

Gosling J., Joy B., & Steele G., *The Java Language Specification*, Addison Wesley, 1997.

[Malhotra]

Malhotra, A., & Munroe, S.J., Support For Persistent Objects: Two Architectures, *Proc. 25th Hawaii Intl. Conf. on System Sciences*, 1992,

[Rosen1]

Rosenberg, J., & Abramson, D.A., MONADS-PC: A CapabilityBased Workstation to Support Software Engineering, *Proc. 18th Hawaii Intl. Conf. on System Sciences*, 1985, pp222-231

[Rosen2]

Rosenberg, J., Koch, D.M. & Keedy, J.L., A Massive Memory Supercomputer, *Proc. 22nd Hawaii Intl. Conf. on System Sciences*, 1989, pp 338-345

[TPC-C]

Transaction Processing Performance Council, TPC Benchmark(TM) C, Standard Specification Revision 3.2, 27 August 1996, URL: <http://www.tpc.org/bench.descrip.html>

Windows95. The difference is explained by a combination of faster processor cycle times and the smaller database fitting entirely in the available DRAM. When running in a three warehouses configuration on the 166MHz system, measures rise to the 116-124 tpmBOB-C range. This shows a relatively graceful degradation for overcommitting real memory.

## 4.0 Conclusion

While these results are early and incomplete they demonstrate that SLS techniques are applicable to 32-bit standard platforms. Combining SLS technology with Java VM technology provides both an effective and highly portable persistent object model. The same persistent object model is a natural fit for 64-bit systems supporting Java and persistent virtual storage (SLS).

Clearly, the JavaSAS interpreter has to do more work than a standard, transient-only JVM and would not be suitable for an application that does not require persistence. But for commercial applications that require some form of persistence JavaSAS performs acceptably on 32-bit systems while providing a superior programming model.

While our performance claims are modest to date (120 tpmBOB-C Vs 1000 tpmC), it is early yet with many potential improvements untapped. Our JavaSAS VM needs additional tuning especially in the structure of the class image. Since we use 16-byte pointers within the structure of the loaded class image, any unnecessary levels of indirection add significant overhead. Additional hand tuning of the Bytecode dispatch loop is also warranted, including a hand turned assembler implementation. Other important areas to be explored include JIT technology and compilation to native code.

Of course the key comparison should be with other Java object persistence mechanisms. While there are a number of products and published papers on Java persistence, we could not find any published results comparable to TPC-C (or BOB-C). It is important to agree on an appropriate benchmark so that results are directly comparable.

## 5.0 Bibliography

[Andrews]

Andrews, D.H., IBM's San Francisco Project: Java Frameworks for Application Developers, *D.H. Andrews Group, 700 West Johnson Ave., Cheshire, CT 06410*, Dec. 1996

[AS/400]

IBM Corporation, *IBM Application System/400 Technology Journal (Version 2)*, 1992

[Chase]

Chase, J. S., Levy, H. M., Baker-Harvey, M., & Lazowska, E. D. (undated), Opal: A Single Address Space System for 64-bit Architectures, *University of Washington, Department of Computer Science and Engineering*

[Gosling]

## 3.2 Measured Performance

Performance is difficult to measure objectively. The goal would be to directly compare fully configured BOB-C measurements to the equivalent TPC-C on the same hardware configuration. Due to limited resources, we were not able to configure BOB-C object populations equivalent to the published TPC-C configurations.

For example, 166MHz Pentium systems with unconstrained (512+MB) DRAM and large (10+GB) RAID storage configuration have published results over 1000 tpmC(TM)<sup>2</sup>. Such a result implies 80 to 110 terminals and requires ~100 TPC-C warehouses to qualify. Our 166Mhz Pentium desktop system had 80MB and 1.7GB of disk space. This configuration was large enough to build a single full scale warehouse or several fractional (1/10th and 1/100th) scale warehouses. The 80MB of DRAM supported a 16MB shared storage pool and 40 instances of the JVM process (simulating 40 independent users/terminals).

So far we have built and measured a single full Warehouse configuration on AIX, Windows95, and WindowsNT. We have measured 11.5 to 12.6 tpmBOB-C running 10 JVM processes each in a separate command prompt (or X-Term) window. This is not very exciting, but right on the money for one TPC-C Warehouse, ten terminals, and terminal wait/think times per specification. A TPC-C run must perform at least 9 tpmC and not more than 12.7 tpmC per warehouse. For higher tpmC (and tpmBOB-C) numbers, more warehouses and terminals would need to be added to the configuration.

We built four 1/10th scale Warehouses and ran 40 JVM processes/windows. We measured 44.6 to 49.8 tpmBOB-C in this configuration on the AIX and WindowsNT platforms -- again, within the qualifying range with nominal wait/think times.

At this point it was clear that our system's DRAM would not support the number of JVM processes required to run larger Warehouse/terminal configurations. In an effort to measure the "intrinsic" capacity of our technology, we added a benchmark option to run multiple threads within a single JVM process and set the wait/think times to zero. This allows the JVM to run unconstrained but forces all terminal output to a single window. Since scrolling large volumes of text is a significant load on windowed systems, we added a option to disable the display of transaction screens. In this case the screen image is built in memory, but the final display write is not called.

This mode allows the persistence storage mechanism to push to the maximum rate for that CPU. This option also serves as a stand-in for a client/server configuration where the screen display load would be distributed over a number of client systems. In this mode we measured 98.3 tpmBOB-C running 4 threads over 4 1% scale warehouses, on a 166MHz Pentium with 80MB DRAM under WindowsNT 4.0. Similarly we measured a 96.4 tpmBOB-C with screens enabled and 176 tpmBOB-C with screens disabled running 1 thread over a single 1% scale warehouse, on a 200Mhz MMX Pentium with 64MB DRAM under

---

2. \*Transactions per minute benchmark C, based on TPC-C

- Each Warehouse, District, and Customer has an associated **Address**

This allows direct comparison between the RDB storage requirements of a TPC-C Warehouse and persistent Java implementations with various reference sizes. The resulting storage requirement for a fully populated BOB-C Warehouse are:

- The Base (RDB equivalent) is 77MBs
- Java object equivalent with 32-bit references requires 245MBs (a 219% increase over the base)
- Java object equivalent with 64-bit references requires 267MBs (a 247% increase over the base)
- Java object equivalent with 128-bit references requires 350MBs (a 356% increase over the base)

Note the 200+% storage increase for the 32-bit Java equivalent. This reflects: the additional object runtime overhead (i.e. object handles), use of `java.lang.String` to replace fixed and variable text, use of Unicode to replace ASCII, use of `java.util.Date` to replace system date and time and additional object references to avoid some lookups. The biggest contributors are: Unicode, Java Strings and Java Dates. ASCII strings would save approximately 62MBs and implementing Date as a simple long would save approximately 18MBs.

However, a 32-bit persistent object reference does not support the scale required for a large business application using SLS technology and is not a valid comparison. A 32-bit JVM would be forced into a Two Level Store (TLS) implementation. TLS implementations do not need to store the complete Java structure, but object references are more complicated. For example, only the characters of String need to be stored, but the Java national language support strategy implies that Unicode should be preserved.

A TLS implementation tends to have fat persistent object references to support polymorphic types and to simplify object activation and distribution. TLS persistent object references require a class ID in addition to an object ID or key. This allows an empty object instance to be created and activation (stream internalization or schema mapping) to be delegated to the object implementation. In a distributed environment, a database or node ID is required in addition to object ID and class ID. This allows function shipping requests to be routed to a server without resolving the object identity to its storage location. For open systems environments, these IDs are usually defined as 16-byte (128-bit) DCE UUIDs.

If you apply these assumptions (32-byte primary keys and 48-byte persistent object references: 16 bytes for the Class, 16 bytes for the object and 16 bytes for the server) to the spreadsheet, the storage required for a 32-bit TLS implementation jumps to 267MBs--a 247% increase over the base and only 31% less than the JavaSAS equivalent. The net: the 16-byte pointers are not the largest impact on the storage footprint -- object infrastructure, String and Unicode are -- and the result compares favorably with the footprint of many functionally equivalent TLS implementations.

We also decided to use a page-based shared memory pool for currently accessed persistent storage pages. This implied that large objects that cross page boundaries would be discontinuous in the page pool. We also decided that references on the operand stack and local variables should be 32-bits. We resolved this problem by enforcing a level of indirection between the operand stack and object fields. Thus, the implementation for each getfield/putfield bytecode must perform the following actions:

- load the current address of the object into the page pool from the reference look-aside
- add any offset to this address
- and check for page boundary crossings
- load/store the target field
- loads/stores of object references must also translate 128-bit references to 32-bit addresses in the page pool and back again

The design decisions above make the JavaSAS object instances larger and object field references slower than the equivalent standard JVM. So the question is “how does the performance and storage size compare to a more traditional (two-level store) persistence model?”. In the following sections we present some informal back-of-the-envelope comparisons.

### 3.1 Storage impact of large pointers

The first observation is that while a large pointer size (16-bytes vs. 4-bytes) does impact the “foot print” of persistent objects, two other changes have a more significant impact. First, Java Strings are stored as 16-bit Unicode characters. Second, persistent Java objects include Java storage handles and the full java.lang.String object structure (not just the characters).

To test this we simply plug in the population requirements of the BOB-C (TPC-C) specification, the field sizes of the various BOB-C objects, and internal object structure requirements of Java into a spreadsheet. A BOB-C warehouse requires the following object population:

- 1 **Company** and 100,000 **Items** independent of the number of Warehouses
- 1 to N **Warehouses**, where each has:
  - 10 **Districts**
  - 100,000 **Stock** corresponding to the Companies Items
  - 30,000 **Customers**
  - 30,000 initial **Orders**
  - 300,000 initial **Orderlines**
  - 9,000 initial **NewOrders**
  - 30,000 initial order **Histories**

the environment (`SecurityException`). Java programmers are not expected to program for these exceptions but can (via `try/catch`) if they want to. Just plain `Exceptions` are part of the API and must be explicitly handled (via a `catch` clause) if the interface specifies the exceptions (via `throws` on the method declaration). `IOException` is an example of this class.

The JavaSAS persistence model introduces some new conditions to be handled:

- `ObjectDestroyed` -- The object no longer exists. The runtime detection of a dangling reference.
- `ObjectAccessDenied` -- a form of `SecurityException`, The object is read-only and the method attempted a store, or an attempt was made to call a method on an object the client has a reference to but is not authorized to.
- `StorageReadFailure` -- The object exists but the Java/SAS VM can not read the media containing the object. This may be a permanent read error (damaged media) or a temporary condition (the link to the remote object storage is down).
- `StorageWriteFailure` -- The object exists but the Java/SAS VM can not write to the media containing the object. This may be a permanent write error (damaged media) or a temporary condition (the link to the remote object storage is down).

The Container abstraction introduces more conditions in this category, including:

- `ContainerFull` -- The physical storage space allocated to this container heap is exhausted.
- `ContainerDamaged` -- The container is in an inconsistent state and requires a recovery action before it can be used.

These exceptions all seem to fit into either the `Error` or `RuntimeException` classification. This relieves the Java programmer from explicitly programming for these conditions and the system default action (terminating the thread) is appropriate in most cases. However, a Java class that implements recoverable resources (like a journal or transaction manager) can catch and program recovery actions.

In the current prototype we have implemented `SASException` and `ContainerException` as subclasses of `RuntimeException`. All exceptions introduced by the JavaSAS package or the JavaSAS VM are subclasses of `SASException` or `ContainerException`.

### 3.0 Some Observations

For this implementation we had to make some fundamental decisions. The first such decisions was that temporary and persistent instances should have the same format and share a single class image. This required that reference fields be the same size and alignment for temporary and persistent object instances. From our experience with the AS/400, we chose a reference size of 128-bits (16-bytes) with a 104-bit Virtual Address.

mers are not SAS aware. The SAS unaware programmer will naturally use the simple **new** operator to create sub-objects. The result would be a shared persistent object containing references to local temporary objects, which may not deliver the desired result.

For example, `java.lang.String` and `java.util.HashMap` are both composite objects. Strings are used pervasively so we implemented special native Container methods (**allocStringNear()**) to insure both the String and the internal char array are created at the same persistence and scope. We could not afford to write special Container methods for each interesting class so we simply avoided using these classes for persistent objects.

Now that we have this working and understood it may be appropriate to pursue class library and/or language changes with JavaSoft. We would like to leverage the larger world of Java programmers who are not necessarily persistence aware. To achieve this we need to make this programming model pervasive in the industry as a simple extension to Java.

Being able to extend `java.lang.Class` would be very useful. It already supports **newInstance()**. It should be simple to add **newPermInstance()**, **newInstanceIn()**, **newInstanceNear()**, **newInstanceWith()**, and **newArrayOfNear()** to the runtime implementation of Class. This implies that the Container abstraction and a default (temporary, local, garbage-collected) are introduced and shipped with the JDK. `java.lang.Object` would also need to be extended to support the **destroyInstance()** and **getInstanceContainer()** methods. We have used this framework extensively across several projects and platforms and believe that it is generally applicable and complete. Similar Container and new-near abstractions are integrated into the San Francisco Business Object Framework design [Andrews].

## 2.5 Exceptions

The single level store paradigm is unique in that it provides access to persistent storage via machine level (load and store) instructions. The traditional file based persistence model requires explicit calls to system APIs. But what happens if the requested operation cannot complete? The file system returns a status variable that reflects the success or failure of each operation: open, close, read, or write. A SLS system supports the equivalent of read and write at the machine instruction level which makes a return code strategy impractical. Instead an exception strategy must be used.

This is a good fit for Java since Java supports and makes extensive use of exceptions. Java supports exceptions in three classes:

- Errors
- Runtime Exceptions
- Exceptions

Errors and `RuntimeExceptions` are normally thrown directly by the Java VM. Errors reflect conditions where the requested resource is: not valid (`VerifyError`), inconsistent (`NoSuchFieldError`), or VM has run out of resources (`OutOfMemoryError`). `RuntimeExceptions` reflect a runtime violation of some constraint (`IndexOutOfBoundsException`, `ClassCastException`), arithmetic error (`ArithmeticException`) or the security policies of

- **newInstanceWith(Class classRef, Object nearRef)** - For generic programming of composite objects. Create an instance into the same Container as the existing nearRef.
- **newInstanceNear(Class classRef, Object nearRef)** - For generic programming of composite objects with a strong suggestion for physical placement. Create an instance into the same Container and as close as possible (same page) to the existing nearRef.

To complete the persistence framework we need some additional methods associated with existing persistent objects:

- **destroyInstance()** - Used to destroy a persistent object instance. Redirects to a deallocate method on the appropriate Container.
- **getInstanceContainer()** - Returns a reference to this instance's Container. Used by the implementation of newInstanceWith, newInstanceNear, and destroyInstance.

The Container allocate methods are implemented as native methods integrated with the Java VM. These native methods allocate storage from the Container heap, initialize the Java object handle, invoke the default constructor for the class, and return a reference to the new instance. Since the Container heap was created in persistent/shared SAS storage, any contained object instance is persistent and shared.

This is an effective work around for the lack of metaclass programming and provides a convenient interface for additional **new** operations. We were able to create persistent versions of simple (non-composite, non-array) classes via the Factory methods **newInstanceIn()** or **newInstanceNear()**. For example, we used java.util.Date extensively in our TPC-C implementation. However arrays and existing complex (composite) java class libraries are still awkward and require special Factory methods or arbitrary restrictions.

Neither the Java (1.0.2) language nor java.lang.Class provides a mechanism to programmatically create arrays of an arbitrary type. Arrays of primitive types were handled by writing a special method for each primitive type, such as **newLongArrayNear()**, **newDoubleArrayNear()**,... and implementing specific native Container methods to support them. Arrays of Java Classes required a special **newArrayOfNear()** method with special native method support. **newArrayOfNear()** accepts a (non array) Class reference for the base type and number of elements as arguments and returned an Object[] that has to be down-cast to the correct type

Another problem concerns the management of compound objects. A compound object is composed of several sub-objects linked by object references. For example, each Person object may reference an Address object. Or simpler yet, any object containing an array, since in Java all arrays are objects. For example, a Person object includes first and last name fields stored as an character arrays.

The SAS aware Java application will use **newInstanceNear()** to create any sub-objects and arrays. This insures that the sub-objects have the same persistence and scope as the parent object. However the current JDK class libraries and the majority of Java program-

## 2.3 JavaSAS Class Loader

The JavaSAS Class Loader takes a standard Java class file and loads it into SAS storage. A SAS loaded Java class is (persistently) activated in SAS storage and is persistent, shared and paged like all SAS storage. A SAS loaded class is capable of supporting both temporary and persistent Java object instances.

This impacts the Java class dictionary (which tracks Java class loads and is used to bind class implementations). The class dictionary must be at the same persistence and scope as the classes themselves, so a class name to class address map is created and maintained in SAS storage.

## 2.4 Java Lifecycle Extensions

Currently Java supports only temporary local objects via the **new** operation. For backward compatibility we preserve the function of **new** unchanged. Applying the **new** operation to a SAS loaded class returns a reference to temporary object in the local Java environment. For SAS loaded classes we add operations to create persistent objects in SAS storage. Once a persistent object instance is created, it behaves exactly like a temporary object, independent of its persistence scope.

To locate related persistent objects close to one another in the vast SAS address space, Java developers need a containment/placement abstraction -- put this object over there! -- which we call Containers, and additional **new** operations. The **new** operations work with the container abstraction to insure that objects instances are created into the correct persistence scope and physical location. This raises some design challenges since the Java language and runtime time does not support metaclass programming. The Class class can not be subclassed! The two possibilities are:

- Generic Factory Class
- Extend java.lang.Object and java.lang.Class

The current prototype supports persistence via a Factory class. The Factory provides static methods to create instances of the specified class. The class is specified as either a (package qualified) class name string or a Class reference. The Factory methods delegate the work of instance creation to either java.lang.Class (newInstance method for temporary objects) or the appropriate Container instance. Containers are special classes with native methods for storage allocation.

The factory methods we have in our prototype are:

- newPermInstance(Class classRef)** - Used primarily to create Containers and Container Heaps.
- newInstanceIn(Class classRef, Container containerRef)** - To support placement of new objects for locality of reference or security reasons. Creates an object instance into the specified Container.

The load/store bytecodes are exposed to the (hopefully) rare case of object instances or arrays spanning page boundaries. The SAS simulator does not guarantee that adjacent SAS pages are contiguous in the page pool. While the Java VM runtime should take care in space allocation to avoid page crossings, eliminating page crossing is impossible without imposing draconian restrictions. Since we wish to support large (>4KB) objects and arrays, we handle loads/stores beyond the first (SAS) page referenced. The put/getstatic bytecodes are not impacted because copy-on-write storage is always allocated contiguously.

Page crossing impacts all bytecodes that take an object reference as an operand and load (get) or store (put) data or access the class. If the object is temporary and local, then no further action is required (since local/temporary objects are allocated contiguously by the runtime). Otherwise, if the object is in SAS storage and the field is not in the first page of the object, then (logically) the field/entry offset must be added to the base SAS address and translated to an offset in a valid page within the page pool. The SAS Simulator optimizes this case by chaining secondary SASRefs off the primary SASRef.

In our current implementation the total (excluding quick variants) number of affected bytecodes is 35.

- ldc1, ldc2, ldc2w
- iaload, l,f,a,b,c,s,d
- iastore,l,f,a,b,c,s,d
- get/putstatic, get/putfield
- invoke virtual/nonvirtual/static/interface
- new, newarray, anewarray, multianewarray
- arraylength, athrow, checkcast, instanceof

Bytecodes that use object references to invoke methods, reference object fields, or check casts must deal with the internal details of Java handles. In the JavaSAS VM implementation, handles contain the (128-bit) SAS addresses of the instance data and class. The get-field bytecode translates the SAS address of the Java Object handle, but the translation of the instance data and class address is delayed until the handle is used. For invoke (method) bytecodes we translate the instance data and class SAS addresses to SASRefs immediately and store them in the called methods frame. This insures that invocation and field references to these are preresolved.

Invocations and field references using another object reference (not “this”), must resolve the instance data and/or class SAS addresses each time. We are looking at a combined SASRef and local handle to allow instance data and class SASRefs to be cached in the handle. The down side is that it would muddy the line between the Java VM and SAS.

Java class static data fields require special handling. Static fields are scoped to the Java VM instance (static is shared by all threads within a VM but not between VM instances). SAS supports this semantic by simulating copy-on-write (COW) storage. When a Java class is loaded into persistent storage static fields are allocated as separate page(s) within the activated image, initialized, and then persistently marked as a COW page(s). COW pages are not read into the shared page pool but instead are copied directly into the local heap of the VM. SASRefs to COW storage point directly to the local Java VM instance heap and are managed as a VM wide resource

Synchronization also requires attention as it is integrated with Java language semantics. Synchronization must have the same scope as the object instance. For JavaSAS we co-located locks with the objects in the shared persistent store. Specifically, “test and set” locks were stored with the persistent object handles and used an operating system “sleep” for back-off. While this is crude by any standard, it did allow Java applications to use Java language synchronization for both local and shared objects. For a production system we would provide a shared memory lock with thread control.

## 2.2 Java VM

We modified the Java VM so bytecodes that reference object (instance, static, and array) storage can support SAS addressing. When a SAS reference is loaded (by the getfield bytecode) they are passed to the SAS Simulator for translation. The current strategy is to store the full (128-bit) SAS address in objects and translate them to short (32-bit) references for the operand stack and local (automatic) storage.

This partitions the Java bytecode into four categories:

- Bytecodes that are unaffected
- Bytecodes that load or store object references
- Bytecodes that load or store from persistent objects
- Bytecodes that reference the loaded class structures

Most bytecodes operate on simple data types on the operand stack or local storage or local branches and are unaffected.

Bytecodes that load (128-bit SAS address) object references will need to translate to local (32-bit) references. Bytecodes that store (SAS address) object references will need to perform the reverse translation (32-bit local to 128-bit SAS). This impacts the following subset of the Java bytecodes:

- field puts and gets of object references (putfield and getfield byte codes)
- object reference array loads and stores (aload and astore byte codes)
- all persistent array loads and stores (Xaload and Xastore byte codes)
- static puts and gets of object references (putstatic and getstatic byte codes)

VM instances. To translate a SAS virtual address to a 32-bit process-local address the SAS client first searches a (thread scoped) TLB hash table, then searches the (shared) inverted page table. If the SAS client cannot translate a SAS address the page request is sent to the SAS server.

The SAS server will locate the page on backing store, read it into the shared page pool, and update the shared page translation table. If the requested page does not exist the SAS client is notified and the client throws an address exception. The SAS server manages the page pool LRU, monitors reference and dirty page information, and insures that dirty pages get written out.

SAS addresses are not used on the Java runtime stack (operand stack and local variables). The Java runtime stack is always local to the Java VM instance and accessed via 32-bit addresses. Object references on the stack are 32-bit pointers to TLB entries (which we call SASRefs). All SASRefs can be referenced by their 32-bit address or looked up in the Local Hash Table using the corresponding SAS address. Each SASRef contains the SAS address, corresponding starting 32-bit virtual address in the page pool, and ending address. So, in most cases, only one level of indirection is required to access fields in persistent objects.

Thus, the implementation uses a standard Java stack and introduces no new bytecodes. As we shall see later, it needs to do a bit more work to interpret some of the bytecodes.

The SAS client maintains SASRefs (TLB equivalent) for each Java thread. The SAS client also manages local heap storage for temporary/local Java object instances. Temporary instances are referenced by temporary SASRefs (TempRefs). All SASRefs are subject to garbage collection when the object reference goes out of scope. Temporary object heap storage is freed via mark and sweep garbage collection. Temporary object handles can't be freed until all the SASRefs and temporary heap (both local object and Copy-On-Write) COW storage) have been scanned for references.

The Java VM supports both local/temporary and persistent/shared (SAS storage) objects. The Java VM primarily uses load/store methods of the SASRef C++ class in its implementation, allowing the VM to largely ignore the details of temporary vs. persistent objects.

References to temporary objects stored in persistent objects require special handling. The Java VM detects the case of storing a temporary reference into a persistent object and the SAS client marks the referenced temporary handle "keep forever" (or until that Java VM instance terminates). Note that this is not a recommended programming practice, since the reference is transient. Temporary SAS addresses are tagged so we can detect if the reference was created in another JVM. There is no problem with storing a persistent reference in a temporary object.

SASRefs to permanent storage (PermRefs) maintain page reference counts for the shared page pool. The page reference count is incremented when the SASRef is created and decremented when the SASRef is garbage collected. Shared SAS pages cannot be removed (stolen) until the page reference count is zero. Thus page removal from the shared pool is indirectly driven by the SASRef garbage collector.

- Persistent objects are not garbage collected but must be explicitly deleted via a special method on a static class<sup>1</sup>.

## 2.1 SAS Simulator

SLS is an address space hungry technology. A 32-bit address is just not large enough to persistently map all the code and data required for interesting commercial applications. Our solution is to extend the Java VM to simulate a large address architecture (104-bits, following the AS/400 object pointer) for 32-bit platforms. This requires:

- Simulating real memory as a page pool allocated in shared virtual memory.
- Simulating virtual (104-bit) to real (32-bit) address translation as an inverted page translation table.
- Simulating the address translation look-aside buffer (TLB) mechanism as a local hash table.

Obviously on a 64-bit architecture we would not need to simulate address translation. For 64-bit systems the transform from 104-bit to 64-bit is trivial and the 64-bit address can be used directly to access storage.

The 104-bit virtual addresses are stored in reference fields as 128-bit quadwords (which we call SAS addresses). The SAS address space is partitioned into persistent shared global and temporary private local spaces. The vast majority of the SAS address space is reserved for persistent shared storage. SAS addresses are used for all Java references, whether persistent or temporary. The internal structures of loaded (persistently activated) classes and persistent storage heaps use SAS addresses throughout.

References to temporary local objects are also stored as quadword SAS addresses. A portion of the SAS address space is reserved for temporary local objects and supports a trivial translation from SAS to 32-bit addresses (the high order bits have a special value and the low order bits are the virtual address). Temporary SAS addresses always point into the local temporary heap of a specific Java VM instance. This quadword object reference format supports interoperation with future 64-bit JavaSAS implementations. This also ensures that persistent Java object references are context independent.

The SAS simulator function is split into a client instance for each JavaSAS VM and a server instance to manage shared resources for each system. Client and server instances run on the same machine. On a particular system JavaSAS VMs (SAS clients) share a single page pool managed by a single server. This allows efficient sharing of objects between

---

1. Data is the life blood of commercial enterprises and they require strict controls over access and deletion. Thus, it is unacceptable that data could be deleted by accident (some other object is deleted) or when some other object is updated (pointer set to null). For this reason, JavaSAS does not implement persistence by reachability but requires explicit deletes which can be audited. In some enterprises such as insurance companies or hospitals the law requires that data never be deleted, just marked inactive.

The results for BOB-C running on JavaSAS are very encouraging. We have successful BOB-C runs for AIX (RS6000 model 250, 60MHz 601 PowerPC, 128MB RAM), Windows95 (Gateway2000, 200MHz MMX Pentium, 64MB RAM), and Windows/NT (IBM model 750, 166MHz Pentium, 80MB RAM). However, for the following reasons, we can not claim fully qualified TPC-C results:

- We have not yet implemented full roll-back and recovery
- We are running on desktop workstations that do not support the terminal simulators specified in the TPC-C specification
- We are running on desktop workstations that do not have sufficiently large real memory and disk capacity for full scale (item, stock, and customer) object populations for multi-warehouse configurations
- Our results have not been audited by the Transaction Processing Performance Council auditors.

We have tested with multiple terminals simulated by running multiple threads displaying to a single shared test window, as well as with multiple processes each with its own text display window. As text display in windowed systems cause significant CPU loads, we implemented a non-display option to serve as stand-in for a client/server configuration. We have built a fully populated (~2.35 million Java object) BOB-C warehouse and run upto 40 processes against it. We also run multiple BOB-C Warehouse configurations with the item, stock, and customer populations scaled down to fit our available disk space (normally 700MB to 1GB). Even with these restrictions, we have achieved impressive and (to some) counterintuitive results.

## 2.0 Implementation

The JavaSAS VM integrates the simple and effective object persistence and sharing semantics of single-level store (SLS) with the Java bytecode interpreter and runtime. The initial focus is to bring SLS persistent programming model to 32-bit Java client platforms. Our goal is to make the SLS programming model available for standard 32-bit desk-top systems and the huge pool of programmers that support them. Our implementation involves extensions to the Java VM and runtime environment:

- SAS Persistence Enabling
  - A SAS storage simulator.
  - Extend the JVM to recognize SAS storage references and call the SAS simulator for translations.
  - A Java SAS class loader for activating persistent Java classes.
  - Java lifecycle extensions for creating persistent objects in SAS storage.
- Persistence Model
  - Persistence is orthogonal to type: instances of any class can be temporary or persistent depending on their method of creation. Transient objects are created, as usual, with the Java **new** operation, persistent objects are created with factory methods discussed later.

Persistent virtual storage offers an interesting alternative to object mapping. This technique allows the persistent form of the objects to (closely) match the object's runtime form. It is also efficient as:

- Data is accessed and shared “in place”, minimizing data movement overhead
- Object data and references are accessed directly with load and store instructions
- The object's virtual address is persistent, eliminating object identity to virtual address mappings and lookups

The result is a programming model where persistence is as simple to program and as fast to access as temporary objects.

Such technology has been commercially available for over 17 years in single-level storage (SLS) systems, starting with the IBM System/38 and now the AS/400 [AS/400]. SLS systems require large address architectures (starting with 48-bit and now 64-bit addressing). Recent papers [Chase] have demonstrated that the SLS storage model can be applied to commodity 64-bit processors.

Several researchers have developed persistent objects systems on large address SLS systems [Rosen1, Rosen2, Malhotra] and have demonstrated that SLS allows a simple, natural implementation of object persistence.

But the majority of programming platforms are still non-SLS 32-bit. This paper describes a prototype effort to provide the same simplicity of persistent programming that SLS provides to 32-bit non-SLS platforms. Essentially, we implemented a Java Virtual Machine (JVM) to simulate a large (> 64 bit) virtual address, then added a SLS paging subsystem (SAS) to support persistent virtual storage. This JVM executes standard JDK 1.0.2 class files and runs on several popular 32-bit platforms. Persistent storage and associated locks are sharable across all instances of this JVM running on a system. We use this Shared Address Space attribute of our JVM to name our project and prototype “JavaSAS.” Details of the implementation are discussed in section 2.

Our implementation was targeted towards standard business applications. In this arena, performance and scalability are key issues that must be addressed, measured, and proven. Currently available Java benchmarks (Caffeine marks) do not measure total system performance under transaction loads (including display and file I/O) that are required for business applications, nor do they address the scaling issue (both total number of persistent and number of concurrently active objects). We implemented a new OO Business Object Benchmark (BOB-C) based on the Transaction Processing Performance Council benchmark C [TPC-C] specification. Since BOB-C reports the same results as TPC-C, we can directly compare OO (BOB-C) and procedural (TPC-C) implementations for popular platforms. A figure of merit for an OO implementation is simply the BOB-C/TPC-C performance ratio. More importantly, it allows for direct comparison between competing object persistence implementations when available for platforms publishing TPC-C results.

# Java(TM) Persistence via Persistent Virtual Storage

**Maynard P. Johnson, Steven J. Munroe, John G. Nistler, James W. Stopyro**  
**IBM Corporation, Highway 52 & Northwest 37th St., Rochester, MN 55901**

**Ashok Malhotra**

**IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Hts., NY 10598**

## Abstract

Persistent Java(TM) objects are important for using Java in business applications. Storing Business objects in a Relational Database (RDB) has proven unsatisfactory. Direct use of a RDB prevents full use of Object Oriented (OO) technology, while schema mapping objects to RDB rows is a difficult problem. An alternative is persistent virtual storage where the persistent and runtime forms of objects are the same. This improves efficiency and allows a simple intuitive persistence program model but requires a large address space. Single Level Store (SLS) large address architecture (48-bit and 64-bit) systems have been commercially available since 1980 (IBM System/38 and AS/400) and recent papers describe similar storage semantics on commodity 64-bit processors. Our prototype implements a Java Virtual Machine (JVM) that simulates large address architecture for 32-bit systems (AIX, Windows95, WindowsNT), then adds a subsystem to create a shared address space (SAS) with persistence. The SAS subsystem provides SLS storage semantics for non-SLS systems. The implementation is targeted towards typical business applications. For performance testing we devised a Business Object Benchmark (BOB-C), based on the well known TPC-C(TM) data processing benchmark, and made some measurements. These are discussed later in the paper.

## 1.0 Introduction

Currently the Java language [Gosling] directly supports only temporary local objects. Persistent Java objects are important to applying Java to business applications. Traditionally, persistent data for business applications is stored in a RDB, but this has proven to be unsatisfactory. Direct application use of the RDB prevents the application of OO technology to core business problems (using polymorphic behavior to build extensible frameworks). The obvious approach is to support persistent objects by mapping objects to RDB rows. In this model, temporary object instances have their images freeze dried and written to one or more rows. These images can be later resurrected from the freeze dried form into another temporary object. This approach makes it difficult to support complex object relationships and forces harsh compromises between the simplicity of the program model and performance of the application. Maintaining object identity across freeze/thaw cycles is also a challenge. Similar problems apply to storing objects in files.