. *UpdateGlobalClock* – incrementing the global clock; and
3. *CriticalEvent* – execution of a critical event.

We have implemented light-weight *GC-critical section* (for Global Clock critical section) codes to implement a single atomic action of the above three events by guarding them with *GcEnterCriticalSection* and *GcLeaveCriticalSection* as shown in Figure 6. It is used when the critical event is a general event, e.g. a shared variable access, and corresponds to the statements in B5 of Figure 3. *GcEnterCriticalSection* and *GcLeaveCriticalSection* are implemented by acquiring and releasing a light-weight lock called GCounter_Lock as shown in Figure 5.

A straightforward application of the GC-critical section works fine for critical events that are accesses to shared variables, we will call it the *general* GC-critical section. However, GC-critical sections different from the general GC-critical section are used to record most synchronization operations because of their different semantics. The `interrupt` operation can be handled using the general GC-critical section. The rest of this section discusses the GC-critical sections used to record the other Java synchronization operations.

3.1 `monitorenter` and `monitorexit`

Java's `bytecode` has two instructions to help implement critical sections: `monitorenter` and `monitorexit`. Examples of Java's critical sections that can be implemented using these instructions are `synchronized` methods or statement blocks. Reproducing the same order in which threads enter a user-defined critical section is necessary to reproduce the same order in which threads access shared variables within the critical section. A straightforward application of the general GC-critical section code to `monitorenter`, however, would fail, resulting in a deadlock as shown in Figure 7(a).

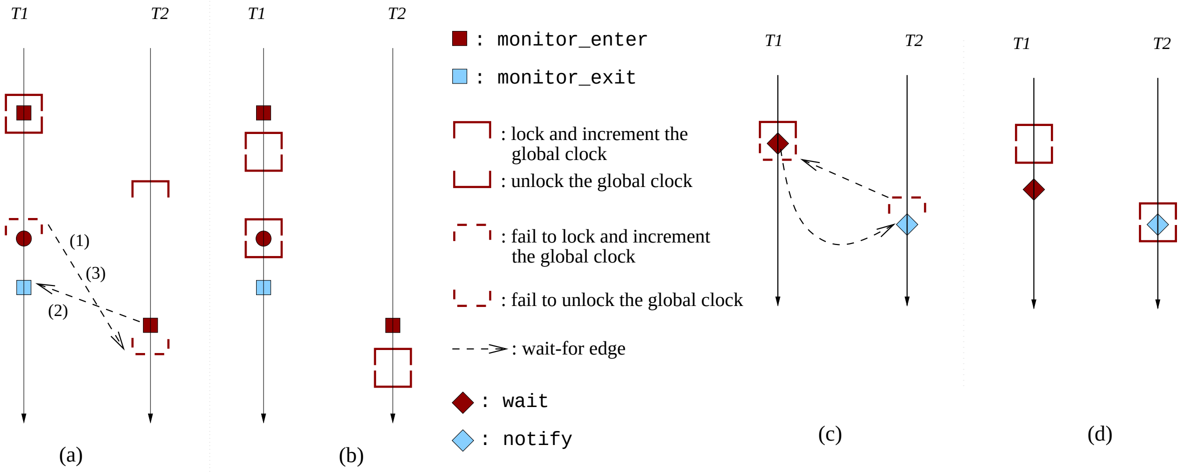


Figure 7: Incorrect and Correct Handlings of **monitorenter** (a, b) and **wait/notify** (c. d)

In the figure, thread $T1$ first enters the user-defined critical section, executing GC-critical section code with **monitorenter** as its critical event. Right after $T1$ enters the critical section but before accessing a shared variable, $T2$ tries to enter the same critical section. In doing so, $T2$ first enters the GC-critical section, then tries to execute its critical event, which in this case is **monitorenter** of the critical section that $T1$ is already in.[4] (We ignore that there must have been a thread switch from $T1$ to $T2$ for $T2$ to try to enter the critical section, and simply describe as if there were multiple processors, one for each thread, which is easier to understand.) $T2$ will have to wait until $T1$ executes **monitorexit** before it can execute its **monitorenter**. $T2$, however, has already entered the GC-critical section, but is blocked from executing **monitorenter**. Now, when $T1$ wants to access the shared variable, it will be blocked from entering the GC-critical section because $T2$ is already in it.

This situation is a classical example of a deadlock as shown with the *wait-for* edges in the figure: (1) $T1$ is waiting for $T2$ to leave the GC-critical section, but $T2$ cannot leave the GC-critical section until it can execute **monitorenter**; (2) $T2$ cannot execute **monitorenter** until $T1$ leaves the GC-critical section via **monitorexit**; (3) $T1$, however, is waiting for $T2$ to leave the GC-critical section, which is identical to (1) above. The solution, as shown in Figure 7(b), is to execute **monitorenter** before GC-critical section, not in the middle of the GC-critical section. Since only one thread can be in a user-defined critical section, we can still correctly capture the order of threads entering the critical section after the execution of **monitorenter**. We still need the GC-critical section to atomically update the global clock. Note that **monitorexit** need not be captured, because capturing the order of successfully executing **monitorenter** is sufficient for replay. Figure 8 shows the GC-critical section for the **monitorenter** critical event.

3.2 wait, notify, and notifyAll

Java provides *event*-style synchronization operations via **wait**, **notify**, and **notifyAll** methods. The language requires that **wait**, **notify** and **notifyAll** methods be invoked within synchronized methods or statements. For example, the code to invoke **wait** would look as follows:

```
synchronized(ev) { ev.wait(); }
```

where **ev** is the object on which the **wait** is invoked. Thus, a thread can successfully invoke a **wait**, **notify**, or **notifyAll** method on an object **ev** only within the critical section where the thread owns **ev**. Hence, when a thread successfully invokes **ev.wait()**, **ev.notify()** or **ev.notifyAll()**, no other thread can own **ev**. Upon (successful) invocation of **ev.wait()**, the thread releases ownership of the object, **ev**, thus allowing another thread to acquire access to **ev** and execute a **notify** or **notifyAll** on **ev**. Any invocation of these operations on an object not owned by the thread can be simply treated as a non-synchronization critical event since it fails to perform as a synchronization operation.[5]

Since **wait** is a blocking event (i.e., the thread executing **wait** suspends and cannot proceed until another thread issues a **notify**), a straightforward application of the general GC-critical section will result in a deadlock. Figure 7(c) illustrates this situation: $T1$ cannot leave the GC-critical section, because it becomes suspended as soon as it executes **wait**. $T2$, which is supposed to execute the **notify** event for $T1$, cannot do that because it cannot enter the GC-critical section, which is currently occupied by $T1$: a deadlock occurs. Recall that a thread ($T1$ in this case) can successfully invoke **wait** on an object only when it already owns the object in a **synchronized** code block, and that another thread

```
GcCriticalSectionMonitor () {
  monitorenter;
  GcEnterCriticalSection;
  global_clock_copy = global_clock;
  global_clock++;
  GcLeaveCriticalSection;
}
```

Figure 8: GC critical section for **monitorenter**

[4] For the simplicity of the discussion, we assume there is only one user-defined critical section.

[5] It throws IllegalMonitorStateException exception, which can be caught and handled by an exception handler.

```
GcCriticalSectionWait () {
  GcEnterCriticalSection;
  global_clock_copy = global_clock;
  global_clock++;
  GcLeaveCriticalSection;
  wait();
}
```

Figure 9: GC critical section for `wait`

```
GcCriticalSectionSelfSuspend () {
  GcEnterCriticalSection;
  global_clock_copy = global_clock;
  global_clock++;
  isAnySelfSuspended = suspending_thread_id;
  suspend(self);
}
```

Figure 10: GC critical section for self-suspend

($T2$ in this case) can invoke `notify/notifyAll` on an object only when it already owns the object in a synchronized code block. Because of this, `wait` need not be inside the GC-critical section to atomically update the global clock; we can place `wait` after the GC-critical section, as shown in Figure 7(d). Figure 9 shows the GC-critical section for `wait`.

When multiple threads are waiting on an object, a `notify` wakes up only one of them in an implementation-dependent way. If the wake-up order is deterministic (a stack or a queue), forcing each thread to invoke `wait` on the same object in the same order during the replay as in the original execution will suffice to wake up the waiting threads in the same order. If the wake-up order is non-deterministic, enforcing the same order to invoke `wait` on the same object is not sufficient, and we use a JVM implementation-dependent approach to make the wake-up order deterministic.

3.3 `suspend` and `resume`

In Java, a thread can suspend itself or others via `suspend`. A thread suspended this way resumes execution when another thread executes `resume` on the suspended thread. `suspend(other)`, which suspends another thread, can be handled with the general GC-critical section code in Figure 6. Resuming other threads can also be handled with the general GC-critical section. The rest of this section discusses handling a self-suspend operation, i.e., a `suspend(self)` operation.[6]

A thread suspending itself introduces a problem similar to, but larger than, the one by `wait`. The problem is similar in that the execution of the thread suspends after the invocation of the operation. The difference is that `suspend(self)` need not be in a synchronized block as `wait` does. In order to allow a thread to suspend itself within the GC-critical section while holding the GCounter_Lock, another thread must execute the *GcLeaveCriticalSection* on behalf of the self-suspended thread, i.e., release the GCounter_Lock owned by the self-suspended thread. The self-suspending thread sets a flag, *isAnySelfSuspend*, with a non-zero-valued thread id of the self-suspending thread. The flag lets other threads know that the GCounter_Lock is currently held by the self-suspending thread so that they can release the GCounter_Lock on behalf of it. A special-valued thread id (such as zero) can be used to reset the flag. Figure 10 shows the GC-critical section for `suspend(self)`. Notice the absence of *GcLeaveCriticalSection* at the end of the critical section.

We also change the original *GcEnterCriticalSection* in Figure 5 into the new one in Figure 11(a) for all the GC-critical sections in order to handle self-suspend. The difference is that, in the latter, when a thread fails to acquire the GCounter_Lock and finds *isAnySelfSuspend* to be set, it tries to release the GCounter_Lock held by the self-suspending thread by executing *tryUnlockSelfSuspended*. *tryUnlockSelfSuspended*, in Figure 11(b), itself is a light-weighted critical section.

Once in *tryUnlockSelfSuspended*, the thread examines the *isAnySelfSuspended* flag again to guard against the case another thread has reset it before the current thread enters the critical section. If the flag is still set, it checks whether the thread that set the flag is indeed in a suspended state (B1 in the figure). If not, the thread simply exits the critical section, waiting for the self-suspending thread to become suspended. If the self-suspending thread is in a suspended state, the thread resets the *isAnySelfSuspended* flag and releases the GCounter_Lock, effectively executing *GcLeaveCriticalSection* on behalf of the self-suspended thread. The thread then exits the critical section and tries to grab the GCounter_Lock again. Determining whether the self-suspending thread is indeed in a suspended state can be done easily if there exists a system call that returns the state of a thread. In that case, the *isAnySelfSuspend*, which is the id of the self-suspending thread can be used to query the state of the self-suspending thread.

When there exists no such system call, we force the other threads to wait another round of thread scheduling to give the self-suspended thread enough time to successfully suspend itself after setting the flag. More specifically, we use a counter, instead of the thread id, as the *isAnySelfSuspend* flag set by a self-suspending thread. The counter, initially zero, is incremented at each self-suspend operation. After the counter is reset to zero in *tryUnlockSelfSuspend*, it starts from the previous counter value, not from zero. Note that this is differnt from Figure 10, where *isAnySelfSuspend* is the id of the self-suspending thread.

After thread $T1$ sets the *isAnySelfSuspend* flag, the statement B1 in Figure 11(b) by another thread $T2$ becomes simply comparing the new *isAnySelfSuspended* flag value with the previous one $T2$ saw before. If they are identical, $T2$ assumes $T1$ has had enough time to suspend itself. If they are different, $T2$ assumes that $T1$ has not had time to self-suspend, remembers the new flag (counter) value, and yields the thread schedule. The assumption behind this scheme is that the thread scheduling is fair: an active thread gets a thread-schedule slot of non-trivial amount of time before any other thread with no higher priority gets two of such thread-schedule slots. (Only threads with priorities no higher than that of the self-suspended thread can participate in unlocking the GCounter_Lock on behalf of the self-suspended thread.) To ensure that there exists at least one thread with priority no higher than that of the self-suspended thread, DejaVu uses a JVM thread that always runs at a lower pri-

[6] The notations `suspend(self)` and `suspend(other)` are used here only for convenience. The real Java syntax for them are `suspend()` and `other.suspend()`, respectively

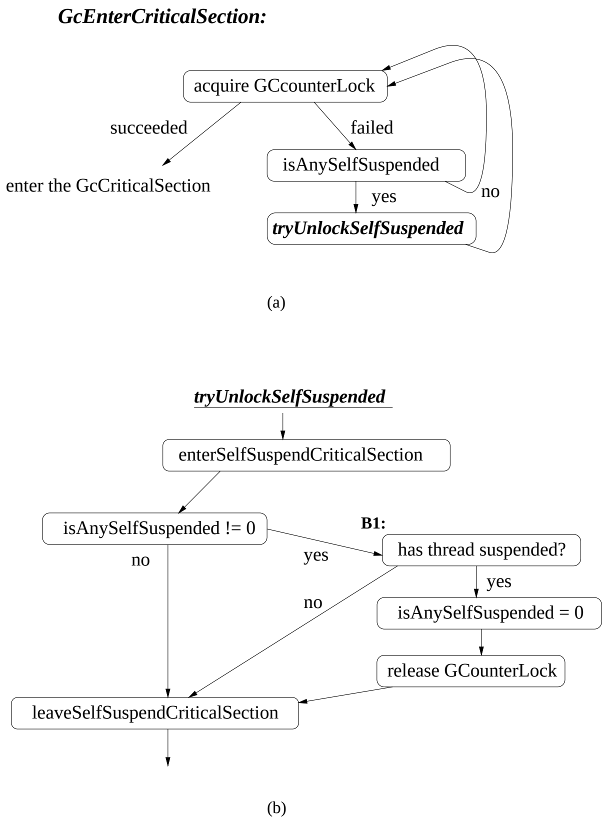


Figure 11: Modified GcEnterCriticalSection

ority than any other threads.

We use Figure 12 to illustrate how the scheme works. In the example above, we assume all the threads have the same priority. In Figure 12(a), $T1$ sets the *isAnySelfSuspended* flag with a new value. After this, all the other threads, during the same round of the thread scheduling, see this new flag value and yield the thread schedule, assuming $T1$ has not had enough time to self-suspend. At the next round of the thread scheduling, $T2$ sees the same flag value again and assumes $T1$ has had enough time to self-suspend: it releases the GCounter_Lock on behalf of $T1$ and resets the flag. $T3$ now grabs the lock and executes its critical event.

In Figure 12(b), the events are the same up to $T3$'s grabbing the lock and executing its critical event during the second round of the scheduling. $T3$'s critical event, however, is another self suspend, and $T3$ sets the *isAnySelfSuspended* flag with a new value. After that, threads $T4$ through TN during this round fail to release the GCounter_Lock on behalf of $T3$ because the flag value they see is different from what they saw before. During the third round, $T2$ sees a different value of the flag and fails to release the GCounter_Lock. $T4$, however, sees the same flag value second time and releases the GCounter_Lock. $T5$ then grabs the lock and executes its critical event.

3.4 Interactions between `wait` and `suspend`

Assume a thread $T1$ is about to **`wait`** on an object and has just finished the GC-critical section for it, as described in Section 3.2. Also assume this is followed by a thread $T2$ suspending $T1$. There are three possible timings:

1. $T2$ suspends $T1$ before $T1$ successfully invokes **`wait`**;

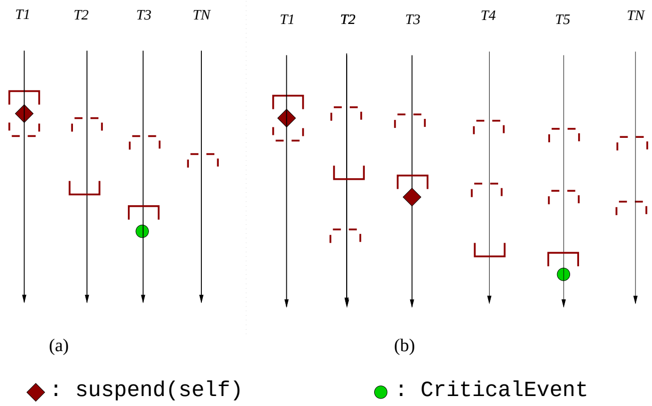


Figure 12: Correct Handlings of **`suspend(self)`**

2. $T2$ suspends $T1$ after $T1$ successfully invokes **`wait`**; and

3. $T2$ suspends $T1$ after $T1$ successfully invokes **`wait`** *and* a third thread wakes up $T1$ (via **`notify`** or **`interrupt`**).

Case 3 does not introduce any interference between **`wait`** and **`suspend`**, and there is no need for any special treatment in this case. Probably this is what the user intended.

Case 1 and Case 2 are the same as far as $T1$ is concerned: $T1$ becomes inactive. Also, there is no difference in terms of the global clock values for the **`wait`** and the **`suspend`**: the clock value of the **`wait`** precedes that of the **`suspend`** since, in this scenario, the GC-critical section for **`wait`** executes before the GC-critical section for **`suspend`**. However, there is a difference between the two in terms of their impact on other threads. In Case 1, $T1$ still holds all the locks it owns when it becomes inactive (from **`suspend`** executed by $T2$), preventing other threads waiting for any of the locks from proceeding any further. In Case 2, $T1$ releases all the locks it owns when it becomes inactive (when it executes **`wait`**), allowing other threads waiting for the locks to proceed.

The implication of the difference between Case 1 and Case 2 on DejaVu is that DejaVu could produce two different execution behaviors in the replay mode, one corresponding to Case 1 and the other corresponding to Case 2. The solution to handle such an interaction between **`wait`** and **`suspend`** operations is similar to the solution for self-suspend. The *waiting* thread that is to execute **`wait`** sets an *isWaiting* flag just before it executes **`wait`**. Then, the *suspending* thread that is about to suspend the waiting thread checks the *isWaiting* flag of the thread to be suspended. If the *isWaiting* flag is set, the suspending thread checks if the waiting thread is in the **`wait`** state. If it is in the **`wait`** state, the suspending thread safely executes the **`suspend`** operation on the waiting thread. If it is not in the **`wait`** state, the suspending thread waits until it is in the **`wait`** state. The *isWaiting* flag is reset by the suspending thread immediately before it suspends the waiting thread or is reset by the waiting thread when it wakes up from its **`wait`** state. Note that this approach essentially enforces Case 2 when either Case 1 or Case 2 is possible.

When there is no system call to determine whether a thread is in the **`wait`** state or not, the suspending thread waits until it sees the same *isWaiting* flag value twice before suspending the waiting thread. Note that in the self-suspend case described in Section 3.3, all the threads with priorities not higher than the self-suspending thread get involved in releasing the GCounter_Lock. In the case of `wait` and `suspend`, only the suspending threads is involved, and the priority of the suspending thread gets temporarily lowered to that of the waiting thread if the suspending thread's priority is higher than that of the waiting thread's priority. This is necessary to ensure that the waiting thread will have enough time to successfully execute `wait`. The priority will be reset to the original value immediately before the suspending thread suspends the waiting thread or when it finds the *isWaiting* flag to be zero again.[7]

4 Replaying Execution Behavior

At the start of a replay mode, DejaVu reads the thread schedule information from a file created at the end of the record mode. When a thread is created and starts its execution, it receives from DejaVu an ordered list of its logical thread schedule intervals. We use the *finite state automaton* in Figure 13 to describe how each thread executes and reproduces the same execution behavior using this ordered list of schedule intervals.

In the figure, state `S0` is the initial state for each thread. At state `S1`, it reads the next schedule interval to update the new *FirstCriticalEvent* and *LastCriticalEvent*, which are represented by global clock values. After that, the thread waits, at `S2`, until the global clock value becomes the same as the current *FirstCriticalEvent*, at which point it moves to `S3`.

`S2` corresponds to *GcEnterCriticalSection* of GC-critical section described in Section 3, and we call it *RpEnterCriticalSection*. There are two major differences between *GcEnterCriticalSection* and *RpEnterCriticalSection*: (1) the former is implemented by acquiring a lock (GCounter_Lock), while the latter is implemented by comparing the global clock with the *FirstCriticalEvent* of the current interval and yielding the thread schedule via `sleep`; and (2) the former need be executed for every critical event during record, while the latter need be executed only for the first event of each interval during replay.

At `S3`, the thread executes the next instruction (event). If the event is a critical event, the thread also increments the global clock at `S3`. At the end of `S3`, the thread executes *RpLeaveCriticalSection*, corresponding to *GcLeaveCriticalSection* of GC-critical section. Since *RpEnterCriticalSection* does not acquire a lock, *RpLeaveCriticalSection* does not release any lock. *RpLeaveCriticalSection*, however, checks whether the global clock value becomes greater than *LastCriticalEvent* of the current interval, at which point it moves to `S4`. If the value is not greater than *LastCriticalEvent*, it repeats `S3`. At `S4`, the thread increments the current interval number and checks if there are any more intervals left. If so, it moves back to state `S1`, and repeats the above steps. If no more intervals are left, it terminates.

We call the following combined steps of `S2` and `S3` the *RP-critical section*:

1. waiting for the global clock value to become the same as the current *FirstCriticalEvent* (`S2`);
2. executing the critical event and incrementing the global clock (`S3`); and
3. checking whether the global clock value becomes greater than *LastCriticalEvent* of the current interval (`S3`).

As stated before, the first step of RP-critical section need be executed only for the first event of each interval.

In the presence of higher-priority threads waiting for their turn at state `S2`, a thread with a lower priority can starve without being able to execute and increment the global clock. This in turn will make the higher-priority threads waiting forever at `S2`, hanging the whole system. The solution we adopt for this is to progressively increase the sleep time after comparing the global clock with the *FirstCriticalEvent* in *RpEnterCriticalSection*. The sleep time is reset to the original value when the thread succeeds in entering *RpEnterCriticalSection*.

The above description applies to all non-synchronization critical events. In addition, the RP-critical section can be applied without modification to `monitorenter`: the thread waits for the correct global clock value before executing `monitorenter`, and increments the global clock after the thread successfully executes `monitorenter`.

Handling `wait`, `notify`, `notifyAll`, `suspend`, `resume`, and `interrupt` synchronization operations during replay is similar to their handling during the record mode. The differences relate to the differences between the GC-critical section and the RP-critical section: waiting for and updating the global clock value are the means of ordering events during replay, not acquiring and releasing the GCounter_Lock as done during record.

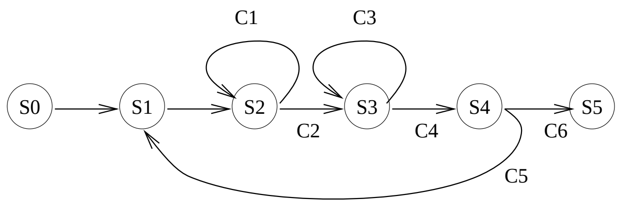
5 Implementation and Performance Results

We have implemented the record/replay mechanism discussed in the previous sections by modifying the Sun Microsystems' Java Virtual Machine (JVM). The modified JVM, called DejaVu, is capable of deterministic replay of Java multithreaded programs, and can be used to implement several tools. Using DejaVu, we have implemented a simple command line debugger that uses execution replay for debugging. We considered modifying the application bytecode, instead of the VM that interprets the application bytecode, but decided against it because we felt the overhead would be a lot higher than modifying the VM. The particular java interpreter we have modified is `java_g`. `java_g` is written mostly in C and is a lot easier to instrument than the regular `java` interpreter. It also has much better support for debugging the application.

Table 1 shows the performance measurements of DejaVu with four synthetic-workload applications – Chaos, MM, SOR, and MTD – and two SPLASH-2 (Stanford Parallel Applications for Shared Memory) kernel benchmark programs [14] – Radix and FFT – on a machine with a Pentium 200 MHz processor running Windows NT. Chaos is an extreme case of a highly non-deterministic, compute-intensive program of multithreads that, without synchronizations, increment two shared variables over and over. MM is a matrix multiply program, and SOR implements a Successive Over Relaxation algorithm for a financial application. Both MM and SOR are highly compute intensive with only a few synchronization operations. MTD is a thread schedule simulator consisting mostly of synchronization operations.

The high instrumented-execution overhead of Chaos, MM, and SOR reflects their compute-intensiveness with high rate of shared-variable accesses: each access of a shared variable is a critical event that incurs the overhead of atomically updating the global clock with the (light-weighted) synchro-

[7] This can theoretically have an unpleasant effect of starving the suspending thread if the priorities of all the other threads get raised above the suspending thread's new priority and remain there. However, this should be rare in practice.



S0: START, i = 0
S1: update FirstCriticalEvent(i) and LastCriticalEvent(i)
S2: yield the thread schedule
S3: execute event.
if CriticalEvent increment global_clock
S4: i = i + 1
S5: END

C1: global_clock < FirstCriticalEvent(i)
C2: not C1
C3: global_clock <= LastCriticalEvent(i)
C4: not C3
C5: i <= last interval
C6: not C5

Figure 13: Replay Finite State Automaton

nization operations of DejaVu. However, their instrumented-execution time is still well within twice the original execution time. Their trace sizes are also small, in spite of their large number of critical events, due to the small number of thread switch intervals. The smaller replay-execution overhead of these three programs also reflects the fact that shared-variable accesses during a replay execution do not require critical events to update the global clock as they do during an instrumented execution: shared-variable accesses are cheaper during replay execution.

MTD is the opposite of Chaos: MTD has almost no shared-variable accesses and consists mostly of Java synchronization operations, which are expensive by themselves with or without any instrumentation by DejaVu. Since the light-weight critical sections of DejaVu have negligible cost compared with the Java synchronizations, MTD does not show any detectable overhead in instrumented execution or in replay execution. We suspect a similar behavior with interactive graphics-oriented Java programs using AWT (Abstract Window Toolkit).

Radix and FFT of SPLASH-2 kernel programs are written in C, which we have ported into Java. Russinovich and Cogswell also use the same programs (and three additional programs from the same benchmark suite) in the performance measurements for their *Repeatable Scheduling* [12]. However, it is hard to compare the performance of DejaVu with that of Repeatable Scheduling solely based on the measurements with these benchmark programs, because DejaVu's measurements are for Java programs interpreted by JVM running on Windows NT while Repeatable Scheduling's measurements are for C programs compiled into native code running on Mach 3.0/UX. The overhead of Repeatable Scheduling range between 5.7% and 16.5% for instrumented execution, and between 8.7% and 15.3% for replay execution. The space overhead of Repeatable Scheduling ranges between 6KB and 20KB. The execution and space overheads, however, will depend on the size of the problem, i.e. input parameters such as the number of threads, which is not shown in their publication.

As stated above, DejaVu is implemented by modifying the `java_g` interpreter from Sun, which is mostly written in C and, therefore, is easier to modify than the regular `java` interpreter. Written in C, `java_g` is itself a few times slower than the regular `java` interpreter in executing the application code. It will be interesting to see the new performance results when DejaVu is ported to the regular `java` interpreter with the instrumentation code written in the assembly language.

DejaVu can also work with a DejaVu-*aware* compilation system, including just-in-time (JIT) compilers for Java. The DejaVu-aware compiler, whether a JIT for Java or a full compiler for a conventional language like C, needs to generate native code corresponding to the instrumentation code for the critical events in DejaVu. A JIT compiler improves the performance of the java interpreter, and will also improve the performance of the DejaVu instrumentation for the JIT-compiled part of the application. Although we suspect the relative performance of DejaVu will in general become worse with JIT compilation, it is hard to predict whether that will be the case, or how much.

Java Native Interface (JNI) allows native methods to work with application bytecode. DejaVu can replay the order in which the native methods are invoked from within the bytecode. However, deterministic replay of the native methods with non-deterministic side effects is beyond the capability of DejaVu unless the native methods are also compiled by a DejaVu-aware compiler.

When applied to multiprocessor systems, DejaVu's overhead will increase because there will be in general more than one thread simultaneously active at any given time. These simultaneously active threads will make the size of each schedule interval smaller, resulting in larger number of total schedule intervals for the same application. Each schedule interval incurs time overhead in accessing the GC-critical section and space overhead in tracing the interval. Also, all the critical events will be totally serialized in terms of the global clock, resulting in loss of parallelism in execution. We are currently extending DejaVu so that there will be multiple global clocks to reduce this loss of parallelism.

6 Related Work

Repeated re-execution is a widely accepted technique for debugging deterministic sequential applications. Repeated re-execution for debugging, however, fails to work for non-deterministic multithreaded applications because a bug that manifested itself during one execution instance might not manifest itself again in another execution instance.

One approach to debug non-deterministic applications is to avoid repeated re-execution and generate traces as is done by PPD [1]. Flowback analysis can then be performed on

Application		Number of Threads	Native Execution (sec)	Instrumented Execution		Replay Execution		Trace Size	Critical Events	Thread Switch Intervals
				Time (sec)	Ovrhd	Time (sec)	Ovrhd			
Synthetic Work Load	Chaos	16	11.6	21.7	87.6%	18.3	57.8%	4KB	10M	0.5K
		32	22.6	42.5	88.1%	35.5	57.1%	8KB	19M	1K
		64	47.6	84.5	77.5%	71.7	50.6%	16KB	39M	2K
	MM	4	26.8	43.9	63.8%	35.8	33.6%	4KB	15M	0.5K
		16	26.6	44.1	65.8%	35.9	35.0%	4KB	15M	0.5K
		64	26.8	44.5	66.0%	35.8	33.6%	5KB	15M	0.7K
	SOR	4	5.1	8.4	64.7%	6.8	33.3%	1.3KB	2.3M	0.2K
		16	5.1	8.5	66.6%	7.0	37.3%	4KB	2.3M	0.5K
		64	5.5	8.8	60.0%	7.4	34.5%	8KB	2.4M	2K
	MTD	4	20.1	20.1		20.1		0.5KB	3K	0.1K
		16	20.1	20.1		20.1		1KB	5K	0.1K
		64	20.1	20.1		20.1		2KB	12K	0.3K
SPLASH Kernels	Radix	4	53.7	77.6	44.5%	74.5	38.7%	16KB	2.3M	2K
		16	56.7	81.4	43.6%	76.1	34.2%	17KB	2.6M	2.3K
		64	66.8	104.7	56.7%	99.8	49.4%	36KB	40M	4.5K
	FFT	16	4.98	5.95	19.5%	6.55	31.5%	6KB	1M	0.5K
		32	5.15	6.18	20.0%	8.52	65.4%	9KB	1M	1K
		64	5.82	6.81	17.0%	8.04	38.1%	18KB	1M	2.4K

Table 1: Performance of DejaVu Replay System

the traces to show the causality of the program events to the user to help identify the bugs.

Re-execution approaches for debugging a non-deterministic application need to generate sufficient traces to reproduce the same execution behavior over and over. Most previous approaches for re-execution of non-deterministic applications have focused on replaying multiprocess applications running on shared memory multiprocessor systems. Like threads, processes of an application can affect the execution behavior of other processes via accesses to shared variables, synchronization operations, or communications.[8] Replaying multiprocess applications requires capturing interactions among processes – i.e., critical events – and generating traces for them. The major drawback of these approaches is the potentially large overhead (in time, and particularly in space [8, 13, 11]) in generating the traces.

To reduce the trace size, *Instant Replay* [8] assumes that applications use a correct, coarse-grained operation, called *CREW* for concurrent-read-exclusive-write, to access shared objects, and generates traces only for these coarse operations. However, this approach fails if critical events within CREW are non-deterministic. The approach by Carver and Tai [13] is similar in that they also generate traces only for coarse-grained critical events, assuming shared variables are well guarded within well-defined critical-sections.

Russinovich and Cogswell's approach [12] differs from the above approaches in that it addresses specifically multithreaded applications running only on a uniprocessor system. To capture the physical thread scheduling information, Russinovich and Cogswell have modified the Mach operating system so that it notifies the replay system of each thread switch. This makes their approach highly dependent on an operating system and also on the availability of the source code of the operating system, which is unlikely for commercial third-party operating systems. Their approach is also strictly for uniprocessor systems and does not work for multiprocessor systems.

Holloman and Mauney's approach [7, 6] is similar to Russinovich and Cogswell's except for the mechanism to capture the process scheduling information. Their approach uses exception handlers instrumented into the application code that capture all the exceptions, including the ones for process scheduling, sent from the UNIX operating system to the application processs. Their approach, therefore, does not require modifying the operating system. Their approach is still operating-system dependent and strictly for uniprocessor systems.

Our approach is similar to Russinovich and Cogswell's in that we also generate traces only for thread switches, resulting in trace sizes similar to theirs. Our approach, however, captures the logical thread schedule interval. Our approach does not require making modifications to the operating system, and is, therefore, independent of the operating system. Also, our approach works on multiprocessor systems whose thread schedule is fair according to the assumption in Section 3. One disadvantage of being independent of the underlying system is that capturing logical thread schedule information can potentially incur higher execution-time cost than getting notified of physical thread schedule information by the operating system.

Netzer's *Optimal Tracing* [10] reduces the trace size further by applying an execution-time algorithm to find the minimum traces sufficient to replay the execution. Optimal Tracing can reduce the trace size by one or two orders of magnitude, but potentially at the cost of substantially increasing the execution time of the application.

Our compact logging scheme for logical thread schedules is similar to the one described by Levrouw et. al. for event logging [9]. A major difference is that our scheme uses a single global clock while theirs uses one clock for each shared object. This difference makes our approach much simpler and more efficient than theirs on a uniprocessor system. They describe an extension of their scheme to reduce the log size further, but at the cost of counting the number of instructions executed between critical events to simulate a real time clock.

[8] Strictly speaking, threads are different from processes in that threads usually share address spaces while processes do not. However, the issues and techniques described in this paper apply to both.

DejaVu, to our knowledge, is the first tool that completely addresses the issues in handling all the Java synchronization operations in the context of deterministic replay of multithreaded Java applications.

7 Conclusions

We have developed a record/replay tool for Java applications, called DejaVu that provides a deterministic replay of a non-deterministic execution. DejaVu is is implemented by modifying Sun Microsystem's Java Virtual Machine (JVM). DejaVu is independent of the underlying thread scheduler such as the operating system. It runs highly efficiently on a uniprocessor system and can be used on a multiprocessor system as well, with higher overhead than on a uniprocessor system expected. While our current implementation of DejaVu is in the context of a Java interpreter, the record/replay mechanisms described in this paper will work just as well in the context of both traditional compilation environments that generate object code, and just-in-time compilers, for general multithreaded programming systems.

Acknowledgements

We thank John Barton, Susan Flynn Hummel, Ravi Konuru and Peter Sweeney for their comments on an earlier draft of the paper which greatly helped improve the paper. We also thank Chris Holt and Monica Lam for their help with the SPLASH kernel programs.

References

[1] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[2] Jong-Deok Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. *Proc. of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[3] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. Technical report, IBM, 1998. In Preparation.

[4] Jong-Deok Choi and Janice M. Stone. Balancing runtime and replay costs in a trace-and-replay system. *Conference Record of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[5] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Seattle, Washington, March 1990. ACM Press.

[6] Edward Dean Holloman. Design and implementation of a replay debugger for parallel programs on unix-based systems. *Master's Thesis, Computer Science Department, North Carolina State University*, June 1989.

[7] Edward Dean Holloman and Jon Mauney. Reproducing multiprocess executions on a uniprocessor. *Unpublished paper*, August 1989.

[8] Thomas J. Leblanc and John M. Mellor-Crummy. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[9] L.J. Levrouw, K.M.R. Audenaert, and J.M. Van Campenhout. Execution replay with compact logs for shared-memory systems. *Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing, IFIP Transactions A-44*, pages 125–134, April 1994.

[10] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging (Also available as ACM SIGPLAN Notices Vol. 28, No. 12)*, pages 1–11, May 1993.

[11] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. *Proceedings of SIGPLAN/SIGOPS Workshop on*, pages 124–129, May 1988.

[12] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. *Proceedings of ACM SIGPLAN Conference on Programming Languages and Implementation (PLDI)*, pages 258–266, May 1996.

[13] K. C. Tai, Richard H. Carver, and Evelyn E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.

[14] S. C. Woo, M. Oharai, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.