RC 21864 (98290) 22 September 2000 **Computer Science**

IBM Research Report

A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications

Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan*, John Vlissides
IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

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

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications

Bowen Alpern[*]
alpern@watson.ibm.com

Jong-Deok Choi[*†]
jdchoi@watson.ibm.com

Ton Ngo[*]
ton@us.ibm.com

Manu Sridharan[*‡]
msridhar@mit.edu

John Vlissides[*]
vlis@us.ibm.com

ABSTRACT

Java applications, in particular server applications, are often multithreaded. Their execution can be non-deterministic, making them difficult to understand and debug. This paper presents a platform for analyzing interleaved program execution, comprising the Jalapeño JVM, the replay capabilities of *DejaVu*, and the perturbation-free reflection afforded by *remote reflection*. A debugger demonstrates the power and novel characteristics of the platform.

DejaVu supports understanding and debugging multithreaded Java applications through deterministic replay of non-deterministic execution. DejaVu replays the execution of the entire Jalapeño JVM, including its thread and garbage collector subsystems. The debugger must not alter the execution behavior of the JVM during replay, which is especially challenging on this platform because all components—from the Jalapeño JVM to DejaVu and the debugger—are written in Java. The keys are symmetric instrumentation in DejaVu and remote reflection which exposes the state of an application perturbing it.

[*]Address: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

[†]Corresponding author.

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

Keywords

Java, program development tool, concurrent programming, program replay, debugging, reflection

1 INTRODUCTION

Tools for accurately repeating non-deterministic computation are important for debugging and tuning server applications. On a uniprocessor, construction of execution replay tools is considerably eased if a clear distinction is maintained between the application and the underlying runtime system that supports its execution. This paper considers building such tools for an environment where the line between application and runtime has been significantly blurred.

Cross-optimization refers to an environment in which an application and its runtime system are analyzed and optimized 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 runtime environment can improve overall performance.

Jalapeño [2] is a Java virtual machine (JVM) for high-performance servers that employs cross-optimization. Written in Java, Jalapeño brings the benefits of cross-optimization to server design and implementation. Jalapeño uses a dynamic compilation-only strategy that further enhances the effectiveness of cross-optimization.

Large scale multithreading in server applications makes their executions highly non-deterministic. Debugging such programs is particularly difficult since it's hard to fix something that doesn't even fail reliably. It is therefore useful to have a tool

that is able to reproduce an errant behavior when it has been observed. This paper describes *DejaVu* (**de**terministic **Jav**a replay **u**tility) for Jalapeño, a tool that deterministically replays uniprocessor[1] Jalapeño executions of multithreaded Java applications.

A replay tool will typically require instrumenting application (and possibly runtime) code. If, as is usually the case, there is a performance penalty for such instrumentation, then the code will normally be executed with the instrumentation turned off. A replay tool strives to be both *accurate*, in that the replayed code exhibits exactly the same behavior as the instrumented code, and *precise*, in that the instrumented code exhibits behavior that is close to that of the uninstrumented code. (Note that the accuracy requirement is absolute while precision is a matter of degree.)

Cross-optimization is a boon to achieving precision, since it allows instrumentation code, application, and runtime code to be integrated and optimized together. However, cross-optimization makes accuracy more difficult.

DejaVu achieves accuracy by dividing the operations of an application and its runtime into *deterministic* operations (such as instruction executions), which necessarily produce the same result on replay, and *non-deterministic operations* (such as environmental queries), which do not. In record mode, DejaVu ignores deterministic operations while recording the results of non-deterministic operations. In replay mode, it again ignores deterministic operations while systematically replacing non-deterministic operations with the retrieval of their prerecorded results.

It is fairly easy to isolate non-deterministic operations (on a uniprocessor) if the application and runtime are distinct: application code is deterministic, and all runtime services may be treated as non-deterministic, although there may be a precision penalty for treating deterministic services as though they were non-deterministic.) With cross-optimization, identifying and isolating non-deterministic runtime services is more challenging.

The archetypical Java runtime service — automatic memory management, both object allocation and garbage collection — is completely deterministic in Jalapeño. However, its implementation has implications for DejaVu. To avoid memory leaks associated with conservative garbage collection and to allow copying garbage collection, all of Jalapeño's garbage collectors are type-accurate. This means that every reference to a live object must be identified during garbage collection. 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.[2] At garbage-collection time, Jalapeño guarantees that every method executing on every mutator thread is stopped at one of these safe-points.

To make good on this guarantee, Jalapeño contains its own thread package that performs quasi-preemptive thread switching only when the current running thread is at a predetermined *yield point* (in method prologues and on loop backedges). Yield points are a subset of safe-points. To achieve some measure of fairness among Java threads, they are preempted at the first yield point after a periodic timer interrupt. These timer interrupts are noteworthy source of non-determinism in Jalapeño. Capturing the effect of such asynchronous interrupts would be a challenge to any replay tool. The multithreading facilities of Jalapeño were designed to be highly efficient, modular, and independently tunable. This design aided greatly in implementing DejaVu, as Jalapeño's thread packages

[1] Replay of *multiprocessor* executions is a considerably harder problem that we hope to be able to address in the future. Nonetheless, it should be apparent that even a uniprocessor replay engine (as described here) will be useful in understanding and debugging multithreaded programs primarily intended to run on multiprocessors.

[2] 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.

were fairly easy to understand and modify.

One of the challenges of integrating DejaVu's instrumentation into the application (and runtime) is that this instrumentation behaves differently in record and replay mode. In record mode, the instrumentation writes information; in replay mode, it reads it. Like Jalapeño, this instrumentation is written in Java. Consider what would happen if the replay instrumentation triggered a class load that didn't happen (or happened at a later point) during record. DejaVu employs symmetry to prevent different behaviors of DejaVu between record and replay from precluding accurate replay. Any side effects of DejaVu that might affect the execution behavior of Jalapeño and the application are faithfully generated during both record and replay.

The requirements of symmetry also place a burden on tools based on DejaVu. Consider, for instance, a DejaVu-based debugger: one would like to be able to interrupt a replay, inspect the state of the Jalapeño heap, and resume the replay. Java's reflection facility provides an effective mechanism for inspecting the heap. However, if this facility is invoked in Jalapeño in replay mode, the symmetry between record and replay is broken and replay cannot be resumed. Since these side-effects of debugging cannot be incorporated into the record and replay mechanism to achieve symmetry, tools built with DejaVu must run in a separate JVM from the one running the application to avoid perturbing the replayed application.

To retain the advantage of reflection, the JVM running the tool (the *tool JVM*) employs a technique called *remote reflection* [8], which enables reflection to operate across the separate address spaces between the two JVM's. The tool JVM interprets the same reflection methods of the JVM running the application (the *application JVM*), but it uses the application JVM's data by intercepting the reflection bytecode and by transparently mapping the objects' data between the address spaces.[3] This allows a debugger running on the tool JVM to query program state by invoking the JVM's internal reflection methods without affecting the state of the application JVM.

By combining symmetric instrumentation and remote reflection, DejaVu for Jalapeño serves as a perturbation-free replay platform that enables a family of replay-based *development* tools for understanding and performance tuning, as well as for debugging, cross-optimized multithreaded applications. The next section presents DejaVu's replay strategy in detail. Section 3 explains remote reflection and its application to DejaVu. Section 4 describes DejaVu's GUI interface. Section 5 considers related work and section 6 concludes.

2 DETERMINISTIC REPLAY

This section describes how DejaVu for Jalapeño deterministically replays an execution behavior.

Background

On a uniprocessor system, execution behavior of an application can be uniquely defined by (1) the sequence of *execution events*, and (2) the program state after each execution event. Therefore, two execution behaviors of an application are identical if (1) their execution sequences are identical, and (2) the program states after any two corresponding events are identical. In Java, an (execution) event can be defined as an execution of a Java bytecode by an interpreter or an execution of a set of machine instructions generated from a bytecode by a compiler. Note that because a bytecode can be executed more that once, 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 event sequence from an event executed by one thread to an event executed by another thread. The timing of a thread switch can affect the order of events after the thread switch and hence can affect the program state after an event.

The example in Figure 1 (A and B) illustrates how

[3] It is possible for such a tool to allow a user to intentionally alter the state of the application, but this would irrevocably break the symmetry between record and replay. Replay could still be resumed, but no guarantee could be made as to its accuracy.

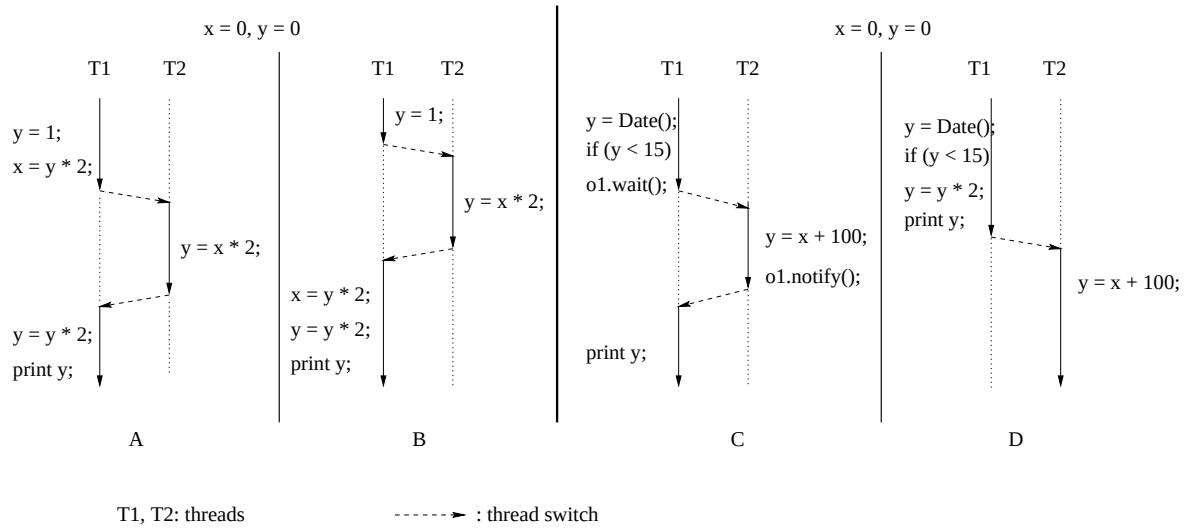

Figure 1: Non-Deterministic Execution Examples.

two different executions of the same program with the same initial state can still result in different behaviors due to the timing of thread switching. The "print y" of Figure 1-A will print 8, while the "print y" of Figure 1-B will print 0.

The program state after an event can itself affect when a thread switch occurs by affecting the execution path following the event. Consider now Figure 1-C and Figure 1-D, in which "Date()" returns today's date from the system wall-clock. In the example, different program states immediately after "y = Date()" made different branches taken after "if (y $<$ 15)": the "true" branch was taken in Figure 1-C, and the "false" branch was taken in Figure 1-D. The "true" branch in Figure 1-C resulted in a thread switch from T1 to T2 due to "o1.wait()" inside the branch, while the "false" branch in Figure 1-D did not result in an immediate thread switch.

We can ensure two executions of a multithreaded application are identical by ensuring identical thread switches and identical program states after corresponding events. We first describe the technique to ensure identical program states after corresponding events, assuming identical thread switches. We then describe the technique to ensure identical thread switches, assuming identical program states after corresponding events. Combining these two techniques ensures identical execution behaviors of different executions.

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 Figure 1-A and Figure 1-B are deterministic. If all the events are deterministic, execution behaviors remain identical as long as thread switches occur the same way, assuming initial program states are identical.

Some events are inherently non-deterministic: the same in-state can produce different out-states. An example non-deterministic event is reading the value of a wall clock during execution, like the "Date()" function in Figure 1-C and Figure 1-D. Another example is reading a key stroke or mouse movements. DejaVu handles a non-deterministic event by capturing the (change in the) out-state during one execution and regenerating the same out-state during another execution.

Ensuring Identical Thread Switches

In Jalapeño, three factors affect thread switches: (1) synchronization events; (2) *timed events* such as `sleep` and timed `wait`; and (3) preemptive thread switches based on timer interrupts. Thread switches due to synchronization events are deter-

ministic, while thread switches due to the other two are non-deterministic.

Replaying Deterministic Thread Switches
A thread switch occurs when a synchronization event blocks the execution of the current thread. A `wait` event or an unsuccessful `monitorenter` event corresponds to this case. Synchronization events can also make a blocked thread ready to execute. Events corresponding to this case are `monitorexit`, `notify`, `notifyAll`, and `interrupt` events.

A thread switch occurred when thread "T1" in Figure 1-C executed "o1.wait()". This thread switch is deterministic in that there will always be a thread switch at that event. The key issue for the replay in this case is how 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) since the current thread is blocked until it can successfully enter the monitor: e.g., a synchronized method or block in Java. Whether a `monitorenter` event is successful or not depends on the program state, including the *lock state* of each thread, and is generally a non-deterministic event. Cross-optimization of Jalapeño and its application, however, benefits DejaVu in this regard, although it also presents some problems (to be discussed later).

When DejaVu replays an application up to a synchronization operation (say `monitorenter`), it replays the program state of Jalapeño as well, including its 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 during replay mode depending on whether it succeeded or failed during record mode. If it fails, moreover, the next thread to be dispatched during replay mode (as determined by the thread package) will be the same thread dispatched during record mode. This is because the data structure used by the thread package in selecting the next active thread will also be exactly reproduced by DejaVu.

Similarly, a `notify` operation, as in Figure 1-C, performed on an object during replay mode will succeed or fail if it succeeded or failed during record mode.[4] If it succeeded during record mode, it will succeed during replay mode and awake the same thread among potentially multiple threads waiting on the same object.

Cross-optimization simplifies the implementation of this behavior in that no additional information need be captured or restored during replay to accommodate programmer-specified synchronization events.

Replaying Non-Deterministic Timed Events
The thread package's state includes a list 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 normally become ready threads as a result of operations from other threads that wake up the blocked threads, 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 (hence the term "timed `wait`"). These timer-dependent operations must be handled specially.

Timer expiration depends on the wall-clock value and is non-deterministic with respect to application state. Consequently, readying a thread for execution based on wall-clock time affects subsequent threading behavior non-deterministically. 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 non-deterministic, but their reproduction is deter-

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

ministic 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 non-deterministic events, described above.

Replaying Preemptive Thread Switches
A non-deterministic 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 non-deterministic 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 after which a preemptive thread switch occurred during record, 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.

Wall-clock time is not a reliable basis for events, because a thread's execution speed 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 recursion. A straightforward counting of instructions executed by each thread will work, but the overhead is prohibitive.

Following the approach of *Instant Replay* [7] DejaVu uniquely identifies an event during execution using a *software PC*, which is a $< NBB, PC >$ tuple. NBB is the number of back branches executed since the start of execution, and PC is the instruction address. Counting the number of back branches can distinguish multiple executions of the same instruction due to loops and method invocations alike.

Jalapeño ensures that a non-deterministic thread switch occurs only at predetermined yield points — back-branch targets or method prologues in the program. Yield points obviate PC, making NBB sufficient for uniquely identifying each non-deterministic thread switch. This simplifies the instrumentation for capturing non-deterministic thread switches, demonstrating another synergy in cross-optimizing Jalapeño and DejaVu. Moreover, NBB need only record the incremental number of back branches since the previous non-deterministic thread switch, thus requiring fewer bits. (A single 32-bit register can accommodate roughly four billion instructions between non-deterministic thread switches.)

The following code is executed at every yield point during DejaVu record. "`nbb`" is initially set to "`0`". "`nbb`" plays the role of a logical clock used in measuring the (logical) time interval between two preemptive thread switches. The role of "`liveClock`" will be described in more detail in the following section.

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

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

The following code is executed at every yield point during DejaVu replay. "`nbb`" is initially set to the first "`nbb`" value during record. Note that, unlike the record phase above,

"`preEmptiveHardwareBitSet`" is ignored during replay.

```
// during DejaVu replay
// at every yield point
if (liveClock) {
 // only when the clock is running
 liveClock = false;
  // pause the clock
 nbb--;
 if (nbb == 0) {
   // preemption performed
   // during record
   nbb = 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();
}
```

Symmetric Instrumentation

Accurate replay precludes replaying DejaVu itself, which behaves differently by definition: it *records* in record mode and *replays* in replay mode. Ideally, DejaVu's execution behavior must not affect Jalapeño—it must be *transparent* to Jalapeño—except that, unbeknownst to Jalapeño, its execution too is being replayed.

Cross-optimizing DejaVu, Jalapeño, and the application, however, makes transparency almost impossible to achieve, because side effects of DejaVu can affect both Jalapeño and the application. 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 new heap 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

To maintain symmetry in object allocation, which can affect the garbage collector, DejaVu allocates and uses (at a given point in the execution) the same heap objects for both record and replay modes. For example, it uses the same buffer to store captured information in record mode and to store captured information read from disk in replay mode. DejaVu pre-allocates the buffer independent of mode during its initialization. Additional heap objects are created as needed at a given execution point in both record and replay modes.

Symmetry in Loading and Compilation

DejaVu maintains symmetry in class loading and method compilation by pre-loading all the classes of DejaVu, whether needed only for record or replay, during its initialization before the application starts. DejaVu also pre-compiles the methods in the pre-loaded DejaVu classes during initialization. Furthermore, DejaVu pre-loads classes needed for file I/O (to store captured information during record and to read it back during replay). The I/O methods DejaVu invokes are input methods during record, and output methods during replay. To maintain symmetry in loading the classes and compiling methods for I/O, DejaVu writes into a temporary file (i.e., invokes output methods) and then immediately reads from that file (i.e., invokes input methods) as part of DejaVu initialization during both record and replay. This forces both input methods and output methods to be compiled during both record and replay.

Symmetry in Stack Overflow
Jalapeño allocates runtime activation stacks in heap objects (arrays), creating a new one when the current stack overflows. Should that be necessary, DejaVu maintains symmetry by ensuring that an overflow occurs at exactly the same point in the execution during 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 runtime activation-stack increments at corresponding invocations of a DejaVu method. Furthermore, runtime activation-stack increments can vary due to differing runtime activation-stack depths. These can result in different behaviors in runtime-stack overflow. DejaVu addresses this problem by eagerly growing the runtime activation stack just before calling a DejaVu method when 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, none of the yield points encountered while executing instrumentation code is counted in the logical clock. (This is the purpose of the "liveClock" flag in the above code.)

Java Native Interface
The Java Native Interface (JNI) allows for a Java application to interact with native code. Execution behavior of a Java application can be affected by native code in two ways: through return values or callbacks. Callbacks can be made only through pre-defined JNI functions. DejaVu captures return values from a native call and callback parameters during record, and 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.

3 REMOTE REFLECTION

The first goal for a debugger integrated with DejaVu is to preserve the execution of the application being replayed. The execution must not be perturbed by the usual action of the debugger such as stopping and continuing, querying objects and program states, setting breakpoints, etc.

Jalapeño's Java-based implementation adds a second goal for the debugger. Jalapeño uses reflection extensively for all objects so that the many system components can be integrated seamlessly and effectively. As a result, there is a strong motivation for the debugger to exploit the same reflection interface in querying and controlling the JVM and the applications instead of using a different ad hoc interface.

These two goals yield many advantages but they lead to a conflict in the implementation. First, to use reflection, the debugger must be an integral component of the system — in other words, the debugger must execute in-process — but maintaining the deterministic execution of the entire system becomes problematic. For example, suppose the application has stopped at a breakpoint and the user wants to display stack trace. The JVM must then execute the debugger and its reflective methods to compute the desired information. This action itself changes the state of the JVM because thread scheduling occurs, classes may be loaded, garbage collection may take place, etc. As a result, it may no longer be possible to resume the deterministic execution when the application continues.

On the other hand, 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 out of the debugger's reach. Although the debugger can load the classes and execute the reflection methods, the desired data resides in the application JVM rather than the tool JVM.

At a higher level, the general problem is that with reflection, the data and the code describing it are tightly coupled. In other words, the code must be

executed in the same address space to obtain information about the data.

Remote reflection solves this problem by decoupling the data and its reflection code, thus allowing a program in one JVM to execute a reflection method that operates directly on an object residing in another JVM. In the case of DejaVu, the debugger can execute out-of-process to avoid perturbing the application, yet it can take full advantage of Jalapeño's reflection interface.

Transparent remote access

Remote Reflection allows remote data to be accessed transparently in the Java programming model. The key to remote reflection is an object in the local (tool) JVM called the *Remote Object*, which serves as a proxy for the real object in the remote (application) JVM.

To set up the association between the two JVM's, the user (i.e., the debugger) specifies a list of reflection methods that are said to be *mapped*: when they are executed in the tool JVM, they return a remote object that represents the actual object in the remote JVM. Typically, these are access methods that return the internal components of an object.

Once a remote object is obtained from a mapped method, all values or objects derived from it will also originate from the remote JVM. The standard reflection method can be invoked on the remote object in the same way as a normal object. Aside from the list of mapped methods, a remote object is indistinguishable from a normal object in the local JVM from the program's perspective.

The uniform treatment of local and remote objects gives the advantage of transparency. Because a remote object is logically identical to a local object, a program uses the same reflection interface whether it executes in-process or out-of-process. As a result, the maintenance of both the reflection interface and programs using it is greatly simplified.

A second advantage is that no effort is required in the remote JVM, since remote reflection relies on the underlying operating system to access the remote JVM address space. This guarantees that the remote JVM is not perturbed by any action of the debugger, unless the user specifically wants to modify the state of the remote JVM.

Consider a simple example in Figure 2. In this case, the debugger is executing in the local JVM that supports remote reflection. The application (with its runtime) being replayed is the remote JVM.

To compute the line number, the *lineNumberOf()* method of *Debugger* invokes the *VM_Dictionary.getMethods()* method to obtain a table of *VM_Method*'s. Then it selects the desired element and invokes its virtual *getLineNumberAt()* method. 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 the *VM_Dictionary.getMethods()* method is to be mapped to an array of *VM_Method*'s in the remote space. Therefore, when it is executed, it returns the initial 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 an instance field of the remote object, it is also 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 across two JVM's.

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 only makes queries and does not modify the state of the application JVM (except in response to a user request to change a *value*); we need not create new objects

```
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 2: A Java method making reflective queries across JVM's. *Debugger.lineNumberOf()* invokes *VM_Dictionary.getMethods()* to obtain a table of *VM_Method's*, then the reflection method *getLineNumberAt()* is then invoked on the remote object. The final result *lineTable[offset]* is obtained from the remote JVM.

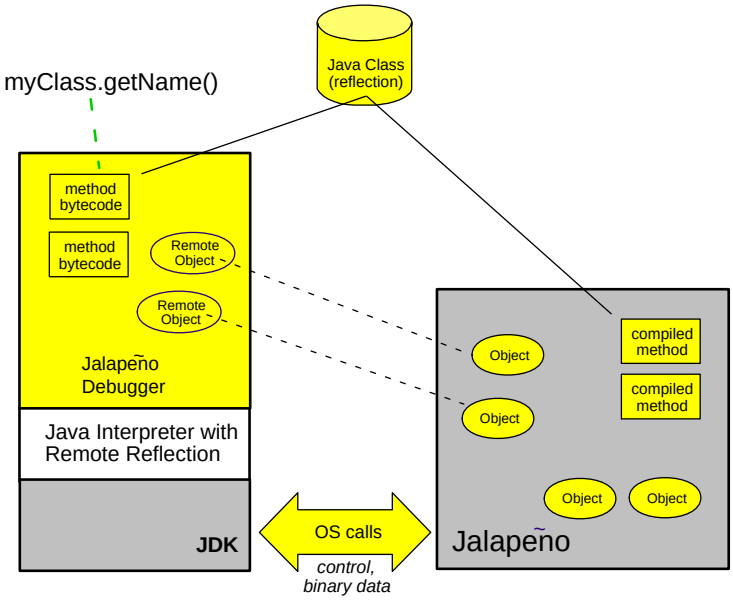

Figure 3: Implementation for Jalapeño: (1) a Java interpreter is extended to support remote reflection, and this in turn runs on top of the Sun JVM; (2) Jalapeño loads and runs the reflection methods as compiled code; (3) the debugger loads and runs the reflection methods as bytecode; (4) remote objects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

in the remote space.

Remote Object

To implement the remote object, it was sufficient to include the type of the object and its real address. Remote objects originate from a mapped method or another remote object. In the first case, the address is provided to the interpreter from the process of building the Jalapeño boot image [2]. For the latter case, the address is computed based on the field offset from the address of the remote object.

For a DejaVu tool to access native methods (on the tool JVM), the JNI implementation (again, on the tool JVM) will have to be extended to handle remote objects. However for our debugger, it proved sufficient to clone the remote objects and the remote arrays of primitives. (Note that this is a separate issue from being able to replay native calls in the *application* JVM.)

Bytecode extensions

Since the initial remote object is obtained via a *mapped* method, the bytecode *invokestatic* or *invokevirtual* to invoke a method are extended as follows. The target class and method are checked against the mapping list. Those to be mapped are intercepted so that the actual invocation is not made. Instead, if the return type is an object, 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 includes 23 byte-

codes. If the result of the bytecode is a primitive value, the interpreter computes the actual address, makes the system call to obtain the value from the remote address space, and pushes the value onto the local Java stack. If the result is an object, the interpreter computes the address of the field holding the reference, makes the system call to obtain the field value and pushes onto the Java stack a new remote object with the appropriate type.

4 GRAPHICAL USER INTERFACE

We have built a Java Swing GUI for the debugger to facilitate its use. The classes which provide the core debugger functionality must be run through the tool JVM to handle their use of remote reflection. However, a Swing GUI would suffer significantly in performance if its classes were also interpreted. Furthermore, the researchers working on Jalapeño typically execute the virtual machine through remote login from a Windows box since both the application JVM (Jalapeño) and the tool JVM run on AIX. However, there is a nontrivial performance penalty for running a GUI on a remote machine (AIX) and displaying it on another (Windows). Therefore, our GUI is designed to be able to run on yet a third JVM and communicate with the debugger JVM through TCP. (In this approach small packets of data, rather than large graphical images, are transmitted between the two machines so bandwidth is not as much of an issue). Our design allows developers to run the debugger remotely while running the GUI on their local machine, allowing for satisfactory performance and easy integration.

The GUI provides all the functionality found in the command-line debugger along with some additional features typically seen in graphical debuggers. A view of the Java source and machine instructions for the currently executing method facilitates setting breakpoints and stepping through the user program. The user can inspect the state of objects and the static fields of classes through a tree-based class viewer. The GUI also provides views of current breakpoints and the call stack linked to the corresponding Java source code. A thread viewer allows the developer to easily track the state of all running threads, aiding greatly in finding subtle bugs in multithreaded applications.

5 RELATED WORK

Repeated execution is a widely accepted technique for debugging and understanding deterministic sequential applications. Repeated execution, however, fails to reproduce the same execution behavior for non-deterministic applications. Replaying a non-deterministic application requires generating enough traces to reproduce the same execution behavior.

Many previous approaches for replay [7, 11, 9] capture the interactions among processes — i.e., critical events — and generate traces for them. A major drawback of such approaches is the overhead in time and, particularly, in space in capturing critical events and in generating traces.

To reduce the trace size, *Instant Replay* [7] assumes that applications access shared objects through a correct, coarse-grained operation called *CREW* (Concurrent-Read-Exclusive-Write), and generates traces only for these coarse operations. Obviously, this approach will not work for applications that do not use the CREW discipline; but it also fails when critical events within CREW are non-deterministic.

Russinovich and Cogswell's approach [10] is similar to ours in that it captures thread switches (rather than all critical events) on a uniprocessor. They modified the Mach operating system so that it notifies the replay system of each thread switch. Since they do not replay the (operating system's) thread package itself, their replay mechanism must instruct the thread package what thread to schedule at each thread switch. This entails maintaining a mapping between the thread executing during record and during replay. This is a significant execution cost that DejaVu does not incur because it replays the entire Jalapeño thread package.

Holloman and Mauney's approach [5, 4] is similar to (and has the same drawbacks as) Russinovich and Cogswell's except for the mechanism to capture the process scheduling information. Their ap-

proach uses exception handlers instrumented into the application code that capture all the exceptions, including the ones for process scheduling, sent from the UNIX operating system to the application process.

Earlier incarnations of DejaVu [3, 6] developed for SUN's JDK running on WIN32 also generate traces only for thread switches.[5] These approaches suffer the same drawbacks as that of Russinovich and Cogswell.

Remote reflection integrates two common debugger features: out-of-process and reflection. Typical debuggers such as *dbx* or *gdb* are out-of-process, but they rely on some 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 also out-of-process and are based on reflection; however, there are several important differences from remote reflection. First, the Sun JDK approach is intended for user applications because it requires the virtual machine to be fully functional. The reflection interface requires a debugging thread running internally in the virtual machine that is dedicated to responding to queries from the out-of-process debugger. In comparison, remote reflection requires no effort on the target JVM. Second, the Sun JDK debugger uses a reflection interface that is different and separate from the internal reflection interface. Although this allows the debugging reflection interface to be implemented in native code to minimize perturbing the JVM, it requires implementing and maintaining two reflection interfaces with similar functionalities. In contrast, with remote reflection the same reflection interface can be used internally or externally.

6 CONCLUSIONS

In this paper, we addressed the problem of building a perturbation-free runtime tool, such as a debugger, for heavily mutithreaded non-deterministic Java server applications cross-optimized with the Java Virtual Machine (JVM). We showed how Jalapeño's design for general extensibility and modularity allows for efficient instrumentation for the application and also for the Jalapeño runtime system.

Cross-optimization of the runtime system and the application can improve the overall performance of the application and the runtime system. Cross-optimization also allows for precise instrumentation for a runtime tool such as DejaVu. Cross-optimization, however, introduces new challenges to program replay due to the side effects of the replay tool that can affect the runtime and the application. We showed how DejaVu employs symmetry in side effects and remote reflection to solve these challenges.

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

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

[4] E. D. 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.

[5] E. D. Holloman and J. Mauney. Reproducing multiprocess executions on a uniprocessor. *Unpublished paper*, August 1989.

[6] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pages 219–228, May 2000.

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

[8] T. Ngo and J. Barton. Debugging by remote reflection. *Proc. of EURO-PAR 2000*, August 2000.

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

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

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