

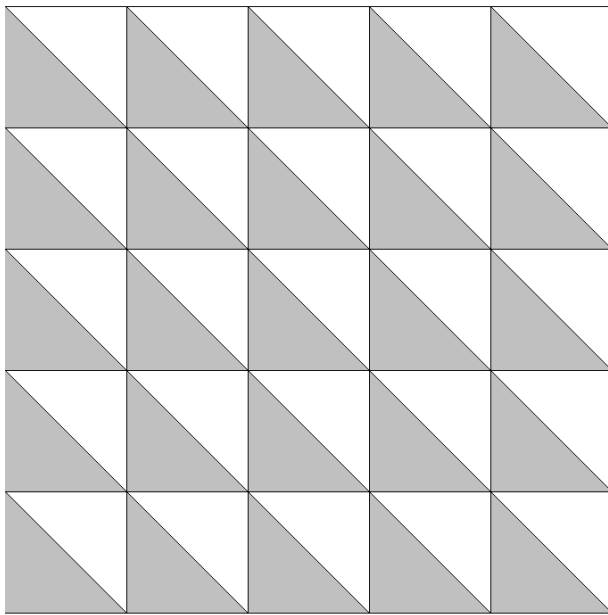
Technical Report

■

IBM Hursley, Hursley Park, UK

A Serially Reusable Java(tm) Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing

TR 29.3406



Sam Borman, Susan Paice, Matthew Webster, Martin Trotter, Rick McGuire(*), Alan Stevens,
Beth Hutchison, Robert Berry

IBM Corporation, Hursley, United Kingdom

(*) IBM Corporation, Cambridge, Mass., USA

ABSTRACT

Transaction processing environments form the backbone of e-business solutions on which much of today's commerce is conducted. The most critical of these environments couple extremely high reliability with performance. Unfortunately, they have done so while retaining a dependency on an aging programming model. Early efforts to introduce Java have been technically successful, but either at the cost of lower performance, or potentially lower reliability. We describe extensions to the Java virtual machine that support the stringent requirements of robust transaction processing environments. The design of a persistent reusable Java virtual machine is presented. Reliability is achieved because each transaction executes in isolation in an apparently unique JVM. Performance is simultaneously achieved through the introduction of new concepts such as the ability to reset the JVM to a clean state, trusted middleware, untrusted application code, and very rapid garbage collection.

ITIRC KEYWORDS

- W** Java Virtual Machine
- W** Isolation
- W** Performance
- W** Transaction Processing

CONTENTS

ABSTRACT

1.0 INTRODUCTION AND MOTIVATION FOR THE PERSISTENT REUSABLE JVM

2.0 OVERVIEW OF KEY CONCEPTS AND THE OPERATION OF THE PERSISTENT REUSABLE JVM

2.1 SERIALY-REUSABLE JAVA VIRTUAL MACHINE

2.2 PRIMORDIAL CLASSES, TRUSTED MIDDLEWARE, UNTRUSTED APPLICATION CLASSES, SHAREABLE APPLICATION CLASSES

2.3 SPLIT HEAPS AND A HEAP-SPECIFIC GC MODELS

3.0 RELATED WORK AND ALTERNATIVE APPROACHES

4.0 DETAILS OF THE DESIGN

4.1 THE ROLE OF TRUSTED MIDDLEWARE

4.2 RESETJAVAVM

4.2.1 RESETTABLE CHECKS

4.2.2 REMOVAL OF APPLICATION CLASS LOADERS

4.2.3 PERFORM RESETGC

4.2.4 REINSTATE CLASS LOADERS

4.3 INFRASTRUCTURE SUPPORTING RESETJAVAVM

4.3.1 CLASS LOADING

4.3.2 WRITE BARRIERS

4.3.3 CONTEXT TRACKING

4.3.4 UNRESETTABLE EVENTS

5.0 PERFORMANCE

5.1 RESULTS/ANALYSIS - RESET PATHLENGTH

5.2 RESULTS/ANALYSIS -THROUGHPUT

6.0 EXPERIENCE AND FUTURE WORK

7.0 ACKNOWLEDGEMENTS

REFERENCES

Introduction and Motivation for the Persistent Reusable JVM

Several years ago there were key challenges barring the introduction of Java as a serious programming environment for distributed transaction processing. In the early days of commercially available Java, the greatest of these was perhaps performance. It was inconceivable that an interpretive Java virtual machine implementation would be able to deliver sufficiently high throughput to sustain realistic transaction rates. Through the efforts of many, this problem has been largely overcome [1] [2] and published performance results suggest that using Java for transactional work might indeed be feasible [3].

Another barrier was maturity of the environment, and the support for a robust programming model. The Servlet model [4] provided a reasonable model for connectors from Web applications to transaction processing environments. The advent of the Enterprise Java Bean (EJB) specification [5], and more importantly, the deployment of now many EJB-compliant platforms [6] [7] has seen a significant increase in the adoption of the Java platform for enterprise transactions. EJBs were the key breakthrough specifically aimed at TP requirements and at providing an abstraction model above “legacy” TP monitors.

However, each of these environments has deficiencies for transaction processing. Servlets are well known for their flexibility - yet they provide little in the way of protection of one transaction, or unit of work, from another. Multiple servlets running within a single JVM can interfere with one another through JVM-wide global statics. Thread-safe programming techniques [8] are now a well known requirement for the servlet application developer.

The EJB programming rules provide substantial isolation, by disallowing much of the antisocial behaviour, but they are not sufficient. Some situations demand isolation to the extent that should the JVM fail, only one transaction at a time is lost. This implies a single JVM for each transaction. Further, we find that not all environments requiring robust Java support for transactions wish to employ the EJB model - we have found that some users require high reliability without the cost of the EJB framework.

Running each transaction in a single JVM reintroduces an old Java concern - performance. Creating a clean JVM for each transaction is very expensive. An alternative is needed.

Herein are the key challenges for the design of the Persistent Reusable JVM: support for stringent isolation of one transaction from subsequent transactions - without undue burden on the application programmer, and delivery of the same with sustained high performance.

The Persistent Reusable JVM design is optimized for short repetitive atomic transactions, where there is no interaction between the transactions, except through persistent storage. However, the transaction processing environments in which it will be deployed also support occasional long-running applications. As the Persistent Reusable JVM is a fully compliant Java implementation, it is capable of running these and any other pure Java application.

Overview of Key Concepts and the Operation of the Persistent Reusable JVM

Our design for simultaneously delivering high performance and high reliability introduces several new concepts into the Java virtual machine. In this section we provide an overview of these concepts, and the motivation behind them. In the next section we discuss related work. This is followed in section 4 with a presentation of lower level details of the design that materialises these concepts. Section 5 describes performance results, and we conclude in section 6 with experience and future directions.

Serially-reusable Java virtual machine

We noted above that a key difficulty with multithreading large numbers of transactions through a single JVM is the lack of isolation that such an environment affords. Isolation can be viewed from several perspectives. The first is interference: Transaction 1 could manipulate some global state (e.g., a global static variable in a system class); transaction 2 will then be exposed to that change. This is undesirable because it can lead to unpredictable behaviour, including failure of the transaction. In the same way, this lack of isolation also introduces potential security problems. A second consequence of such a lack of isolation is lower reliability: if a rogue transaction causes a single JVM to fail, say because of an error in native code library, or runaway transaction consuming large quantities of memory, then all transactions active within that JVM would fail.

We introduce the notion of running transactions consecutively through a JVM, with only a single transaction running at any one time. This approach eliminates some of these problems and limits the impact of others. One transaction cannot interfere with the execution of another transaction running simultaneously in the same JVM. Further, any JVM failure results in the loss of only a single transaction. However, even in a serial reuse model, it remains possible for one transaction to materially affect the behaviour of a subsequent transaction. Therefore, we extend the JVM with the concept of *resetability*, and introduce a new JNI function, `ResetJavaVM`. `ResetJavaVM` resets the JVM state following the transaction so that the next transaction can not be directly affected by the actions of any previous transaction. At a high level, the operation of the `ResetJavaVM` function can be viewed in the context of the simple launcher in Figure 1.

```
jlaunch.c:  
  
{  
...  
options[.] = "-Xresettable";  
...  
}
```

```

rc = JNI_CreateJavaVM(jvm,env,options)
...
while (!resetFailed) {
    ...
    getWork(ClassName, MethodName, args)
    jc = FindClass(env, ClassName)
    jm = GetStaticMethod(env, jc, MethodName...)
    CallStaticVoidMethod(env, jc, jm, args)
    ...
    ExceptionCheck(env)
    ...
    resetFailed = ResetJavaVM(jvm)
}
...
rc = DestroyJavaVM(jvm)
}

```

Figure 1. Sample Persistent Reusable JVM Launcher

A launcher is a C, or other native language, program that creates a Java virtual machine and submits a unit of work (a class, method and arguments) for execution in that JVM. Note that the **java** command present in most Java execution environments is such a launcher. The pseudo code in Figure 1 is illustrative of a launcher intended to operate in the context of a transaction processing environment. The JVM is created in this launcher in the normal manner (see [9]), except the arguments to the `JNI_CreateJavaVM` call have been extended with a new option, **-Xresetable** which designates the JVM as capable of being reset. In this environment a transaction is obtained through a call to a middleware-specific function, `getWork`, that, for example, removes transaction requests from a work queue. The class and method for this transaction are found, again in the usual manner, and then the transaction is executed. At transaction end, the JVM is reset with a call to `ResetJavaVM`. Ideally, the JVM resets without problem, and then the next transaction runs in a *clean* jvm. Note however, that the JVM might be *dirty*, in which case resetting the JVM will fail, the JVM is destroyed, and must be recreated for the next transaction.

The JVM can become dirty, or *unresetable*, for a variety of reasons. Intuitively, these include any activities by transactions that would invalidate our abilities to ensure that one transaction cannot interfere with another. We do not prevent the application from making the JVM dirty; the dirty actions are perfectly legitimate, though antisocial in a shared JVM environment. The effect of them will be to reduce the transaction performance of the overall system, as the dirty JVM is destroyed and recreated. We discuss reasons for a dirty JVM in the subsequent detailed sections.

Primordial classes, Trusted middleware, untrusted application classes, shareable application classes

In the Persistent Reusable JVM, all classes fall into one of three categories - Primordial, Middleware or Application.

Primordial classes, also referred to as System classes, consist of the primitive classes (int, long, char, etc), the classes loaded by the primordial (bootstrap) classloader (Thread, String, Integer, etc.), and any loaded by the Standard Extensions classloader. Any such class will persist for the lifetime of the JVM, as these classloaders are never unreferenced during the lifetime of a JVM.

Middleware classes are typically part of the infrastructure of the application servers which run Java transactions and applications, for example the container for EJBs, or the Java interfaces to IBM's MQSeries. The infrastructure may need to maintain state across transactions, so middleware objects may persist across resets of the JVM. Middleware classes may also legitimately want to set some of the characteristics of the JVM for all the transactions which run in that JVM, or load native libraries to implement some of its function. Middleware is allowed to perform these types of actions without making the JVM 'dirty'. It is assumed that it also resets any transaction-specific state between transactions. For these reasons it is referred to as *trusted* middleware.

Application Classes are effectively 'normal' java which executes a particular business function: either simple classes or Enterprise Java Beans. They may use underlying Middleware to update data resources in enterprise systems such as transaction monitors or databases, or they may be self-contained. Such Application code may be written in house or may be bought in from a vendor. Application code is expected to hold data pertaining only to the transaction being run. It is not expected to have any knowledge of reusability and is not expected to perform any tidying-up of itself. Any classes not determined to be primordial or middleware are deemed to be Application.

Application code is untrusted, and must follow stricter rules than trusted middleware. Not too surprisingly, these rules are similar to those imposed by EJB restrictions [5]. EJB-based restrictions include single-threaded execution, no loading of native libraries, no use of global statics. These rules are discussed later in section 4.3.4.

System class objects and Middleware class objects persist across resets of the JVM. However, application class objects are transient in nature, persisting only for the duration of the transaction. We introduce the concept of 'Shareable' application classes; an installation can identify some or all of the application classes which are likely to be used repeatedly as 'shareable'. The JVM will maintain the bytecodes and compiled code associated with these classes, and re-use them if the same application is rerun within that JAM.

Our design employs distinct class loaders and class paths for each of these types of software. For example, trusted middle ware is loaded by a special loader called a Trusted Middle ware class loader which ensures that corresponding objects are allocated appropriately. The loader designation is also employed at method execution time to allow middle ware methods more freedom than application methods. Application software is loaded from distinct paths by an Application class loader, or a Shareable Application class loader. Application methods have more restrictive checks on their manipulation of the environment.

Split heaps and a heap-specific GCS models

Split heaps separates objects into separate heaps on the basis of expected lifetimes. We exploit this separation by providing a distinct GCS policy tailored to that heap's expected use. The key advantage of this scheme is the a priori separation of transactional data from longer-lived data.

The benefits of this separation are:

- support for a very rapid garbage collection (GCS) of transactional data, without the full cost of a traditional GCS mechanism
- reduced impact of GCS on very long-lived objects by immediately removing them from the scope of GCS

The split heaps, their respective GCS policies and their contents are summarized in Figure 2.

Transient Heap:	Lifetime: short, transactional Allocation rate: high GC: quick clear of transactional data Objects: Non-shareable Application classes Application objects Primordial objects created by Application methods Arrays created by Application methods
Middleware Heap:	Lifetime: medium-long, persistent Allocation rate: medium GC: standard (e.g. mark/sweep) Objects: Objects reachable from the statics of Primordial classes Non-shareable Middleware classes Middleware objects Primordial objects created by Middleware methods Arrays created by Middleware methods Interned strings and the arrays for those strings
System Heap:	Lifetime: long, JVM lifetime Allocation rate: low - mostly startup GC: none Objects:

Figure 2. Split heaps and object allocations

The **System Heap** contains only objects which have a life-expectancy of the life of the JAM. The objects in this heap are the class objects for system and shareable middle ware and application classes. The System Heap is never Garbage Collected as all objects in it will either be reachable for the lifetime of the JAM, or in the case of shareable application classes, have been selected to be reused during the lifetime of the JAM. The System Heap will expand as necessary to satisfy allocation requests.

The **Middle ware Heap** contains objects which have a life expectancy longer than a single transaction, which will persist across resets. Such objects include those non-shareable Middle ware classes, other Middle ware objects, primordial class objects and arrays created by Middle ware code and interned Strings. Allocations into this heap are not expected to be at a high rate. This expected behavior allows for the use of a traditional GC policy. We attempt to collect it only during a `ResetJavaVM` (but not automatically at every `ResetJavaVM`) rather than during a transaction. In order to facilitate this an allocation failure will cause an expansion of the heap (within bounds) rather than performing a GC. GC during a transaction may still occur if necessary, for example if the transaction is long-running.

The **Transient Heap** contains objects with no expected currency beyond the end of an application/transaction. Such objects therefore include only application objects, non-shareable Application class objects and any non-Middleware objects created by Application code. At the end of each transaction, if the allocation policy has been successful, all the objects in this heap should be unreachable. The GC policy most appropriate for this heap is a very rapid reset to the empty state; in the ideal case, a traditional GC involving live root marking is not required. However, for long running transactions, and certain other cases, a traditional GC of the heap may take place.

The objective of very rapid reset of the Transient heap cannot be achieved if there are any references to objects in that heap from more persistent parts of memory. Our design provides mechanisms for tracking cross-heap references, and for facilitating clearing them.

Related Work and Alternative Approaches

The most straightforward approach to providing a highly reliable transaction environment is to create and destroy the JVM with each transaction. The path length of this operation is easily measured with a standard Java runtime, and is on the order of between 20 and 100 million instructions, dependent on the underlying platform. This option is too expensive, since transaction path lengths are typically 1 to 2 orders of magnitude smaller.

Another approach is to checkpoint the JVM state and keep several lying around in a pool of ready-to-go JVMs. To some extent, this technique was employed with the HPCJ technology from IBM [10], and was found to be an effective approach for mitigating the cost of JVM startup in a transactional environment.

An alternative to a high performance reset function is found in JVM Persistence, such as described in [11], and categorised in general in [12]. We chose not to proceed down this path in part because of concerns of platform dependency, and the complexity of making persistence work on one particular platform owing to platform-specific memory management. The present approach can be, and indeed has been, implemented on a wide variety of quite distinct hardware and software platforms. Another concern was performance. We had aggressive performance targets for resetting the JVM and were not convinced they could have been addressed through a persistence mechanism. However, persistence remains of some interest, as it might yet serve as a useful approach to deal with the overhead of bringing a JVM up from scratch.

Perhaps the most significant concern in deploying Java for production transaction processing is the issue of garbage collection. Garbage collection can occur at any time and its impact on the execution of a single transaction can be devastating. One approach uses the JVM for N transactions, where N has been determined a priori to be small enough to (probably) avoid GC altogether, and then discards the JVM entirely. A new JVM, possibly taken from a pool, is then used to serve the next N transactions. This approach works well for a homogeneous mix of transactions, provided N can be made large enough to offset the cost of obtaining and discarding a JVM. There is a risk, however, that this relatively conservative approach may discard the JVM prematurely.

The split heap approach detailed in this design is similar to that presented in generational garbage collection[13]. In generational schemes, Java heap memory is divided into several generations; the first, typically called the nursery, is used to hold very young objects. After some period, these objects are promoted to the older generations. Ultimately, long lived objects are migrated to a very stable portion of the heap and are no longer subject to, and no longer contribute to, regular garbage collection cycles. In one sense, generational schemes are an adaptive version of what we describe here. The Persistent Reusable JVM allows the developer to specify *a priori* the type of objects and their expected life times. Further, the Persistent Reusable JVM provides a level of isolation that generational schemes do not.

The Reusable JVM design here is based on an early prototype introduced by Dillenberger in [14].

Details of the Design

We divide the discussion of design details into three parts. The first part discusses the role of trusted middle ware. The second part addresses what happens at `ResetJavaVM`, which is invoked between transactions to return the JVM to a known, middleware-defined state. Finally we describe changes to the infrastructure supporting JVM reset.

The Role of Trusted Middleware

Middleware is Java code which can be serially reused in a manner similar to the JVM. Unlike application code, it is designed specifically to run in the resettable environment. It may have state which persists between applications, and is responsible for resetting its state between applications, so that an application will not be able to tell whether it is the first or n th application to run in a JVM.

By allowing the middleware to be reused in this way, rather than regarding all Java code as untrusted application code, the overhead associated with processing each transaction is again reduced.

Middleware is also trusted to perform certain activities excluded from applications such as loading native libraries and modifying system properties, without making the JVM non-resettable. It is 'trusted' to change the JVM's state in ways which do not allow one transaction to influence subsequent transactions. This allows it to define a useful and consistent environment for all the transactions processed through this JVM.

Some Middleware objects will be aware of application objects while the application is running. This results in cross-heap references between the middleware heap and the transient heap, which should be cleared when the application terminates. It is the responsibility of the middleware to clear all such references, or the rapid GC of the transient heap will be impacted; keeping cross-heap references to a minimum will improve performance, as well as giving cleaner design.

Two optional static Java methods, TidyUp and Reinitialize, can be provided by middleware classes. If present, they are called by the Persistent Reusable JVM to aid with state management.

The "TidyUp()" method is called during ResetJavaVM() processing for every middleware class that was used in the execution of the preceding transaction. The middleware can perform tasks necessary to ensure a successful reset, such as clearing references to the application heap, as well as cleaning up its own resources. If the middleware cannot clean up its resources adequately, the TidyUp method must return "false"; this will mark the JVM unresettable, and the launcher code will destroy and recreate it.

The Reinitialize() method is provided to allow middleware to prepare itself for a new transaction. Static initializers for middleware classes are run only once when they are loaded. There may however be certain initialization that needs to take place before the start of each application. If supplied a "Reinitialize()" method is called on the first use of a middleware class after each successful reset.

An example of a simple piece of middleware is shown in Figure 3. It can be used for example to load and run an unmodified application program such as "HelloWorld".

```
import java.lang.reflect.*;

public class Middleware
{

    private static ClassLoader loader;

    static {
        ibmJVMReinitialize();
    }

    public static void main (String[] args) throws Exception
    {
        String appName = args[0];
```

```

    Object[] appArgs = { new String[0] };
    Class appClazz = Class.forName(appName,true,loader);
    Method mainMethod = appClazz.getDeclaredMethod("main",
                                                    new Class[] { String[].class } );
    mainMethod.invoke(null, appArgs);
}

private static void ibmJVMReinitialize ()
{
    loader = Thread.currentThread().getContextClassLoader();
}

private static boolean ibmJVMTidyUp ()
{
    loader = null;
    return true;
}
}

```

Figure 3. A simple middleware class

ResetJavaVM

At the end of a transaction the launcher calls ResetJavaVM. ResetJavaVM consists of the following major steps:

- Allow middleware to reset itself, and confirm that the JVM is resettable
- Removal of application class loaders
- Perform ResetGC - the rapid GC of the Transient heap
- Reinstall application class loaders

Resettable checks

First, the JVM must confirm that there are no outstanding exceptions. Exceptions thrown by a Middleware or Application class during transaction execution must be dealt with, possibly by the launcher, else the JVM cannot be reset.

Next, the JVM calls the TidyUp methods of any Middleware classes which have been used since the previous call to ResetJavaVM. For example, the Middleware should release any references to Application objects. If any TidyUp methods return failure, indicating Middleware cannot reset itself to a clean state, the JVM will not be Reset.

At this point the JVM checks that no unresettable events took place during the previous transaction. It also checks that there are no 'user threads' still in existence.

The initial steps, detailed above, pertain to allowing Middleware to clean up and checking whether the JVM is potentially resettable. If that is still the case the actual reset steps begin. If it is not the case, ResetJavaVM returns false.

Removal of Application class loaders

The default application class loaders provided by the JVM are now removed from the class loader hierarchy. Any references held by middleware should have been cleared by the TidyUp methods, if not earlier. This allows any application classes in the transient heap to be collected during the following ResetGC.

Perform ResetGC

During a reset of the JVM, ResetGC is called to quickly reset the Transient Heap so that at the start of each transaction it is always in the same initial state. Logically, the transient heap can be immediately set to its empty state, as it contains objects associated with the transaction which has just completed, and which no longer has any need for the objects. However, correct operation of the JVM requires that

- any references from persistent objects into the transient heap are identified, and
- any finalizers associated with objects in the transient heap are executed

There may be references into the transient heap from local variables. We scan the execution stacks and registers of all system threads and if we find a pointer to the Transient Heap we mark the JVM dirty.

There may be references into the transient heap from the statics associated with the primordial classes. We scan all Primordial Statics in the System Heap and any reachable objects in the Transient Heap are promoted to the Middleware Heap - unless they are a Class object or an Application object, in which case we mark the JVM dirty.

There may be references into the transient heap from the Middleware Heap. To avoid having to do a full scan of all live objects in the middleware heap, we use a Card Table, which has been updated with a dirty flag whenever a reference has been changed. We scan each object in the area mapped by the card and check each reference in these objects. If references to the Transient Heap are found, the JVM may still be clean as the object containing the reference may not be live, so we have to perform a complete check for live references to the Transient Heap (we call this TraceForDirty). If any are found, the JVM is dirty.

Finalizers for application objects are discouraged, but if they exist, Transient Heap finalizer objects will have their finalization method run. As finalizers can create objects in any heap, including objects that themselves have finalizers, we must then loop back to check for invalid pointers to the Transient Heap. This loop will continue until there are no more Transient Heap finalizers. Transient heap finalizers are run on the JVM's main thread of execution, which allows the launcher to ensure that the application finalizers run within the correct operating system environment.

At this point, if the JVM is not dirty, it is safe to reset the Transient Heap state to empty, and to restore it to its initial size.

Reinstate Class Loaders

New application class loaders are now added to the hierarchy and the context class loader is reestablished. The byte-codes and JIT compiled code for shareable application classes are

recovered, and those classes are established as already loaded, but not initialized. On the first use of each shareable class any static initializers are run and static variables reset.

After reinitializing the classloader, any Middleware class which has a Reinitialize method and was used during the previous transaction, is flagged as needing to be reinitialized before it is next used.

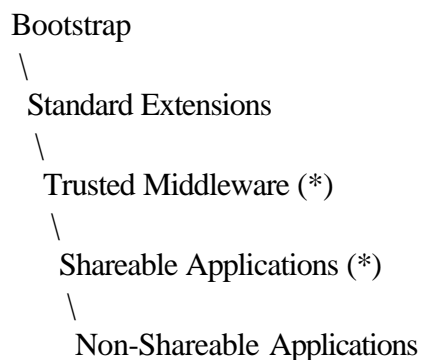
Provided all the steps were completed successfully, ResetJavaVM returns success to the launcher.

Infrastructure supporting ResetJavaVM

Several aspects of the Persistent Reusable JVM infrastructure enable a high performance reset operation (and thus, high transaction throughput). We explain the class loading scheme implemented to achieve the separation of software into Application, Middleware and Primordial Classes. An efficient write barrier mechanism is also put in place to facilitate rapid determination at reset time of the presence of references from long-lived heap space into transient heap. Next we describe the context tracking mechanism used to guide the allocation of objects into the right heaps. Finally we describe the technique employed to police the activity of application code to ensure that no inappropriate actions are taken - we term these 'unresettable events'. The context tracking mechanism is also employed in this determination.

Class Loading

The status of a class within the Persistent Reusable JVM is dictated by its class loader. It is therefore a matter of *configuration* rather than implementation that determines the status of a class. Classes are either designated primordial, middleware or application. The Persistent Reusable JVM augments the existing Java 2 class loader hierarchy to achieve this:



(*) New

Bootstrap or system classes, standard extensions and trusted middleware classes are loaded only once and never unreferenced as their class loader remains active. These classes are loaded into the System Heap and persist across a JVM reset. Bytecodes for Shareable application classes are also loaded only once, but the classes become unreachable at the end of each transaction, and so are reset at each ResetJavaVM, as if they were unloaded and reloaded. Non-shared application classes are loaded for each use of the JVM using the CLASSPATH property and discarded at reset because they are no longer in use.

Middleware and Shareable class loaders are identified by one of a pair of "tagging" interfaces. These can also be used by middleware providers to create their own custom class loaders. A class loader implementing the "Middleware" interface will load classes that are designated trusted middleware. A class loader that implements the "Shareable" interface will load classes that will not be physically reloaded across multiple JVM resets. Byte-code and JIT compiled code are both retained.

As shown above, the implementation provides two shareable class loaders, for middleware and applications.

Write Barriers

As noted above, at ResetGC the JVM needs to determine whether there are any live references into the Transient Heap. This could be established by tracing all live objects from the normal roots, however that would perform no better than a standard GC. In order to speed up the checking process, the Middleware heap is mapped to a 'card table' and any potential reference from the Middleware Heap to the Transient Heap causes the appropriate card in the table to be marked. ResetGC then scans the card table and checks only those objects which map to the marked cards. If those objects do not currently hold any references into the Transient Heap, there can be no references from the Middleware Heap to the Transient Heap.

The cards are marked by 'write barriers' inserted at object slot and array element updates, at element-type setting on array creation and in various classloading and reflection functions as well as in some internal functions such as exception stack trace creation.

Context Tracking

There are two situations within the Persistent Reusable JVM when it is necessary to determine whether a thread is performing Middleware or Application code:

- when allocating an array or an instance of a Primordial class
- when performing any action which would mark the JVM unresettable if carried out by application code

The designation of a thread as Application or Middleware is known as its method-type context.

However, it is often the case that the method being run actually belongs to a Primordial class in which case it is not immediately obvious whether the method-type context is Application or Middleware. In fact, Primordial methods are deemed to be 'workers' and do not have a specific `method_type` of their own so during a method belonging to a primordial class the `method_type` used must be that of the invoking method.

In order to know the current method-type quickly, the method-type is held in the current Frame of the Java Stack and is set/reset as follows:

- if a Primordial class method is invoked, `method_type` is unchanged
- if an Application class method is invoked, `method_type` becomes Application
- if a Middleware class method is invoked, `method_type` becomes Middleware
- At method return, the `method_type` reverts to its value at the time the method was invoked..

Note that a 'method belonging to a class' means the method is actually defined by that class - hence if class A sub-classes class B but does not override method C, C belongs to class B hence the method-type is that of class B even if method C is being run for an instance of A.

Unresettable Events

Applications running within the Persistent Reusable JVM must adhere to a certain set of rules. If they do not, then the JVM is no longer resettable, and the system performance will deteriorate.

The rules are broadly in line with those imposed on Enterprise JavaBeans. At each point in the JVM where an application could break these rules a check has been placed which can mark the JVM unresettable. Many of these activities however such as modifying security settings are valid for middleware code. The context tracking mechanism described above is used to determine the type of caller, middleware or application, and set the JVM state accordingly. Here are some examples of what an application cannot do:

- Modify system properties
- Load native libraries
- Create a new process
- Create a thread
- Redirect standard input/output

In contrast, Middleware code is mostly unrestricted in its activity, but must terminate any threads and tidy up its references to application objects, either in its normal logic flow, or at transaction end, in the TidyUp() method.

If at any time the JVM has been marked unresettable a subsequent call to ResetJavaVM() will fail. To determine the cause of the failure a logging mechanism can be used which creates a text file describing the unresettable events that have occurred. In the case where the check has been made in a Java method a stack trace may be included.

Performance

To determine the extent to which the previously described abstract design features succeeded, a concrete implementation was required. We took an existing JVM implementation with good and well-understood performance characteristics and modified it to include the class categorisation, class loaders, split heaps, middleware and transient heap garbage collection, unresettable events, reset processing, write barriers and context tracking functions. Since the main motivation of this project was to prove that good performance could be achieved in conjunction with transaction isolation, whilst retaining the function and robustness of a complete JVM and JIT compiler, the obvious measures of success were the levels of performance reached. In common with other performance-oriented projects, we allowed ourselves some additional cycles of modification and tuning of the original prototype to demonstrate the effectiveness of lessons learned in coding and observing a concrete implementation.

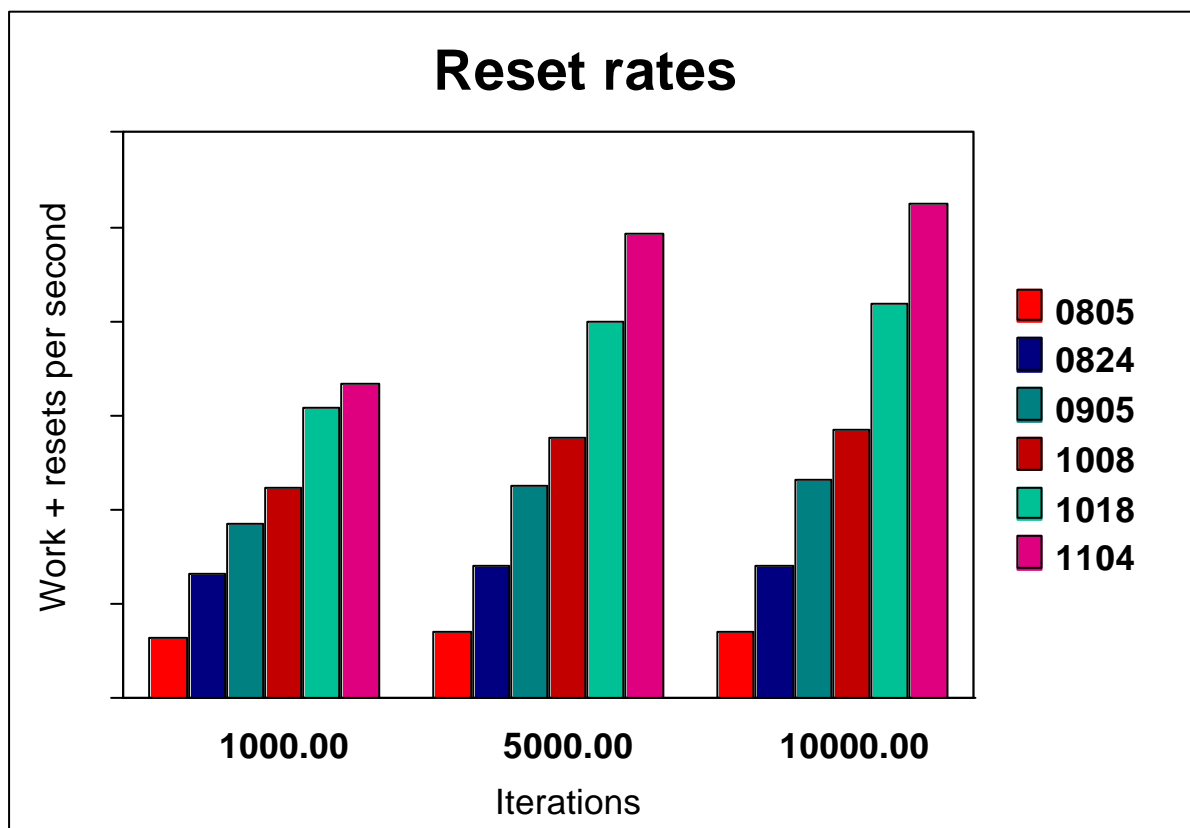
Two main measures of performance were identified. Firstly the cost of returning the JVM to an initial state; for this to be considered proof of the reset concept as viable, a minimal launcher would need to be capable of completing a small piece of Java application code and JVM reset itself at a rate of several hundred or more per second. For the case of more complex, productive work such as

EJBs, the overhead added by running reset would have to be significantly smaller than the duration of the application code.

The second main proof of success was to demonstrate that the overall execution rate of Java code (between JVM resets) was not decreased unacceptably by the extra path length introduced in the JVM and JIT compiler - for example, method context tracking. To determine this, a straightforward comparison between throughput rates in resettable and non-resettable modes using internal and public Java benchmarks was used. The determination of 'unacceptable' decreases in throughput is not so straightforward - it ranges perhaps between 'none', through 'noise' to some specific upper limit. To deliver no net loss in performance would certainly require some compensatory invention that was specifically only applicable to the resettable mode (otherwise it would also boost non-resettable performance). For our purposes, we set the success criterion at 90-95% of the performance of the non-resettable mode when running in resettable mode.

Results/analysis - reset pathlength

The tests shown here were conducted using a system with 3 active processors, though in practice the test runs single threaded. The test case measured the time to perform a simple application followed by a reset, and was used to track progress during successive builds with added optimisations against objectives. The rates were measured by recording the time taken to complete either 1000, 5000 or 10,000 iterations. The increased rates at higher number of iterations are due to greater amortisation of start-up costs (mainly class loading and JIT compilation).



This chart shows that in common with many performance-driven projects, the Persistent Reusable JVM's reset rates have increased very significantly (by a factor of up to 7) and in a series of steps over a short period.

Some factors have proved to strongly influence reset time -

- Jar file manifests. A complex middleware environment can introduce jar files from a variety of sources not entirely under the control of a JVM provider or user, and in our experience policies on the generation and content of jar file manifests vary widely. It was found that simply adding jar files with large manifests to the Persistent Reusable JVM's middleware CLASSPATH significantly increased start-up time and heap footprint as well as reset time. In most cases jar file manifests are not expected to be used in the deployment environments for which the Persistent Reusable JVM is designed.
- Scanning of references from the execution stacks and registers of the system threads and user thread into the Transient Heap.
- Running TraceForDirty - see section 4.2.3 for a description.

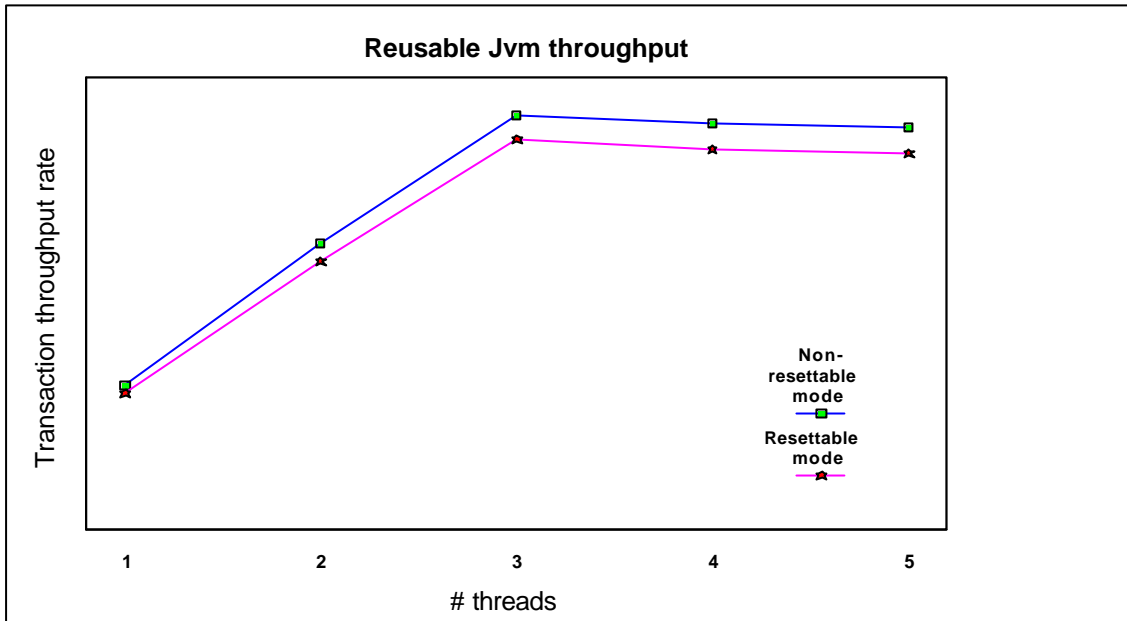
These results show a path length comparable to a trivial transaction in this environment. The overall path length for more functional applications will be affected by middleware and application logic, and measurements and performance improvements are ongoing.

Results/analysis -throughput

The tests shown here were conducted using a system with 3 active processors running a server benchmark in resettable and non-resettable modes.

Comparison of the chart data shows -

- In resettable mode, the Persistent Reusable JVM prototype gave 94% of the performance of the non-resettable mode. This was consistent at all measurement points. The expected use of the reusable JVM would be with a small number of application threads - typically one.
- The Persistent Reusable JVM prototype builds show similar scalability characteristics in both resettable and non-resettable modes.



Experience and Future Work

One challenge in this design has been to contain occurrences of non-application objects holding references to transient object(s) at reset time. While some responsibility for this lies with Middleware (e.g., through `TidyUp()` methods), we have found situations in which `Primordial (System)` classes, when run by Middleware, have a side effect of creating such links. For example, a common usage is for Middleware to create a `java.lang.reflect.Method` object to refer to the method of an Application class that is to be invoked. Following execution of the Application method, the `java.lang.reflect.Method` object is nulled, making it available for collection at the next Middleware GC. However, reset processing finds the reference from the Middleware heap into the Transient heap and forces a complete Mark phase to ensure there are no *live* references to the Transient Heap (Trace For Dirty). The JVM is found to be clean, but the damage has been done in that we have effectively performed a complete (and expensive) GC. We address this by ensuring that any `java.lang.reflect.*` objects created to refer to Application classes are always allocated in the Transient heap, preventing such cross-boundary references from occurring.

Another challenge lies in the relatively rigid allocation model that compels objects created by trusted Middleware to be allocated in the Middleware heap. A concern with the current model is that we may sometimes violate the assumption that frequent GCs for the Middleware heap can be avoided. Therefore, we are exploring an adaptation that explicitly considers Middleware from two perspectives. The first views Middleware as being memoryless and utility oriented (e.g., to invoke a method over RMI/IIOP). In this case, objects allocated by Middleware should be allocated in the Transient heap, so they can be quickly removed at transaction end. The second supports a more persistent role (e.g., Middleware creates and persists buffers or pools of objects for use by Applications). In this case, objects allocated by Middleware persist over `ResetJavaVM`, remaining in the Middleware heap.

Our experience with this design has been very valuable. We have demonstrated that it is possible to create a reusable JVM that is capable of delivering high performance transaction processing support. More work is needed, however, before this capability can be deployed in a highly scalable manner. In particular, it is essential that a JVM sharing scheme be employed in order to reduce overall footprint. Since each JVM instance runs one transaction at a time, it is necessary to have multiple JVMs running in parallel to deliver high levels of throughput. We have extended our design to include sharing of classes, class metadata and JIT-compiled code. We defer discussion of this extension to a subsequent paper.

Acknowledgements

Much of this work was based on an early prototype [14]. We are grateful to those designers, and in particular to Donna Dillenberger, Donald Schmidt and Alan Webb who worked closely with us on the high-level design of this project. We are grateful to James West for his personal contributions to the design of the Persistent Reusable JVM, and most importantly for providing and supporting the environment in which it was created. Thanks also to Tim Banks, Kean Kuiper, Simon Nash for their help with design issues and resolution. We owe much to Pete Hickson for work on performance. Thanks also to Graham Rawson, Paul Harris and John Burgess for performance measurements and their understanding of production transaction processing. Thanks to Mike Oliver, Bob Dahle and John Rankin for their comments and assistance. Special thanks to the development team, both in Hursley and Haifa, and to their and our management, Sue Collis, Hillel Kolodner and Mark Thomas.

REFERENCES

- [1] Osvaldo Doederlein, The Java Performance Report, September, 2000. Available at, <http://www.javalobby.org/features/jpr>.
- [2] IBM Systems Journal special issue on Java Performance, Volume 39, No. 1, 2000.
- [3] Standard Performance Evaluation Corporation, SPEC JBB2000, Java Business Benchmark, 2000. See, <http://www.spec.org/>
- [4] The Java Servlet API, Sun Microsystems. See: <http://java.sun.com/products/java-server/documentation/webserver1.0.2/servlets/api.html>
- [5] Enterprise JavaBeans Specification 1.1, <http://java.sun.com/products/ejb>
- [6] BEA Weblogic, <http://www.weblogic.beasys.com/products/>

- [7] IBM Websphere Application Server, <http://www.ibm.com/software/webservers>
- [8] D. Flanagan et al, Java Enterprise in a Nutshell, O'Reilly, September 1999.
- [9] Rob Gordon, Essential JNI Java Native Interface, Prentice Hall PTR, 1998.
- [10] V. Seshadri, "IBM High Performance Compiler for Java", AIXpert Magazine, <http://www.developer.ibm.com/library/aixpert/> (September 1997).
- [11] Jon. Howell, Straightforward Java Persistence Through Checkpointing, Technical Report PCS-TR98-330, Dartmouth College, May 2, 1998,
Available at <http://www.cs.dartmouth.edu/reports/abstracts/TR98-330/>
- [12] J. Eliot B. Moss and Tony L. Hosking. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java, September 1996*.
- [13] R. Jones and R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996.
- [14] D. Dillenberger et al., Building a Java virtual machine for server applications: The JVM on OS/390, IBM Systems Journal, Vol 39, No. 1, 2000.