RC 22036 (98924) April 17, 2001 Computer Science

IBM Research Report

A Debugging Platform for Java Server Applications

Bowen Alpern, Jong-Deok Choi ,Ton Ngo Manu Sridharan,
John Vlissides, Hytm-Gyoo Yook
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
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 nondetenninistic execution behavior, availability requirements, and extended running times. New tools are needed to help programmers understand server behavior. Key to the realization of such tools is the ability to repeat nondeterministic execution behavior.

This paper presents a platform for understanding and debugging Java server applications. 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 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 to the basic functions to petit independent control and inspection of threads.

One aspect of multithread debugging has defied solution. 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 trends in software technology:

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 trends are not independent; addressing one of them can exacerbate the other. For example, Jalapeño [2] is a Java virtual machine (JVM) designed for high-performance servers. Written in Java, Jalapeiio uses a compilation-only

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

model to achieve dynamism *and* high performance by compiling Java bytecode directly to machine code at run-time. Unfortunately, two aspects of its compilation model can make program behavior even less predictable than usual: *cross-optimization* and *dynamic optimization*.

Cross-optimization improves overall 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 optimization that's performed and the application code that results 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 repeat 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 system 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.

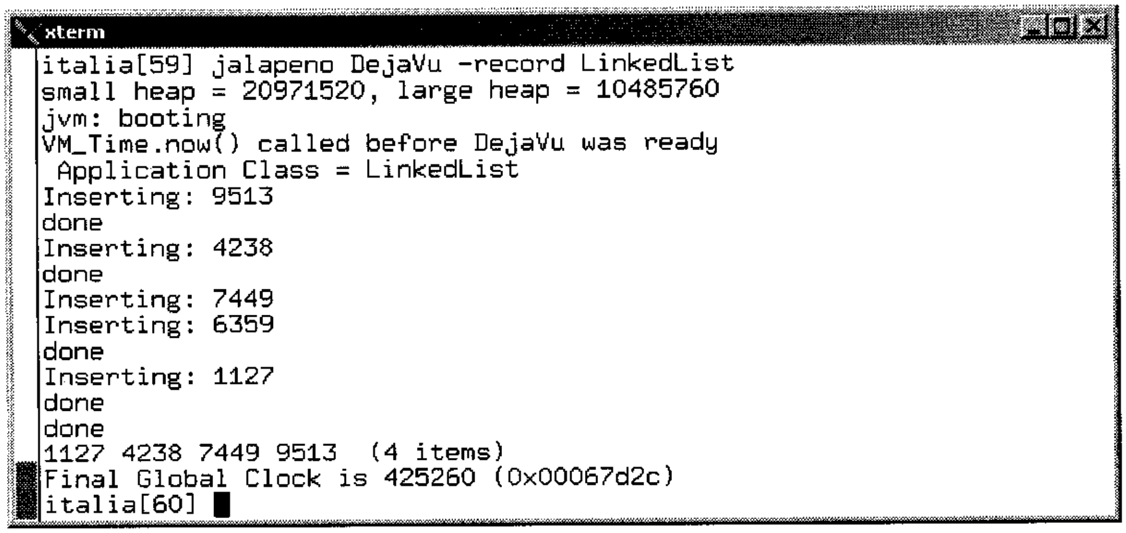
This paper describes *DejaVu*, a replay tool for multithreaded Java applications running on the Jalapeño JVM on uniprocessors.[1] DejaVu (*De*terministic *Ja*va Replay *U*tility) provides a replay capability on which to build higher-level tools for debugging and analyzing multithreaded applications.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

The next section illustrates the value of a replay tool for debugging multithreaded programs. Section 3 provides the necessary background on the Jalapeño JVM. Section 4 explains how DejaVu achieves accuracy and precision. Section 5 shows how a debugger running on DejaVu in replay mode can execute code in the JVM being debugged without changing its state. Section 6 reviews other approaches to deterministic replay. We conclude by considering three topics of further research: support for a sophisticated application-level debugger, support for replaying long-running applications, and support for multiprocessor replay.

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 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. One execution results in only four of the five numbers being printed—1127, 4238, 7449, and 9513 (Figure 1). The fourth number, 6359, has been lost. The DejaVu replaying debugger will be used *on this execution* to discover the cause.

[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.



```
italia[59] jalapeno DejaVu -record LinkedList
small heap = 20971520, large heap = 10485760
jvm: booting
VM_Time.now() called before DejaVu was ready
 Application Class = LinkedList
Inserting: 9513
done
Inserting: 4238
done
Inserting: 7449
Inserting: 6359
done
Inserting: 1127
done
done
1127 4238 7449 9513  (4 items)
Final Global Clock is 425260 (0x00067d2c)
italia[60] █
```

Figure 1: Output of an erroneous execution.

We begin by setting a breakpoint at the beginning of the `insert` method (Figure 2). At the initial breakpoint, the local variables window indicates the head member of the list is `null`, as expected, since nothing has been inserted yet. The value 9513 is about to be inserted into the list.

Continuing to the second breakpoint, we find 9513 on the list (as expected) with 4238 being inserted. At the third breakpoint, 4238 is at the head of the list, with 7449 apparently about to be inserted. The state at the fourth breakpoint is depicted in Figure 3; the value 6359 is about to be inserted. Recall that this is the value that did *not* appear in the output. Notice that 4238 and 9513 are on the list, but 7449 is not.[2]

At the next breakpoint (Figure 4), the last value, 1127, is about to be added to the list. Notice that 6359 is on the list, but 7449 is still missing.

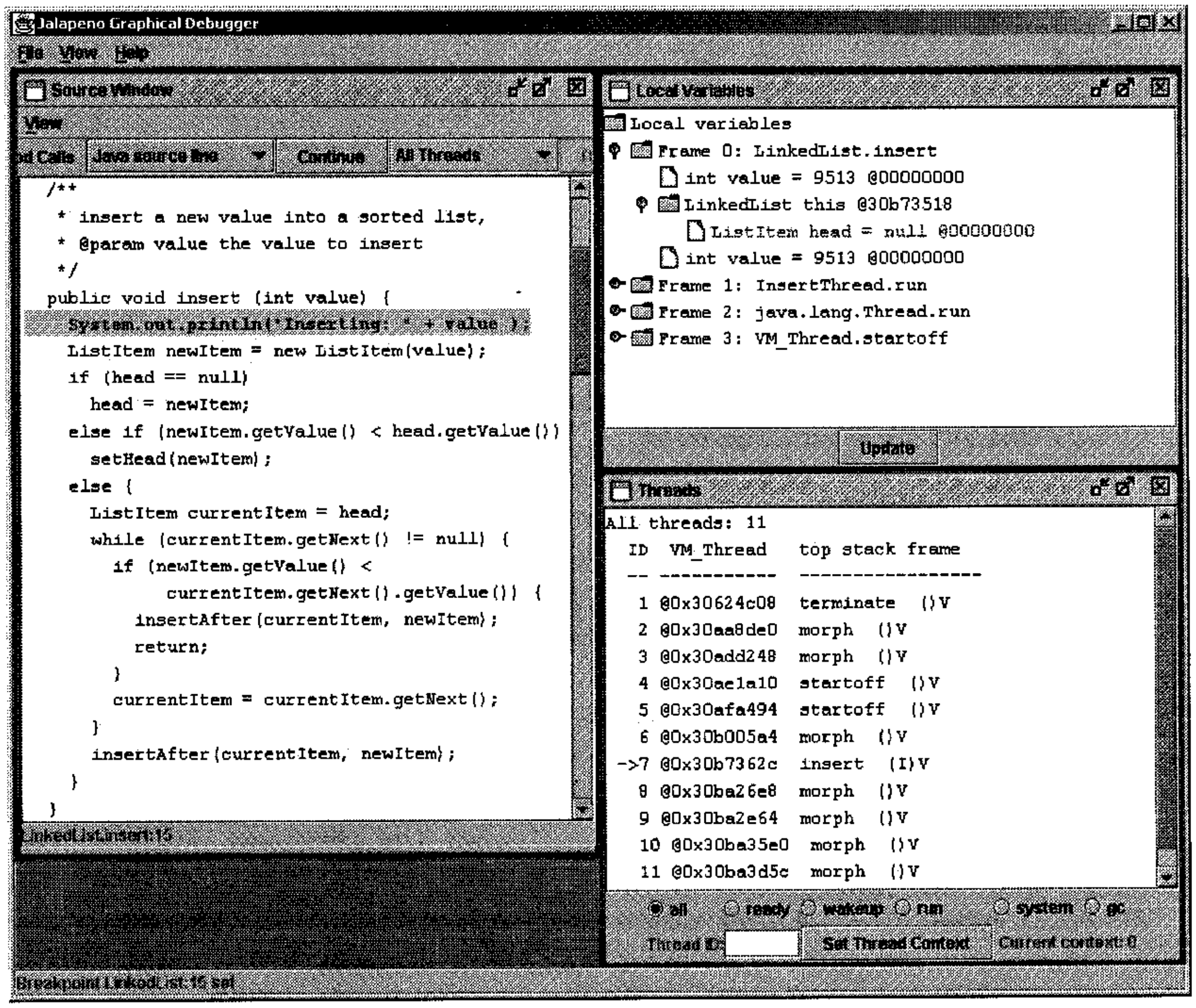
Single stepping from here, Figure 5 shows Thread 9 in the act of corrupting the list. The `insertAfter` method is supposed to insert a `newItem` (7449) into the list between `currentItem` (4238) and `oldNext` (9513). This will disconnect 6359, which is already in the list between `currentItem` (4238) and `oldNext` (9513). In a concurrent system, such an update should to be atomic. This method is not. Because of the artificial delay, execution of the next two statements will corrupt the list.

Although this example is contrived, similar intermittent synchronization bugs are the bane of concurrent programming. Perhaps the most annoying feature of such bugs is the difficulty of reproducing them. DejaVu solves that problem by recording the execution of a multithreaded program so that it can be replayed at will.

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.

[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.

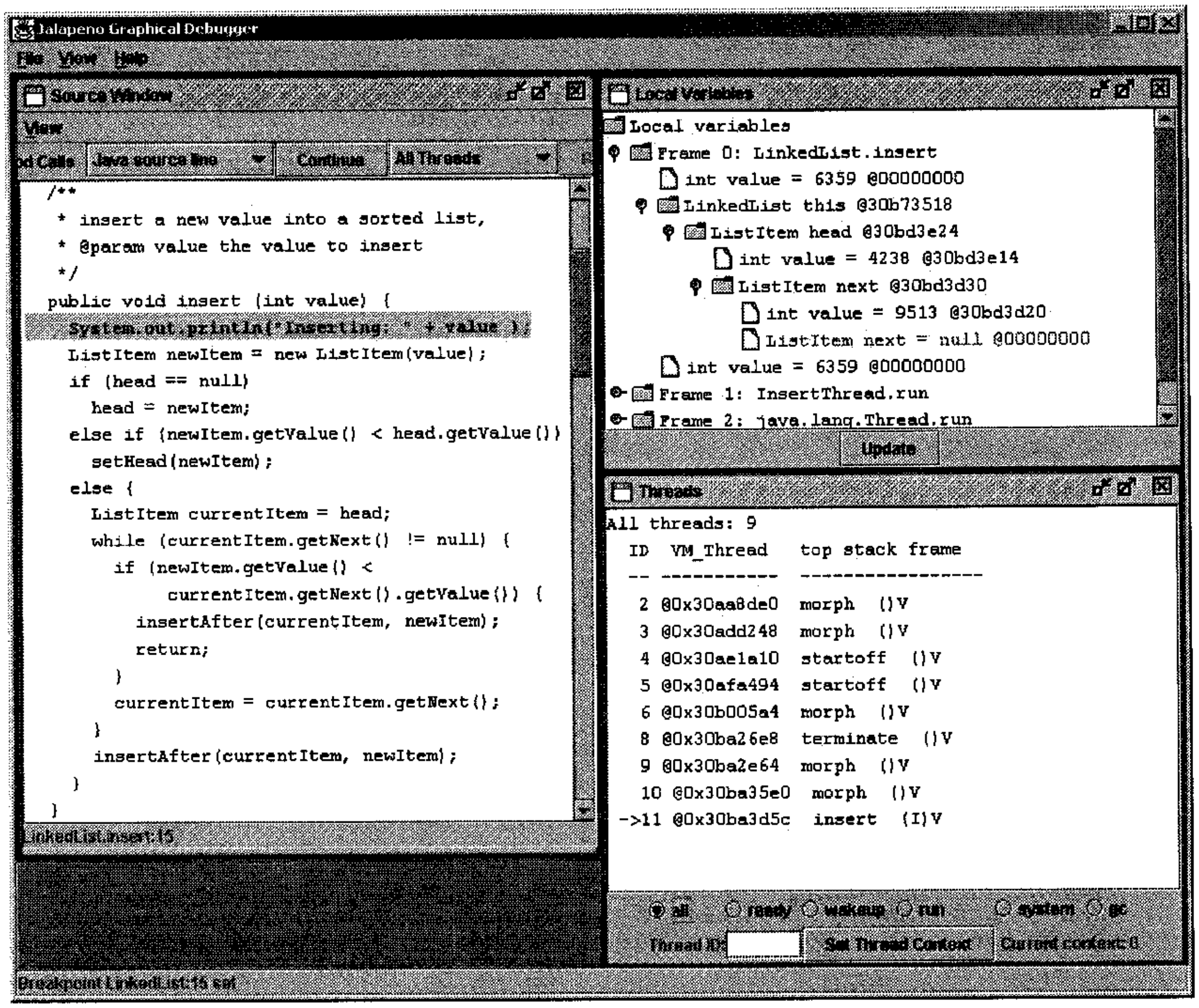


```
/**
 * insert a new value into a sorted list,
 * @param value the value to insert
 */
public void insert (int value) {
  System.out.println("Inserting: " + value );
  ListItem newItem = new ListItem(value);
  if (head == null)
    head = newItem;
  else if (newItem.getValue() < head.getValue())
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

```
Local variables
Frame 0: LinkedList.insert
    int value = 9513 @00000000
  LinkedList this @30b73518
      ListItem head = null @00000000
    int value = 9513 @00000000
Frame 1: InsertThread.run
Frame 2: java.lang.Thread.run
Frame 3: VM_Thread.startoff
```

```
All threads: 11
  ID  VM_Thread    top stack frame
  --  -----------  -----------------
   1 @0x30624c08  terminate  ()V
   2 @0x30aa8de0  morph  ()V
   3 @0x30add248  morph  ()V
   4 @0x30ae1a10  startoff  ()V
   5 @0x30afa494  startoff  ()V
   6 @0x30b005a4  morph  ()V
 ->7 @0x30b7362c  insert  (I)V
   8 @0x30ba26e8  morph  ()V
   9 @0x30ba2e64  morph  ()V
   10 @0x30ba35e0  morph  ()V
   11 @0x30ba3d5c  morph  ()V
```

Figure 2: Thread 7 at the breakpoint in `insert()` about to add 9513 to the (empty) sorted list.



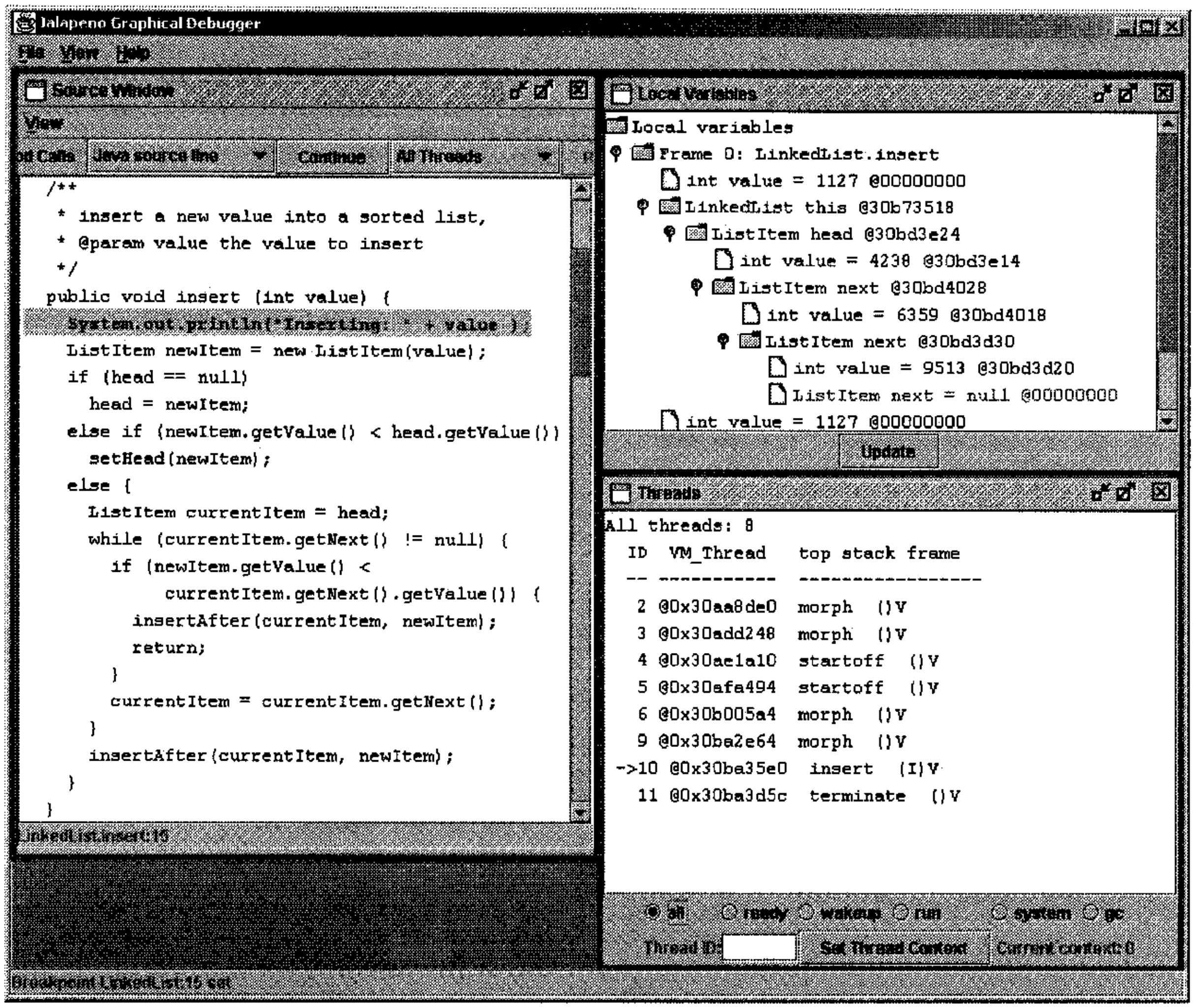
```
  /**
   * insert a new value into a sorted list,
   * @param value the value to insert
   */
  public void insert (int value) {
    System.out.println("Inserting: " + value );
    ListItem newItem = new ListItem(value);
    if (head == null)
      head = newItem;
    else if (newItem.getValue() < 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 3: Thread 11 at the breakpoint about to add 6359 to the list. Where is 7449?

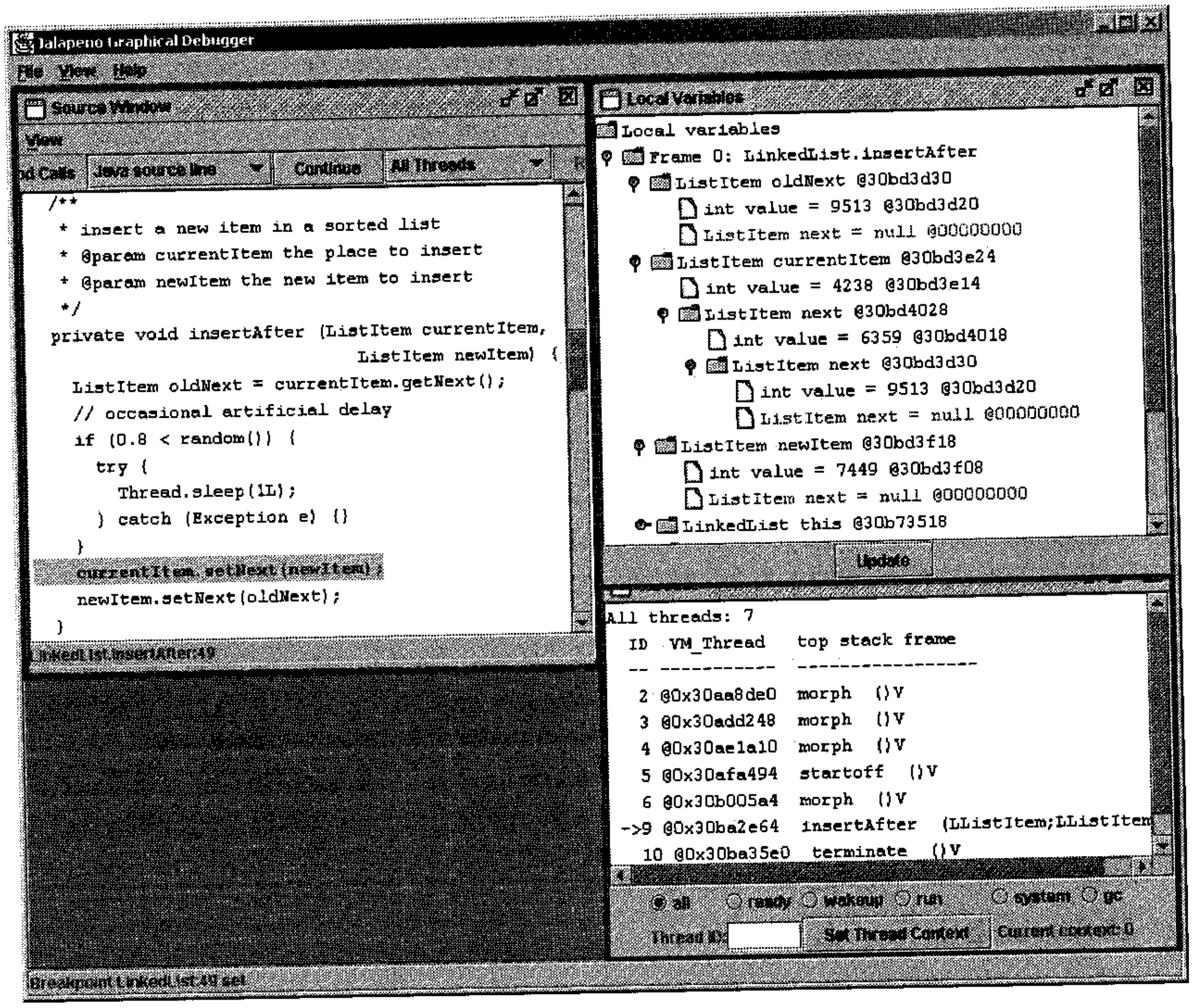


```
/**
 * insert a new value into a sorted list,
 * @param value the value to insert
 */
public void insert (int value) {
  System.out.println("Inserting: " + value );
  ListItem newItem = new ListItem(value);
  if (head == null)
    head = newItem;
  else if (newItem.getValue() < head.getValue())
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

```
Local variables
  Frame 0: LinkedList.insert
    int value = 1127 @00000000
    LinkedList this @30b73518
      ListItem head @30bd3e24
        int value = 4238 @30bd3e14
        ListItem next @30bd4028
          int value = 6359 @30bd4018
          ListItem next @30bd3d30
            int value = 9513 @30bd3d20
            ListItem next = null @00000000
    int value = 1127 @00000000
```

```
All threads: 8
  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
   9 @0x30ba2e64  morph   ()V
 ->10 @0x30ba35e0  insert  (I)V
  11 @0x30ba3d5c  terminate  ()V
```

Figure 4: Thread 10 about to add 1127 to the list.



```
/**
 * 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
```

Figure 5: 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.

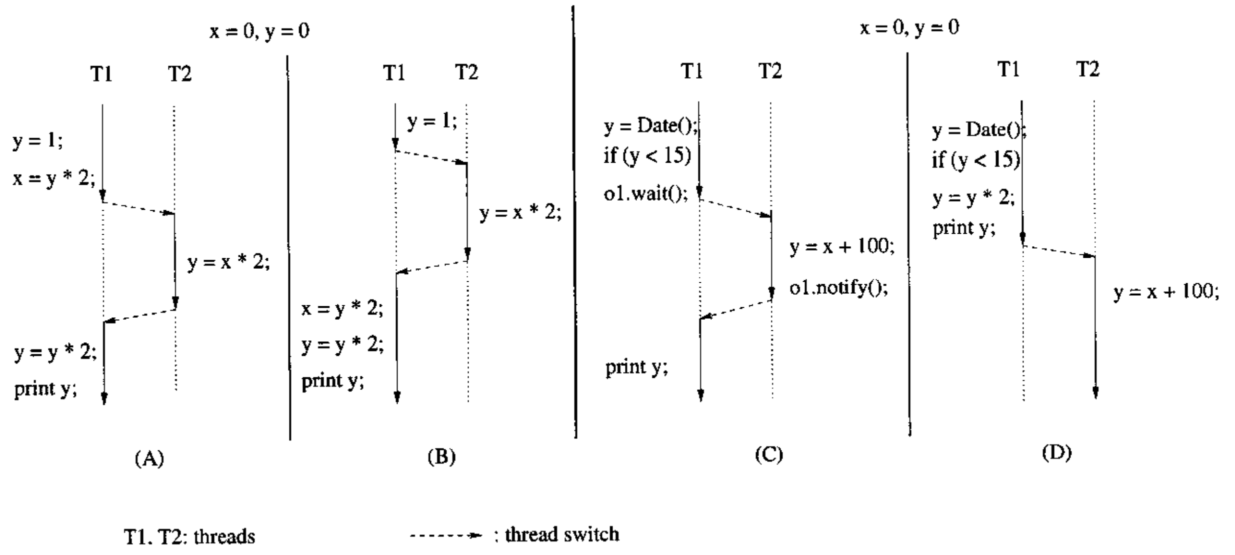


Figure 6: Nondeterministic Execution Examples

Identifying such references in the frames of a thread's activation stack is particularly 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. While their design aided greatly in implementing DejaVu (Jalapeño's thread packages were fairly easy to understand and modify), capturing the effect of the asynchronous 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* 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. In Java, an execution event can be defined either as the execution of a Java bytecode by an interpreter or the execution of a set of machine instructions generated from a bytecode by a compiler. (Note that a bytecode can be executed more than once; hence it may correspond to several events.)

For a multithreaded application, events can be executed by different threads. A *thread switch* is the transition in the

[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.

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 some 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 to be dispatched during replay mode (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 in selecting the next active thread.

Similarly, a `notify` operation (as in Figure 6C) performed in replay mode will succeed or fail if it succeeded or failed in record mode.[4] If it succeeded during record mode, it will succeed during replay mode and will awake the same thread among potentially multiple threads waiting on the same object.

Cross-optimization simplifies the implementation of this behavior. No additional threading information need be captured or restored during replay to accommodate programmer-specified synchronization events. For example, there is no need to remember which thread was scheduled when a thread blocks for synchronization. The thread scheduler will schedule the correct thread because the scheduler itself is replayed.

Replaying Nondeterministic Timed Events

The thread package'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. Consequently, readying a thread for execution based on wall-clock time affects subsequent threading behavior nondeterministically. To ensure deterministic threading behavior during replay, timer expiration is based on equivalent program state, not wall-clock values alone. DejaVu achieves this by reproducing the wall-clock values during replay mode.

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.

4.3 Replaying Preemptive Thread Switches

A nondeterministic thread switch occurs in Jalapeño as a result of preemption, based on a wall-clock timer interrupt. Since the number of instructions executed in a fixed wall-clock interval can vary, a nondeterministic number of

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

```
// during DejaVu record
// at every yield point
if (liveClock) {
   // only when the clock is running
   liveClock = false;
      // pause the clock
   nyp++;
   if (preemptiveHardwareBit) {
      // preemption required
      // by system clock
      recordThreadSwitch(nyp);
      nyp = 0;
         // reset the counter
      threadSwitchBitSet = true;
         // set the software switch bit
   }
   liveClock = true;
      // resume the clock
}

if (threadSwitchBitSet) {
    threadSwitchBitSet = false;
    performThreadSwitch();
}

            (A)
```

```
// during DejaVu replay
// at every yield point
if (liveClock) {
   // only when the clock is running
   liveClock = false;
      // pause the clock
   nyp--;
   if (nyp == 0) {
      // preemption performed
      // during record
      nyp = replayThreadSwitch();
         // initialize the counter
         // for the next thread switch
      threadSwitchBitSet = true;
         // set the software switch bit
    }
    liveClock = true;
      // resume the clock
}

if (threadSwitchBitSet) {
    threadSwitchBitSet = false;
    performThreadSwitch();
}

            (B)
```

Figure 7: DejaVu Instrumentation at Yield Points for Record (A) and Replay (B)

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 occurring after a preemptive thread switch 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 events, because a thread's execution rate can vary due to external factors such as caching and paging. Instruction addresses are also insufficient, as 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* [11] 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 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] Moreover, this count can be stored as a difference from the last such event (`nyp` in Figure 7).

The code in Figure 7A is executed at every yield point during recording. `nyp` is initially set to 0 and is incremented at each yield point.[6] `nyp` is recorded (and reset to 0) when a thread switch takes place.

The code in Figure 7B is executed at every yield point during replay. `nyp` is initially set to the first recorded value and is decremented at normal yield points. `nyp` reaches 0 at yield points that experienced a thread switch during

[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.

[6]The increment will not be performed when `liveClock` (described later) is `false`.

recording. It is then assigned a new value from the recorded data. The `preemptiveHardwareBit`, set by timer interrupt (and cleared by `performThreadSwitch()`) during record, is ignored during replay.

4.4 Symmetric Instrumentation

Note the similarity between Figures 7A (record) and B (replay). Such similarity is key to achieving accurate replay.

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: actions of DejaVu that might affect the JVM (or DejaVu itself) are performed identically during both record and replay. Such actions include:

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

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 captured information in record mode and captured 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 execution point in either mode.

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 captured information during record and to read it back during replay. DejaVu invokes input I/O methods during record and output I/O methods during replay. To maintain symmetry in loading these classes and compiling their methods, 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 own 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. (This is the purpose of the `liveClock` flag in Figure 7.)

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 direct 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. During record, 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 just 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 records 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.

Adherence to these constraints yields many benefits, but it also presents implementation challenges. First, to use reflection, the debugger must be an integral component 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 may take place, etc. As a result, it may no longer be possible to resume deterministic execution when the application continues.

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, reflection code is tightly coupled to the data it accesses. Conventional reflection assumes this code executes in the same address space as the client seeking reflection information.

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 and still take full advantage of Jalapeño's reflection interface.

5.1 Transparent Remote Access

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

```
class Debugger {
  public int lineNumberOf (int methodNumber, int offset) {
    VM_Method[] mTable     = VM_Dictionary.getMethods();
    VM_Method   candidate  = mTable[methodNumber];
    int         lineNumber = candidate.getLineNumberAt(offset);

    return lineNumber;
  }
}

class VM_Method {
  private int[] lineTable;

  public int getLineNumberAt(int offset) {
    if (offset > lineTable.length) {
      return 0;
    }
    return lineTable[offset];
  }
}
```

Figure 8: A Java method making reflective queries across JVMs. `Debugger.lineNumberOf()` invokes `VM_Dictionary.getMethods()` to obtain an array of `VM_Methods`; then the reflection method `getLineNumberAt()` is invoked on the remote object. Finally, `lineTable[offset]` is obtained from the remote JVM.

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.

An Example

Consider the simple example of Figure 8. A debugger executes on a local JVM that supports remote reflection, and DejaVu is replaying a remote JVM executing the application. To compute the line number, the `lineNumberOf()` method of `Debugger` invokes `VM_Dictionary.getMethods()` to obtain an array of `VM_Methods`. Then `lineNumberOf()` selects the desired element and invokes `getLineNumberAt()`. This reflection method then consults the object's internal array to return a line number.

To execute this code with remote reflection, we specify that `VM_Dictionary.getMethods()` should be mapped to an array of `VM_Methods` in the remote space. When executed, the code returns a remote object representing the actual array. Next, the `candidate` variable accesses the remote array and gets a second remote object. The `getLineNumberAt()` reflection method is then invoked on the remote object. Since the `lineTable` array is a member of the remote object, it too is a remote object. When this third remote array is accessed, the array element is obtained from the remote JVM. The net result is that the reflection method has transparently described an object two JVMs away.

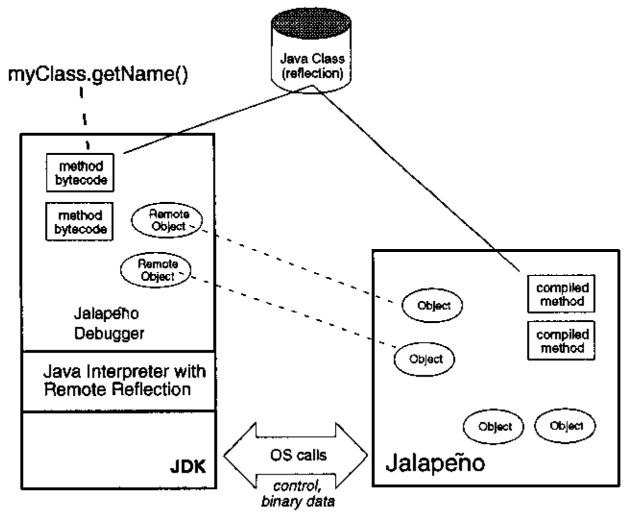


Figure 9: Implementation for Jalapeño: (1) a Java interpreter is extended to support remote reflection and runs on top of the Sun JVM; (2) Jalapeño loads and runs the reflection methods as compiled code; (3) the debugger loads and runs the reflection methods as bytecode; (4) remote objects are associated with the actual objects in the Jalapeño space.

5.2 Implementation

A standard Java interpreter is extended to implement remote reflection. The extension includes managing the remote object and extending the bytecodes to operate on the remote object. Remote reflection also requires operating system support for access across processes. This functionality is typically provided by the system debugging interface, which in the Jalapeño implementation is the Unix *ptrace* facility. Our implementation is simplified by the fact that the debugger merely issues queries and does not modify the state of the application JVM (except in response to a user request to change a value). Thus we need not create new objects in the remote space.

5.3 Remote Object

To implement the remote object, it is sufficient to record its type and real address. A remote object originates either from a mapped method or from another remote object. In the former case, the address is provided to the interpreter through the process of building the Jalapeño boot image [2]. In the latter case, the address is computed based on the field offset from the address of the remote object.

Native methods must be handled differently. Such methods are executed directly on the machine, and the interpreter will not be able to intercept references to remote objects. The JNI implementation on the tool JVM must therefore be extended to handle remote objects. For our debugger, however, the native methods that are required only operate on arrays of primitives such as byte and integer. So if a native method needs such an array of primitives as parameter, it is sufficient to clone the remote array on the local JVM before control is transferred to the native method. Note that this issue is separate from DejaVu's ability to replay native methods in the *application* JVM.

5.4 Bytecode Extensions

Since the initial remote object is obtained from a mapped method, the `invokestatic` and `invokevirtual` bytecodes for invoking a method must be extended.

The target class and method are checked against the mapping list to intercept invocations on mapped methods. The interpreter computes their return type to determine whether to return a remote object or a value. If the return type is an object, then a remote object is created containing the type and the address of the corresponding object in the remote JVM. If the return type is a primitive, the actual value is fetched from the remote JVM.

In addition, all bytecodes that operate on a reference need to be extended to handle remote objects appropriately; for Java, this amounts to 23 bytecodes. If the result of the bytecode is a primitive value, the interpreter (1) computes the actual address, (2) makes the system call to obtain the value from the remote address space, and (3) pushes the value onto the local Java stack. If the result is an object, then the interpreter (1) computes the address of the field holding the reference, (2) makes the system call to obtain the field value, and (3) pushes onto the Java stack a new remote object with the appropriate type.

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 [11, 15, 13] 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 in generating traces.

Igor, *Recap*, and *PPD* are representative of early work in providing replay capability as part of debugging [7, 13, 12, 5]. They all support replay (or "reverse execution") by checkpointing and re-executing from a previous checkpoint. Igor does not deal with nondeterminism in multithreaded applications explicitly [7]. Recap checkpoints program state by forking and suspending a new process [13]. It handles nondeterminism by capturing the effect of every read of shared memory locations, which is quite expensive. PPD performs program analysis to reduce the size of snapshots at checkpoints, and it captures the effect of every read of shared memory locations [12, 5]. Boothe's approach [4] is similar in that it too "reverse executes" by checkpointing and re-executing from a previous checkpoint. It also forks an idle process, like Recap, for checkpointing.

To reduce the trace size, *Instant Replay* [11] 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. It also fails when critical events within CREW are nondeterministic.

Russinovich and Cogswell's approach [14] 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 thread package which thread to schedule at each thread switch. This requires a mapping between the thread executing during record and during replay—a significant execution cost that DejaVu does not incur because it replays the entire Jalapeño thread package.

Holloman and Mauney's approach [9, 8] 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.

Earlier incarnations of DejaVu [6, 10] developed for SUN's JDK running on Win32 also limit tracing to thread switches.[7] These approaches suffer the same drawbacks as that of Russinovich and Cogswell.

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 a fixed data format 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 Sun JDK is geared toward user applications because it calls for the cooperation of the virtual machine. The reflection interface requires a dedicated debugging thread in the virtual machine to respond to queries from the out-of-process debugger. In comparison, remote reflection requires no effort with respect to the target JVM; the JVM does not execute any 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, Sun's JDK debugger uses a reflection interface that is different and separate from the internal reflection interface. This allows the debugger's reflection interface to 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 multi-threaded nondeterministic Java server applications cross-optimized with the Java Virtual Machine (JVM). We showed how Jalapeño's design for general extensibility and modularity allows efficient instrumentation of the application and the Jalapeño run-time system.

Cross-optimization improves overall performance. It also allows 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.

[7] Logging data for non-reproducible events such as reading the wall clock need be done independently of thread switch information in any replay scheme.

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

The debugger's graphical user interface (GUI) is based on Java's Swing framework. The classes providing the core debugger functionality must execute on the tool JVM to enable remote reflection, but the 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 box, since both the application JVM (Jalapeño) and the tool JVM 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. (Bandwidth is minimized by transmitting small packets of data rather than large bitmaps.) Our design lets developers run the debugger remotely while running the GUI on their local machine, affording both simple integration and satisfactory performance.

The GUI provides all the functionality found in most command-line debuggers, with added features of graphical debuggers. A view of the executing method's Java source and machine instructions allows setting breakpoints and single-stepping. The user inspects instances through a tree-based class viewer. The GUI also provides views of current breakpoints and the call stack along with the corresponding Java source code. A thread viewer is useful for finding subtle bugs in multithreaded applications.

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 completely isolated from the JVM; therefore the daemon must take care to avoid perturbing the application being replayed. Specifically, the daemon must be initialized and run in the background during recording so that its side effects are recorded. The activity of the daemon itself is not recorded, and the daemon executes out of the scheduling control of DejaVu during replay. Moreover, the daemon must not induce garbage collection during replay, and it must make a clear distinction between merely querying the state of the application and modifying it: as long as the user merely inspects values during a debugging session, the execution is preserved during replay.

To complement this high-level debugging capability, we plan to incorporate the debugger client from IBM VisualAge for Java. The daemon will implement the Java Debug Wire Protocol (JDWP) to communicate with the debugger client. The result will be a robust graphical user interface for DejaVu that is well-suited to a production environment.

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] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, August 1998.

[7] 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.

[8] 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.

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

[10] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed java applications. In *Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pages 219–228, May 2000.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.