RC 22133 (W0107-026) 30 July 2001 Computer Science

# IBM Research Report

## A Debugging Platform for Java Server Applications

**Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, John Vlissides, Hyun-Gyoo Yook**

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

# A Debugging Platform for Java Server Applications

Bowen Alpern Jong-Deok Choi Ton Ngo Manu Sridharan[*] John Vlissides
Hyun-Gyoo Yook

IBM T. J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598
{jdchoi, alpernb, ton}@us.ibm.com, msridhar@mit.edu, vlis@us.ibm.com, hyun@watson.ibm.com

**Abstract**

Development of multithreaded server applications is particularly tricky because of their nondeterministic execution behavior, availability requirements, and extended running times. New tools are needed to help programmers understand server behavior.

DejaVu supports deterministic replay of nondeterministic, multithreaded Java programs running on the Jalapeño virtual machine on uniprocessors. Jalapeño is written in Java, and its optimizing compiler combines application, virtual machine, and DejaVu instrumentation code into unified machine-code sequences. Such integration boosts performance, but it also compounds the difficulty of replaying nondeterministic behavior accurately and with minimal interference from the instrumentation. DejaVu ensures deterministic replay through *symmetric instrumentation*—side-effect-preserving instrumentation in both record and replay modes—and *remote reflection*, which exposes the state of an application without perturbing it.

## 1 Introduction

Software development tooling has matured to the point that any programming environment will provide a debugger that can single-step, set breakpoints, inspect values, and evaluate expressions. Multithreading support is less common. Still, many current tools add straightforward extensions that permit independent control and inspection of threads. But even the most carefully designed and implemented multithreaded program can behave nondeterministically; that is, successive runs with the same input data may nonetheless produce different outputs. Nondeterminism makes errors difficult to reproduce, greatly complicating the hunt for bugs.

Compounding this problem are two technology trends:

1. Software is increasingly dynamic, with more and more configuration, translation, linking, and optimization being performed at run-time. Dynamism is the enemy of high performance.

2. Distributed systems often incorporate multithreaded middleware components. Nondeterminism may arise within the component, in its interactions with other components, or both. The errors that result can elude unit testing, surfacing only under production conditions. It may not be feasible to halt a large production system to debug such errors, and yet they might not be reproducible on a smaller scale.

These difficulties are not independent; addressing one can exacerbate the other. For example, Jalapeño [2] is a Java virtual machine (JVM) designed for high-performance servers. Written in Java, Jalapeño uses a compilation-only model to achieve dynamism *and* high performance by compiling bytecode to machine code at run-time. Unfortunately,

[*]Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

two aspects of Jalapeño's compilation model can make program behavior even less predictable than usual: *cross-optimization* and *dynamic optimization*.

Cross-optimization improves performance by analyzing and optimizing the application and its run-time system together. Just as interprocedural analysis yields benefits beyond what can be achieved with purely local optimizations, "co-analysis" and "co-optimization" of the application and run-time environment can expose many new opportunities for optimization. But because cross-optimization blurs the distinction between application code and system code, it may be more difficult to isolate problems in the application.

Meanwhile, dynamic optimization improves performance by recouping the time cost of optimization through speedup of the optimized code. This trade-off is driven by dynamic profiling; hence the optimizations performed and the application code produced can vary with workload. As a result, a long-running application may get optimized in a way that's difficult to reproduce.

A *replay* tool can address these problems by reproducing a program's execution behavior at will. Such a tool must record execution behavior, which requires instrumenting the program. Because instrumentation may impact performance, the tool should be able to run the program without instrumentation. We distinguish three modes of program execution: *uninstrumented*, *record*, and *replay*.

To be at all useful, a replay tool must be *accurate*: the behavior in replay mode must correspond exactly to record mode's (although the two modes need not perform identically). An additional criterion for a replay tool is its *precision*—how well the behavior and performance of the system in record mode match uninstrumented mode. The accuracy requirement is absolute; precision is relative.

This paper describes *DejaVu*, a replay tool for multithreaded Java applications running on the Jalapeño JVM on uniprocessors.[1] DejaVu (*De*terministic *Jav*a Replay *U*tility) provides a replay capability on which to build higher-level tools for debugging and analyzing multithreaded applications. Jalapeño allows for high performance, while DejaVu is designed to eliminate the complications that arise from multithreading and from Jalapeño itself.

## 2 An Example of Debugging using Deterministic Replay

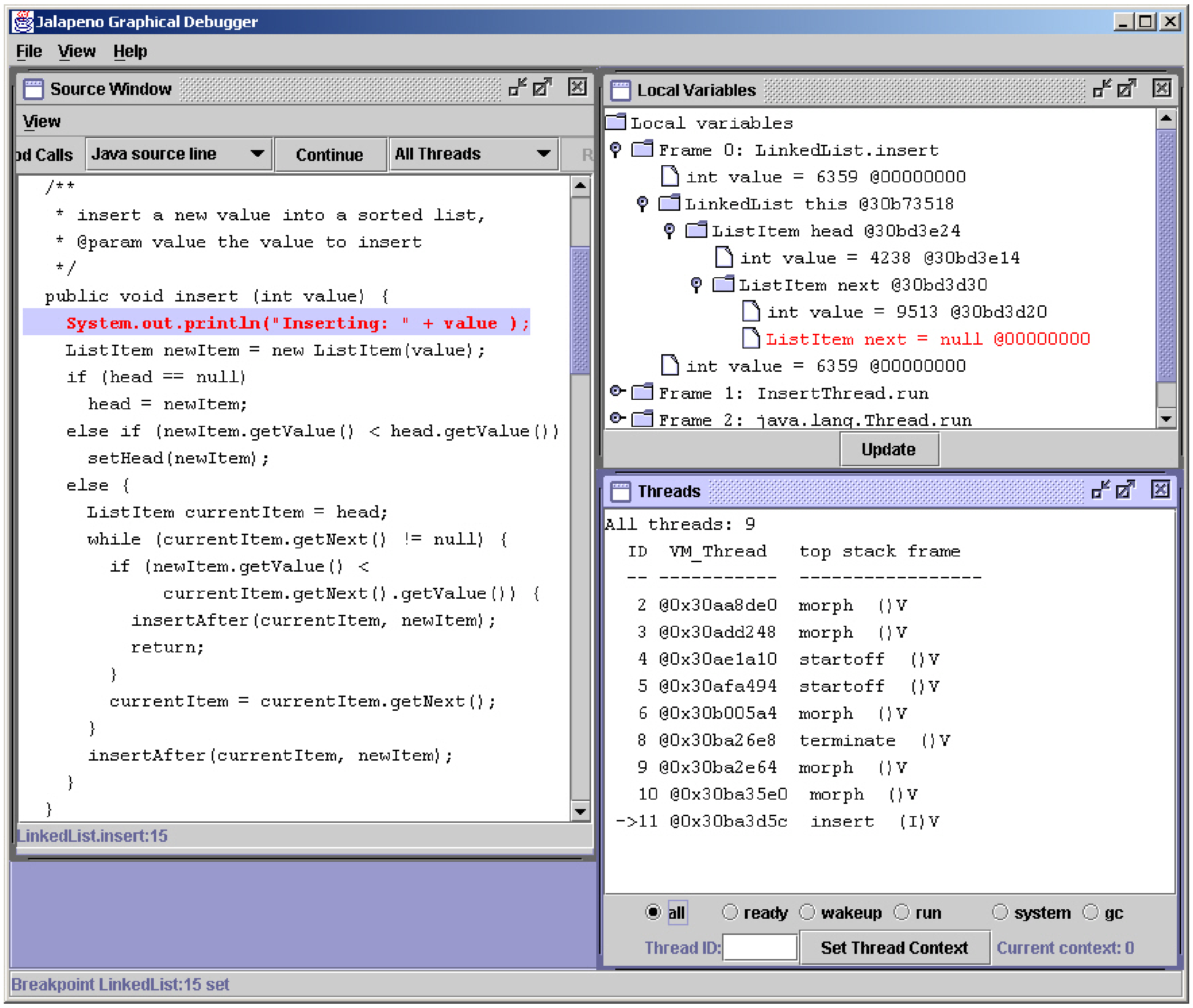
To demonstrate the value of deterministic replay for debugging multithreaded Java programs, consider a simple example in which five threads each attempt to add a single number to a sorted linked list. The list's contents is then printed. The insertion order is random, depending on the interleaving of the threads.

With inadequate synchronization, these insertions may interfere with each other and produce incorrect behavior. Suppose that on a given execution the insertion order is $9513$, $4238$, $7449$, $6353$, and $1127$, but the result is that only four of the five numbers are printed—$1127$, $4238$, $7449$, and $9513$. The fourth number, $6359$, has been lost. The DejaVu replaying debugger will be used *on this execution* to discover the cause.

We start by setting a breakpoint at the beginning of the `insert` method. Figure 1 shows the screen shot of the Jalapeño graphical debugger when the breakpoint is hit the fourth time. As expected, $6353$ is about to be inserted. Recall that this is the value that did *not* appear in the output. Note also that $4238$ and $9513$ are on the list, but $7449$ is not.[2]

[1]Replay of *multiprocessor* executions is a considerably harder problem that we hope to address in the future (see Section 7). Nonetheless, we claim that a uniprocessor replay engine is useful in understanding and debugging multithreaded programs even if they are meant to run on multiprocessors.

[2]The thread that is inserting $7449$ has been artificially delayed. The disastrous consequences of this delay will soon be apparent. Although this delay was induced, a JVM's thread scheduler could have produced it just as well.



```
/**
 * insert a new value into a sorted list,
 * @param value the value to insert
 */
public void insert (int value) {
  System.out.println("Inserting: " + value );
  ListItem newItem = new ListItem(value);
  if (head == null)
    head = newItem;
  else if (newItem.getValue() < head.getValue())
    setHead(newItem);
  else {
    ListItem currentItem = head;
    while (currentItem.getNext() != null) {
      if (newItem.getValue() <
            currentItem.getNext().getValue()) {
        insertAfter(currentItem, newItem);
        return;
      }
      currentItem = currentItem.getNext();
    }
    insertAfter(currentItem, newItem);
  }
}
```

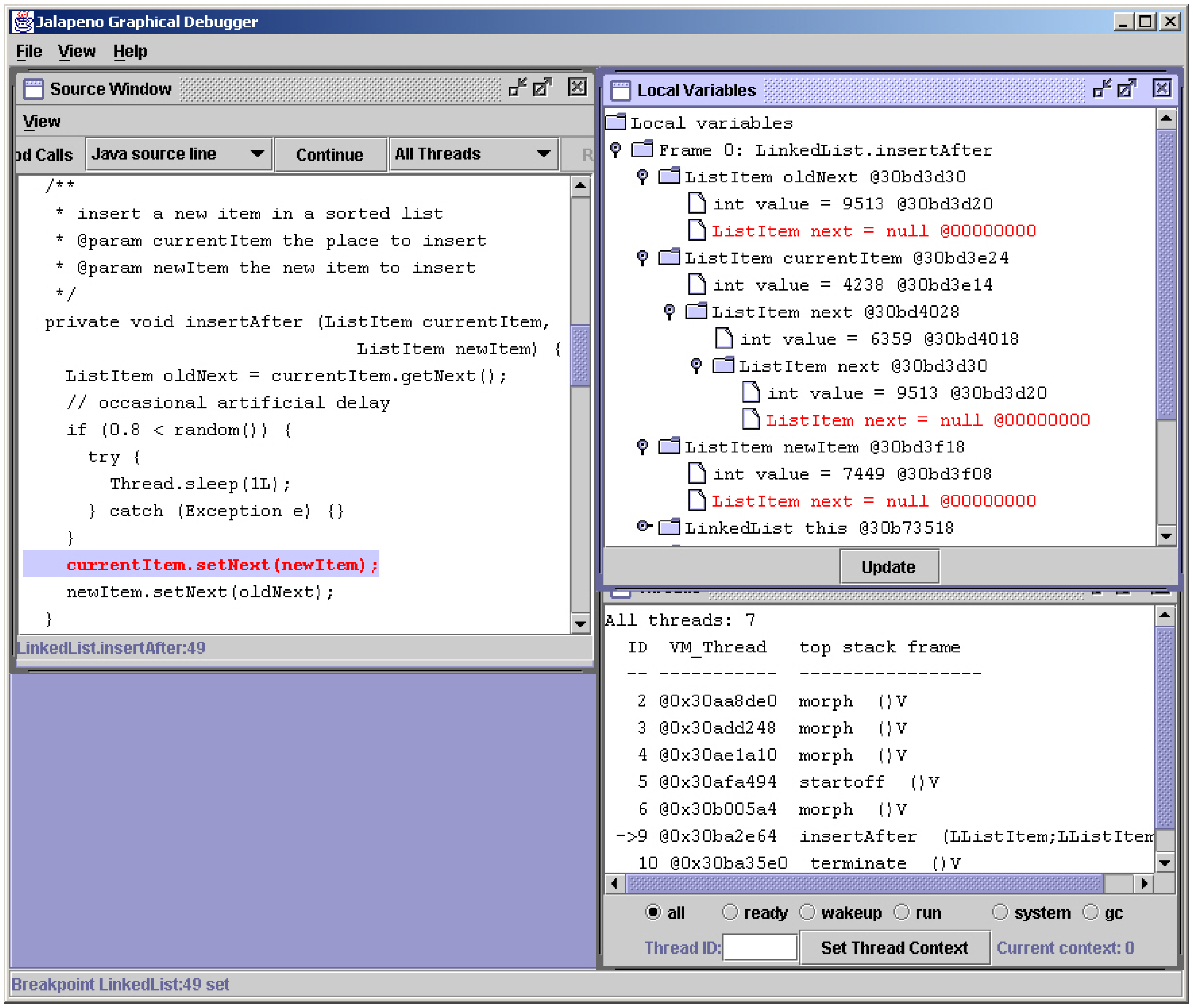
LinkedList.insert:15

```
Local variables
  Frame 0: LinkedList.insert
      int value = 6359 @00000000
    LinkedList this @30b73518
      ListItem head @30bd3e24
          int value = 4238 @30bd3e14
        ListItem next @30bd3d30
            int value = 9513 @30bd3d20
            ListItem next = null @00000000
      int value = 6359 @00000000
  Frame 1: InsertThread.run
  Frame 2: java.lang.Thread.run
```

```
All threads: 9
   ID  VM_Thread    top stack frame
   -- -----------  -----------------
    2 @0x30aa8de0  morph  ()V
    3 @0x30add248  morph  ()V
    4 @0x30ae1a10  startoff  ()V
    5 @0x30afa494  startoff  ()V
    6 @0x30b005a4  morph  ()V
    8 @0x30ba26e8  terminate  ()V
    9 @0x30ba2e64  morph  ()V
   10 @0x30ba35e0  morph  ()V
 ->11 @0x30ba3d5c  insert  (I)V
```

Breakpoint LinkedList:15 set

Figure 1: Thread 11 at the breakpoint about to add 6359 to the list. Where is 7449?



```
 /**
  * insert a new item in a sorted list
  * @param currentItem the place to insert
  * @param newItem the new item to insert
  */
 private void insertAfter (ListItem currentItem,
                           ListItem newItem) {
   ListItem oldNext = currentItem.getNext();
   // occasional artificial delay
   if (0.8 < random()) {
     try {
       Thread.sleep(1L);
     } catch (Exception e) {}
   }
   currentItem.setNext(newItem);
   newItem.setNext(oldNext);
 }
```

LinkedList.insertAfter:49

```
Local variables
  Frame 0: LinkedList.insertAfter
    ListItem oldNext @30bd3d30
      int value = 9513 @30bd3d20
      ListItem next = null @00000000
    ListItem currentItem @30bd3e24
      int value = 4238 @30bd3e14
      ListItem next @30bd4028
        int value = 6359 @30bd4018
        ListItem next @30bd3d30
          int value = 9513 @30bd3d20
          ListItem next = null @00000000
    ListItem newItem @30bd3f18
      int value = 7449 @30bd3f08
      ListItem next = null @00000000
    LinkedList this @30b73518
```

```
All threads: 7
  ID  VM_Thread    top stack frame
  --  -----------  -----------------

   2  @0x30aa8de0  morph   ()V
   3  @0x30add248  morph   ()V
   4  @0x30ae1a10  morph   ()V
   5  @0x30afa494  startoff   ()V
   6  @0x30b005a4  morph   ()V
 ->9  @0x30ba2e64  insertAfter   (LListItem;LListItem
  10  @0x30ba35e0  terminate   ()V
```

Breakpoint LinkedList:49 set

Figure 2: After a delay, Thread 9 is about to add `newItem` (7449) to the sorted list between `currentItem` (4238) and `oldNext` (9513). Notice how `currentItem` and `oldNext` are sadly out of date.

Single stepping from here, Figure 2 shows Thread 9 in the act of corrupting the list. The `insertAfter` method is supposed to insert a `newItem` $(7449)$ into the list between `currentItem` $(4238)$ and `oldNext` $(9513)$. This will disconnect $6359$, which is already in the list between `currentItem` $(4238)$ and `oldNext` $(9513)$. In a concurrent system, such an update should be atomic. This method is not. The artificial delay thus allows the next two statements to corrupt the list.

Although this example is contrived, such synchronization bugs are the bane of concurrent programming because of how hard they are to reproduce. DejaVu solves that problem by recording the execution of a multithreaded program so that it can be replayed on demand.

## 3 Jalapeño Background

The archetypal Java run-time service—automatic memory management, both object allocation and garbage collection—is completely deterministic in Jalapeño. However, its implementation impacts DejaVu's. To avoid memory leaks associated with conservative garbage collection, and to allow copying garbage collection, Jalapeño's collectors are type-accurate. That means every reference to a live object must be identified during collection.

Identifying such references in a thread's activation stack is problematic. Jalapeño *reference maps* specify these locations for predefined *safe-points* in the compiled code for a method.[3] At collection-time, Jalapeño guarantees that every method executing on every mutator thread has halted at one of these safe-points.

Jalapeño satisfies this guarantee through a custom thread package. Threads are switched quasi-preemptively at predetermined *yield points*, which occur exclusively in method prologues and on loop backedges. Yield points are a subset of safe-points. To ensure fairness, threads are preempted at the first yield point after a periodic timer interrupt—a key source of nondeterminism in Jalapeño.

The multithreading facilities of Jalapeño were designed to be highly efficient, modular, and independently tunable. Nevertheless, capturing the effects of interrupts would be a challenge to any replay tool. While not a run-time service per se, Jalapeño's support for the Java Native Interface (JNI) presents another challenge. JNI allows Java programs to make arbitrary calls to native code and vice versa. This flexibility further complicates replay because native code is beyond DejaVu's control.

## 4 Deterministic Replay

On a uniprocessor, an application's execution behavior is uniquely defined by (1) a sequence of *execution events* (i.e., bytecodes or instructions) and (2) the program's state after each execution event. Two executions are identical if (1) their execution sequences are identical and (2) their states after any two corresponding events are identical. For a multithreaded application, events can be executed by different threads. A *thread switch* is the transition from an event executed by one thread to an event executed by another. As we saw in Section 2, timing of a thread switch can affect the order of subsequent events, thereby potentially affecting program state.

The example in Figures 3A and B highlights how two different executions of the same program with the same state can result in different behaviors due to thread switches. The "`print y`" of Figure 3A will print "`8`", while the "`print y`" of Figure 3B will print "`0`".

[3]Jalapeño does not interpret Java bytecodes. Rather, one of three Jalapeño compilers translates these bytecodes to machine code. Currently, DejaVu uses Jalapeño's *baseline* compiler.

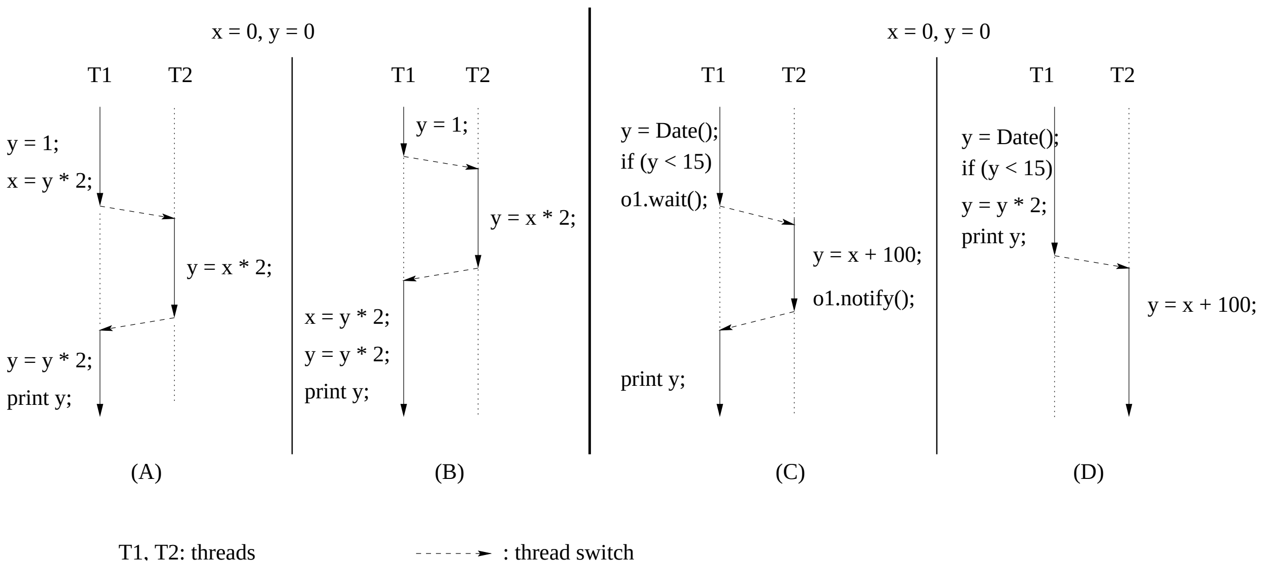


Figure 3: Nondeterministic Execution Examples

The program's state after an event can itself influence a thread switch by affecting the execution path following the event. Consider now Figures 3C and D, in which "`Date()`" returns today's date from the system wall-clock. In the example, different program states immediately after "`y = Date()`" take different branches after "`if (y < 15)`": the "`true`" branch is taken in Figure 3C, while the "`false`" branch is taken in Figure 3D. The "`true`" branch in Figure 3C results in a thread switch from T1 to T2 due to "`o1.wait()`" inside the branch. Meanwhile, the "`false`" branch in Figure 3D does not incur an immediate thread switch.

We can ensure that two executions of a multithreaded application are identical by ensuring identical thread switches and identical program states after corresponding events.

## 4.1 Ensuring Identical Program State

An event is *deterministic* if the same *in-state* produces the same *out-state*, where in-state and out-state are the program states immediately before and after the event, respectively. All the events in Figures 3A and B are deterministic. If all the events are deterministic, then execution behaviors remain identical as long as threads are switched identically, assuming the same starting state.

Some events are inherently nondeterministic: the same in-state can produce different out-states. An example of a nondeterministic event is reading the value of a wall clock during execution, like the "`Date()`" function in Figures 3C and D. Another example is reading a mouse motion or a keystroke. DejaVu handles such events by recording the out-state (or the change therein) during one execution and subsequently replacing nondeterministic operations with the retrieval of their prerecorded results.

## 4.2 Ensuring Identical Thread Switches

Three factors can cause thread switches in Jalapeño: (1) synchronization events, (2) timed events (such as `sleep` and timed `wait`), and (3) timer interrupts. Thread switches due to synchronization events are deterministic, while thread switches due to the latter two factors are nondeterministic.

## Replaying Deterministic Thread Switches

A thread switch occurs when a synchronization event blocks the execution of the current thread, as would a `wait` event or an unsuccessful `monitorenter` event. Synchronization events can also make a blocked thread ready to execute. Such events include `monitorexit`, `notify`, `notifyAll`, and `interrupt`.

A thread switch occurs when thread "T1" in Figure 3C executes "`o1.wait()`". This thread switch is deterministic; there will always be a thread switch on a `wait` event. The main challenge for replay here is to ensure thread "T2" becomes the next active thread in the presence of multiple ready threads.

An unsuccessful `monitorenter` event also generates a thread switch in Jalapeño, because the current thread is blocked until it can successfully enter the monitor (for example, a synchronized method or block in Java). Whether a `monitorenter` event is successful or not depends on program state, including the *lock state* of each thread. Thus `monitorenter` is usually a nondeterministic event. Cross-optimization of Jalapeño and its application actually benefits DejaVu in this regard, although it also presents problems that will be discussed later.

When DejaVu replays an application up to a synchronization operation (say a `monitorenter`), it replays the program state of Jalapeño as well. That includes Jalapeño's thread package, which maintains the lock state of each thread and lock variable plus the dispatch queue of threads. Therefore the synchronization operation will succeed or fail in replay mode depending on whether it succeeded or failed in record mode. If it fails, then the next thread dispatched during replay (as determined by the thread package) will be the same thread dispatched during record mode. That's because DejaVu also reproduces the data structure that the thread package uses to schedule threads. Similarly, a `notify` operation (as in Figure 3C) performed in replay mode will succeed or fail if it succeeded or failed in record mode.[4]

Cross-optimization simplifies the implementation of this behavior. No additional threading information need be captured or restored during replay to accommodate synchronization events. The thread scheduler will choose the correct thread because the scheduler itself is replayed.

## Replaying Nondeterministic Timed Events

The thread scheduler's state includes a queue of threads ready to execute (the *ready* threads) and a list of threads blocked due to synchronization operations (the *blocked* threads). Under DejaVu, blocked threads become ready threads normally as a result of other threads' wake-up operations such as `notify`, `notifyAll`, and `monitorexit`. Two exceptions are `sleep` and timed `wait` operations. A sleeping thread wakes up after a period specified in an argument to the `sleep` operation. A `wait` operation can specify a period after which a thread should wake up unilaterally. These timer-dependent operations must be handled specially.

Timer expiration depends on the wall-clock value and is nondeterministic with respect to application state. To ensure deterministic threading behavior during replay, timer expiration is based on equivalent program state, not wall-clock values alone. To handle `sleep` and timed `wait`, Jalapeño reads the wall clock periodically. The values read are nondeterministic, but their reproduction is deterministic under DejaVu. Therefore events that depend on wall-clock values, such as `sleep` and timed `wait`s, will execute deterministically. Reproducing wall-clock values is a special case of replaying nondeterministic events as described earlier.

[4]A `notify` operation on an object "succeeds" if there exists a thread waiting on the same object.

## 4.3 Replaying Preemptive Thread Switches

A nondeterministic thread switch occurs in Jalapeño as a result of a timer interrupt. Since the number of instructions executed in a fixed interval can vary, a nondeterministic number of instructions will be executed within each preemptive thread switch interval.

Cross-optimization simplifies things here too, since DejaVu replays Jalapeño's thread package. Ensuring identical preemptive thread switches requires identifying the events that occur following preemption while recording, and enforcing thread switches after the corresponding events during replay. The key issue here is how to identify the corresponding events in record and replay modes.

Wall-clock time is not a reliable basis for identification, because a thread's execution rate can vary due to external factors such as caching and paging. Instruction addresses are also insufficient—the same instruction can be executed many times during an execution through loops and method invocations. A straightforward counting of instructions executed by each thread will work, but the overhead is prohibitive.

The developers of *Instant Replay* [9] observed that events can be uniquely identified by a tuple containing an instruction address and a count of the number of backward branches executed by the program. Jalapeño exploits this observation by equating the number of yield points encountered to the number of backward branches executed. Since preemptive thread switches in Jalapeño occur exclusively at yield points, the yield-point count can uniquely specify preemptive thread-switch events.[5]

## 4.4 Symmetric Instrumentation

DejaVu cannot replay its own instrumentation, which behaves differently by definition: it writes data in record mode and reads data in replay mode. Ideally, DejaVu's execution should be *transparent* to Jalapeño—having no impact on its behavior except to effect replay.

However, cross-optimizing DejaVu, Jalapeño, and the application makes absolute transparency impractical. Side effects of DejaVu instrumentation may affect the virtual machine, the application, or both. For example, any class that DejaVu loads affects Jalapeño, since a class loaded by DejaVu will not be loaded again for Jalapeño. Hence class loading on DejaVu's part can change Jalapeño's execution behavior and potentially that of the application. Class loading can also affect the garbage collector, because loading usually involves allocating objects.

Where transparency cannot be achieved, DejaVu employs *symmetry* between record mode and replay mode: DejaVu activity that could affect the JVM (or DejaVu itself) is performed identically during both record and replay. Such activity includes

- object allocation,
- class loading and method compilation,
- stack overflow, and
- updating the logical clock.

[5]Recall that there is a yield point in every method prologue and on each loop backedge. So while the two counts are not identical, they serve the same purpose.

#### Symmetry in Object Allocation

DejaVu preserves symmetry in object allocation by allocating and using the same heap objects for both record and replay modes at a given point in the execution. For example, the same buffer stores information captured in record mode and information read from disk in replay mode. DejaVu's initialization phase pre-allocates the buffer in both modes. Additional heap objects are created as needed at a given point in either mode's execution.

#### Symmetry in Loading and Compilation

To maintain symmetry in class loading and method compilation, DejaVu pre-loads *all* its classes during initialization whether or not they will be instantiated. DejaVu also pre-compiles the corresponding methods at that time.

Furthermore, DejaVu pre-loads classes needed for file I/O, which is used to store information captured while recording and to read it back during replay. To preserve symmetry here, DejaVu writes to a temporary file (i.e., invokes output methods) and then immediately reads from that file (i.e., invokes input methods) during its initialization phase in both record and replay modes. This forces compilation of input and output methods in both modes.

#### Symmetry in Stack Overflow

Jalapeño allocates run-time activation stacks in heap objects (arrays), creating one when the current stack overflows. Should that happen, DejaVu maintains symmetry by ensuring that an overflow occurs at exactly the same point in the execution under both modes, whether in Jalapeño or in the application.

DejaVu's instrumentation in Jalapeño invokes different DejaVu methods in record and replay modes, since the modes do different things. The result can be unequal run-time activation-stack increments at corresponding invocations of a DejaVu method. These can result in different behaviors should a run-time-stack overflow occur. DejaVu addresses this problem by eagerly growing the run-time stack just before calling a DejaVu method whenever available stack space falls below a heuristically determined value.

#### Symmetry in Updating the Logical Clock

DejaVu's logical clock keeps track of the number of yield points executed by a thread. Since the instrumentation for record and replay perform different tasks, one might entail more yield points than the other. To keep the logical clocks in synch, yield points encountered in the course of executing instrumentation code are not reflected in the logical clock.

## 4.5 Java Native Interface

Native code can affect a Java program's execution in two ways: through return values or through callbacks. JNI callbacks can be made only through predefined JNI functions. DejaVu captures callback parameters and return values from native calls during record, and it regenerates them at the corresponding execution points during replay. This approach is sufficient since Jalapeño's implementation of JNI does not allow native code to obtain pointers into the Java heap.

DejaVu's current support for JNI replay assumes that native code runs for brief periods and does not block. DejaVu does not switch Jalapeño threads during native code execution. This has the effect of "freezing" time as a native method executes. Thus it's enough for DejaVu to record only return values and callback parameter values and not the time of their occurrence.

#### JNI Callbacks

Jalapeño generates a wrapper for each native method invocation. Each wrapper implements a prologue and epilogue, with the invocation to the native method in between. DejaVu instruments every wrapper, including those for predefined JNI callback functions. While recording, a callback function's prologue records a unique ID for the function in the DejaVu trace, and it records the native call's parameter values being passed to the callback function.

During replay, a wrapper invokes *JNIproxy*, a DejaVu method written in native code, instead of the original native method invoked during record. JNIproxy does two things. First, it reads the unique ID of the callback function stored during recording in the DejaVu trace. Second, JNIproxy invokes the callback function. The callback function's instrumented prologue then builds the parameter values by reading them from the trace.

#### JNI Returns

When the native method returns control to its enclosing wrapper during recording, the instrumented epilogue of the wrapper writes a value into the DejaVu trace. This value is guaranteed to be different from the unique ID of any callback function. Then the epilogue records the return value of the native-method invocation.

During replay, the wrapper invokes JNIproxy instead of the real native method. JNIproxy reads a value from the DejaVu trace and uses it to determine whether to return immediately (as indicated by having read an invalid callback ID) or to invoke a callback function. If the value indicates a return, JNIproxy returns to the wrapper of the native-method invocation. The epilogue of the wrapper then reads the returned value from the DejaVu trace and uses it as the return value.

## 5 Remote Reflection

The primary design constraint on a DejaVu-based debugger is to preserve the execution of the application being replayed. The execution must not be perturbed by basic debugger operations such as stopping and continuing, querying objects and program states, setting breakpoints, and so forth.

Jalapeño's Java-based implementation introduces another constraint. Jalapeño uses reflection extensively in its implementation. The debugger should thus exploit the same reflection interface in its interactions with the JVM and applications rather than introducing an ad hoc interface.

### 5.1 Implementation Challenges

Adherence to these constraints yields many benefits, but it also presents difficulties. First, to use reflection, the debugger must be an integral part of the system—that is, the debugger must execute in-process. But preserving deterministic execution across the entire system becomes problematic.

Suppose the application has stopped at a breakpoint, and the user wants to see a stack trace. The JVM must execute the debugger and its reflective methods to compute the requisite information. This action itself changes the state of the JVM: thread scheduling occurs, classes may be loaded, garbage collection might take place, etc. As a result, it may no longer be possible to resume deterministic execution.

Evidently, keeping the application JVM unperturbed during replay requires an out-of-process debugger—that is, a debugger that runs on an independent JVM. But that will put the application's reflection facilities out of the debugger's reach. Although the debugger can load the classes and execute reflection methods, the desired data resides in the application JVM, not the tool JVM.

More generally, conventional reflection code is assumed to execute in the address space of the client seeking reflection information. Reflection code is thus tightly coupled to the data it accesses.

### 5.2 Transparent Remote Access

Remote reflection solves this problem by decoupling reflection data and code. It allows a program running on one JVM to execute a reflection method directly on an object in another JVM. This allows a DejaVu-based debugger to execute out-of-process to avoid perturbing the application but still take full advantage of Jalapeño's reflection interface.

The key to enabling remote reflection is an object in the local (i.e., tool) JVM called the *remote object*, which serves as a proxy for the real object in the remote (i.e., application) JVM.

To set up the association between the two JVMs, a client (the debugger in our case) specifies a list of reflection methods that are *mapped*: when they execute in the tool JVM, they return a remote object that represents the actual object in the remote JVM. Typically, mapped methods are accessors that return internal state.

Once a remote object is obtained from a mapped method, all values or objects obtained from the remote object will originate from the remote JVM. A standard reflection method can be invoked on the remote object in the same manner as a normal object. From the client's perspective, a remote object is indistinguishable from a normal object in the local JVM save for the list of mapped methods.

The uniform treatment of local and remote objects offers the advantages of transparency. Because a remote object is logically identical to a local object, a client uses the same reflection interface whether it executes in-process or out-of-process. This simplifies maintenance of the reflection interface and the clients that use it.

A second advantage is that no overhead accrues in the remote JVM, since remote reflection relies on the operating system to access the remote JVM address space. In other words, the remote JVM does not execute any code to respond to queries from the debugger, and no JVM code is modified to support the debugger. This guarantees that the remote JVM is not perturbed by the debugger unless the user specifically wants to modify the state of the remote JVM.

Further details on DejaVu's implementation of remote reflection, along with examples of its use, appear elsewhere [11].

## 6 Related Work

Repeated execution is a widely accepted technique for debugging and understanding deterministic sequential applications. Repeated execution, however, cannot reproduce execution behavior of nondeterministic applications by itself. Replaying a nondeterministic application requires generating traces repeatedly until a given execution behavior is reproduced.

Many approaches for replay [9, 14, 12] capture the interactions among processes—i.e., the *critical events*—and generate traces for them. A major drawback of such approaches is the overhead, in time and particularly in space, of capturing critical events and generating traces.

*Igor*, *Recap*, and *PPD* are representative of early work in replay for debugging [6, 12, 10, 5]. All support replay (or "reverse execution") by checkpointing and re-executing from a previous checkpoint, as does recent work by Boothe [4]. Minimizing checkpoint overhead is the main challenge in such systems.

To reduce trace size, *Instant Replay* [9] assumes that applications access shared objects through coarse-grained operations called *CREW* (Concurrent-Read-Exclusive-Write). Instant Replay generates traces for just these coarse operations. This approach will not work for applications that do not use the CREW discipline, of course, but it also fails when critical events within CREW are nondeterministic.

Russinovich and Cogswell's approach [13] is similar to ours in that it captures only thread switches (rather than all critical events) on a uniprocessor. The Mach operating system was modified to notify the replay system on each thread switch. Since the approach does not replay Mach's thread package, the replay mechanism must tell the scheduler which thread to schedule. This requires a mapping between the thread executing during record and during replay—a significant execution cost that DejaVu avoids by replaying the entire Jalapeño thread package.

Holloman and Mauney's approach [8, 7] is similar to (and has the same drawbacks as) Russinovich and Cogswell's except for the mechanism that captures process scheduling information. Application code is instrumented with exception handlers that capture all exceptions sent from the UNIX operating system to the application process, including those for process scheduling.

Remote reflection integrates two common debugger features: out-of-process execution and reflection. Typical debuggers such as *dbx* or *gdb* are out-of-process too, but they rely on convention instead of reflection to interpret the data. The Sun JDK debugger [1] and the more recent Java Platform Debugger Architecture are out-of-process and reflection-based; however, both differ significantly from remote reflection in their approach.

First, the JDK debugger is geared toward user applications because it calls for the cooperation of the virtual machine. The reflection interface requires a dedicated JVM thread that responds to queries from the out-of-process debugger. In comparison, remote reflection requires no effort with respect to the target JVM; it executes no remote reflection code, and no JVM code is modified to support remote reflection. As a result, remote reflection can be used even when the JVM itself is defective.

Second, the JDK debugger uses a reflection interface that is different and separate from the internal reflection interface. The debugger's reflection interface may thus be implemented in native code, thereby minimizing JVM perturbation. But it also requires implementing and maintaining two reflection interfaces with similar functionalities. Remote reflection allows the same reflection interface to be used internally or externally.

## 7 Conclusion

This paper addressed the problem of building a perturbation-free run-time tool, such as a debugger, for heavily multithreaded nondeterministic Java server applications cross-optimized with the Java Virtual Machine (JVM). Cross-optimization improves overall performance. It also affords the precise instrumentation needed for run-time tools like DejaVu. However, cross-optimization introduces new challenges for replay from its own side-effects. We showed how DejaVu employs symmetry and remote reflection to meet these challenges.

We are working on two improvements to DejaVu. A third awaits further research.

### 7.1 Checkpointing Long Executions

It is usually inconvenient if not impossible to bring down a server to debug a client application. Since DejaVu captures all calls to native code as nondeterministic operations, replay can be carried out on a machine external to the production system.

For long-running server applications, replaying the application from the beginning may well be undesirable. We would like to be able to checkpoint a running Jalapeño JVM periodically and replay from any of these checkpoints. A checkpoint mechanism can leverage two existing Jalapeño mechanisms: bootstrap loading and garbage collection.

Initially, a special *boot image* [3] of a ready-to-execute Jalapeño JVM is created on an independent JVM and then written to a file. A short C program, the *boot image runner*, reads this file, copies the data to a predetermined location in memory, and transfers control to Java code that starts up each of Jalapeño's subsystems.

The format of a checkpointed Jalapeño image is a simple generalization of the boot image format. Instead of one big chunk of data with an implicit length and address, there would be a sequence of smaller chunks, each with an explicit length and address. The boot image runner would be adapted to handle such images. The Java code for restarting execution will closely follow the startup code but will reestablish the checkpointed state.

It remains to be shown how to write the state of a running application to a file in an appropriate format. The first task is to "quiesce" the system, halting all application activity in a state from which it can be restarted. Jalapeño's parallel, type-accurate, stop-the-world garbage collectors [2] already do this. The copying collector will also compact small objects into a large contiguous chunk. (Large objects are kept in a separate, uncopied area.) At the end of garbage collection, a Java method will write the image to a file in the proper format.

In general, checkpointing a system with open files (or sockets) is problematic, because a file might not be available when execution is restarted. This is not a problem for DejaVu since replay mode already simulates all I/O accesses.

## 7.2 Less-Intrusive Debugging Interface

Some debugging classes must execute on the tool JVM to enable remote reflection. But the debugger's Java-based graphical user interface (GUI) would be unacceptably slow if it were thus interpreted. Furthermore, the researchers working on Jalapeño typically execute the virtual machine remotely from a Windows machine, since both application and tool JVMs run on AIX. This too incurs overhead. Hence the GUI is designed to run on yet a third JVM, communicating with the debugger JVM through TCP. Developers can run the debugger remotely and the GUI locally, realizing both simple integration and satisfactory performance.

Remote reflection provides an effective debugging interface that exerts complete control over the JVM, allowing low-level debugging without perturbing the JVM. However, the JVM is frozen while the debugger runs. That makes remote reflection unsuitable for a production environment where the user application may be suspended but the JVM must remain running.

To accomplish that, we are investigating a more traditional approach that employs a debugging daemon in the JVM to serve a debugger client. The debugger is no longer isolated from the JVM; therefore the daemon must take care to avoid perturbing the application being replayed. Specifically, the daemon's side-effects must be recorded. The activity of the daemon itself is not recorded, and the daemon executes out of the scheduling control of DejaVu during replay.

## 7.3 SMP Record and Replay

On a uniprocessor, the interaction between threads can be completely characterized by the points at which the processor moves from one thread to another. By exploiting Jalapeño's quasi-preemptive thread-switching mechanism, DejaVu minimizes the overhead of recording this information.

Naturally, we would like to be able to record and replay the interaction of threads executing on a multiprocessor. But the multiprocessor problem is much more difficult, as threads potentially interact whenever they access a shared object. Systems that record and replay each shared-memory access on a multiprocessor could typically experience a hundredfold degradation in performance.

We hope to do substantially better than that by leveraging the features of the Java Memory Model [1]. Still, the problem is severe, and it seems unlikely that the overhead of multiprocessor record and replay can be reduced to the 15% or so level that would make it tolerable for production use. Even with a factor of five or ten overhead, however, record and replay in conjunction with a checkpoint and restart facility (itself a more difficult problem on a multiprocessor) could still be a useful tool for debugging and perhaps tuning multiprocessor applications.

## References

[1] Java Development Kit 1.1. Technical report, Sun Microsystems.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, 1999.

[4] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 299–310, June 2000.

[5] 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.

[6] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, May 1988.

[7] 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.

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

[9] 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.

[10] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144, June 1988.

[11] Ton Ngo and John Barton. Debugging by remote reflection. *Proc. of EURO-PAR 2000*, August 2000.

[12] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, May 1988.

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

[14] 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.