

Architecture of the PEVM: A High-Performance Orthogonally Persistent Java™ Virtual Machine

Brian Lewis, Bernd Mathiske, Neal Gafter

Architecture of the PEVM: A High-Performance Orthogonally Persistent Java™ Virtual Machine

Brian Lewis, Bernd Mathiske, and Neal Gafter

SMLI TR-2000-93

October 2000

Abstract:

This paper describes the design and implementation of the *PEVM*, a new scalable, high-performance implementation of orthogonal persistence for the Java™ platform (OPJ). The PEVM is based on the Sun Microsystems Laboratories Virtual Machine for Research (ResearchVM), which features an optimizing Just-In-Time compiler, exact generational garbage collection, and fast thread synchronization. It also uses a new, scalable persistent object store designed to manage more than 80GB of objects. The PEVM is approximately ten times faster than previous OPJ implementations and can run significantly larger programs. It is faster than or comparable in performance to several commercial persistence solutions for the Java platform. Despite the PEVM's speed and scalability, its implementation is simpler than our previous OPJ implementation (e.g., just 43% of the VM source patches needed by our previous OPJ implementation). Its speed and simplicity are largely due to our pointer swizzling strategy, the ResearchVM's exact memory management, and a few simple but effective mechanisms. For example, we implement some key data structures in the Java™ programming language since this automatically makes them persistent.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:

brian.lewis@eng.sun.com
bernd.mathiske@eng.sun.com
neal.gafter@eng.sun.com

© 2000 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE, HotSpot, JVM, and PJama are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>. The entire technical report collection is available online at <http://www.sun.com/research>.

Architecture of the PEVM: A High-Performance Orthogonally Persistent JavaTM Virtual Machine

Brian Lewis, Bernd Mathiske, Neal Gafter
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900

1 Introduction

The Forest project at Sun Microsystems Laboratories and the Persistence and Distribution Group at Glasgow University are developing *orthogonal persistence* for the Java platform (OPJ) [AM95]. This gives programs, with only minor source file changes, the illusion of a very large object heap containing objects that are automatically saved to stable storage, typically on disk, and fetched from stable storage into virtual memory on demand [JA00].

The PEVM is our most recent OPJ implementation. Based on our experience with our previous PJama system (“PJama Classic”) [DA97], we wanted a system with more performance, scalability, and maintainability to support our development of OPJ and a related project that is investigating controlled sharing and concurrency control through transactions [Day00]. The PEVM is based on the high performance Sun Microsystems Laboratories Virtual Machine for Research (“ResearchVM”)¹, which includes an optimizing Just-In-Time (JIT) compiler [DA99], fast thread synchronization [ADG⁺99], and exact generational garbage collection. It is also based on the Sphere recoverable persistent object store from Glasgow University [Pri00] [PAD⁺97]. Sphere is intended specifically to overcome the store-related limitations we experienced with PJama Classic. It can store up to 2^{31} objects, or about 80GB with 40 byte objects.

This paper describes the design and implementation of the PEVM. Our design goals are listed in Section 2. Section 3 outlines and describes in detail the PEVM’s architecture. This is followed by a section that discusses the performance of the PEVM, another that describes related work, and then by a summary and conclusion section.

¹The ResearchVM is embedded in Sun’s Java 2 SDK Production Release for the SolarisTM Operating Environment, available at <http://www.sun.com/solaris/java/>.

2 Design Goals

This section discusses our goals for the PEVM. Our primary goals were to provide high performance and scalability:

- Execution speed of persistent applications should be as high as possible. For example, while some other persistent systems (e.g., PJama Classic) use indirect pointers to objects to simplify their implementation of eviction, we wanted the PEVM to use direct pointers for speed.
- Object caching should be efficient and non-disruptive: the pauses it produces should be small. As an example, while some other systems such as PM3 [HC99] use virtual memory traps to retrieve objects, we wanted the PEVM to use explicit read barriers since these have lower cost.
- The PEVM must be able to operate with the same very large number of objects as Sphere.
- The ResearchVM supports heaps of more than one gigabyte. The PEVM should be able to use such large heaps without significantly increasing the disruption caused by garbage collection.
- The ResearchVM supports server applications with hundreds of threads. The PEVM should not introduce bottlenecks that significantly reduce its multi-threading capabilities.

A secondary design goal was simplicity. We wanted to create a system that was easy to maintain and enhance. We wanted to use straightforward solutions and to only add complexity when necessary. We deliberately chose simple initial solutions as long as they were easy to replace. This allowed us to build a running system quickly, which we used to gain experience and to discover where improvements were needed. For example, our first implementation of the Resident Object Table (ROT) (see section 3.3.1 below) was a single-level hashtable that required long pauses during reorganization when it grew to hold a large number of objects. We subsequently replaced it with a more sophisticated implementation.

In addition, we also began with the goal of having the PEVM support persistent threads—to be able to store the state of all threads, and then later restart each thread at the machine instruction where it left off. However, we were unable to implement persistent threads because the thread support libraries used by the ResearchVM do not allow us to read and later restore the state of threads. As a result, we can only checkpoint the state of objects.

3 Architecture of the PEVM

The ResearchVM consists of an interpreter, a JIT compiler, runtime support functions, and a garbage collected heap. The PEVM extends the ResearchVM with an object cache and a persistent object store. Figure 1 shows its main components and indicates where data transfers between them occur:

- The heap and the object cache are integrated and contain both transient and persistent objects (see section 3.3). The PEVM uses the ResearchVM's default heap configuration with a nursery generation and a mark-compact old generation.
- The scalable resident object table (ROT) (see section 3.3.1) translates references to objects in the store into virtual memory addresses in the object cache.
- The store driver (see section 3.2) hides details of interacting with a store from the rest of the PEVM.

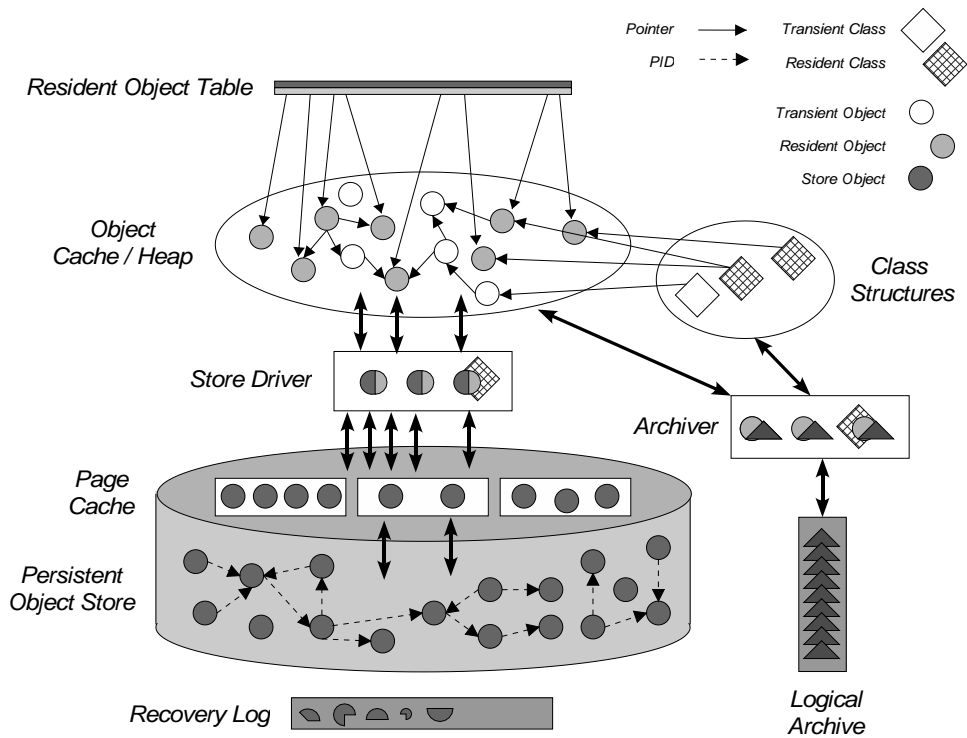


Figure 1: Architectural Components

- The default persistent object store implementation used by the PEVM is Sphere, which includes an internal page cache and a recovery log. Other store architectures can be used by writing a new driver that implements the store driver interface.
- The logical archiver exports a store’s objects to a file in a format that survives changes to the PEVM. It also supports reconstructing the “store” from an archive file. (see section 3.7).
- The *evolution* facility supports application change while preserving the existing data in a store (see section 3.8).

3.1 Representation of objects and references

An object reference in the PEVM is either a direct pointer to an object in the heap or a persistent identifier (*PID*) that uniquely identifies the object in the store. A *PID* is allocated by the store the first time an object is written to it by a *checkpoint* operation. *PIDs* are distinguished from heap pointers by having a *1* in their low-order bit (heap objects are allocated on an eight byte boundary). We use direct pointers to maximize the PEVM’s CPU performance; see section 3.3.3 for more detail.

The PEVM adds additional information to the header of each heap object. This includes an object’s *PID* (if persistent) or 0. A flag indicates whether the object has been modified. Other flags support eviction and indicate, for example, whether the object can never be evicted because it is essential for the PEVM’s operation. The modified flag actually indicates, when set, whether the object is “clean” (unmodified); this allows a single instruction to mark objects dirty.

To make checkpointing and object faulting as fast as possible, the PEVM uses representations for instance and array objects in the store that closely resemble those in memory, except that the object references they contain are replaced by the corresponding PIDs (*unswizzled*). It represents classes on disk using an architecture-neutral description that includes a PID for an array containing the class file's bytes, a PID for its class loader, the values of static fields, and a set of flag bits that indicate whether the class has been verified, linked, and initialized. This representation is much simpler than one that tries to mirror the in-memory representation of classes (as in PJama Classic). The PEVM reloads classes (without reinitialization or relinking) when it faults them in, and uses the flag bits and static field values to restore the class's state. We found that reloading classes from the store is not a significant runtime cost.

3.2 Store driver

While Sphere is the default store used by the PEVM, we occasionally use a simpler store implementation to isolate problems. We are just beginning to experiment with a third store that is log-structured. A store driver insulates the PEVM from the details of a store. It implements a high-level interface of store operations used by the rest of the PEVM. It also performs all conversions between the heap and store representations of objects, and is the only system component that understands both representations.

3.3 Object cache architecture

As described above, the heap and the persistent object cache are combined. Other persistent systems (including PJama Classic) sometimes use a separate object cache so that persistent objects can be managed differently from transient ones. Despite the potential advantages of this architecture, we chose a combined heap because it required fewer changes to the ResearchVM and because it uses storage more effectively: space not needed for transient objects can be used for persistent objects, and vice-versa. The remainder of this section describes other components of the object cache.

3.3.1 ROT (Resident Object Table)

The ROT is a lookup table holding pointers to every persistent object in the object cache. It can be queried with a PID to determine whether the corresponding persistent object is already cache-resident and, if so, its address (see the description of object faulting in section 3.3.6). The ROT also serves as a GC root, which prevents (reachable) persistent objects from being discarded by a garbage collection. The PEVM uses a separate ROT for each heap generation to speed up young generation garbage collections.

Since the size of the ResearchVM's heap can be more than one gigabyte, the ROT must be capable of holding references to hundreds of millions of objects. To scale well, the ROT is organized as two tiers. A large fixed-size "high" table points to a number of small, fixed-size "low" tables. The high-order bits of the PID are used to index the high table to get the appropriate low table, then the low-order PID bits are used to search that low table. The small low tables allow fast reorganization and fine-grained locking. The PEVM must lock a low table when entering a new persistent object to prevent another thread from also entering an object and possibly reorganizing the table.

Table 1: Language operations that are sources of object references

Operation	Example
instance field access	<i>instance.field</i>
static field access	<i>Clazz.field</i>
array element access	<i>array[index]</i>

3.3.2 The PCD and the String intern table

The PEVM uses two other key data structures, the Persistent Class Dictionary (PCD) and the String intern table. These are metadata that are maintained in the store. Both data structures are implemented in the Java programming language because their performance is not critical and persistence is automatic. PJama Classic implemented these in *C* and we found the code that translated between the memory and store representations error prone. However, implementing them in the Java programming language does complicate the PEVM’s bootstrap initialization somewhat because they cannot be used until enough of the VM is running to support programs.

The String intern table manages String literals, which the Java programming language requires be unique (and shared) within a program’s execution. The PEVM uses the String intern table to ensure that a literal String is unique across the entire lifetime (that is, repeated executions) of a persistent program.

The store must contain the class for each object in order to ensure behavioral consistency for persistent programs. It uses the PCD to hold that class information. The PCD is searched whenever a class is loaded: if the class is in the store, it is loaded using the PCD’s information. Like the ROT, the PCD is organized as a two-level hashtable. The PEVM speeds up class lookup with the following optimization. It interns the class name before doing a lookup: if the name is not persistent (e.g., it is new), the class cannot already be persistent.

3.3.3 Swizzling strategy

Pointer swizzling speeds up programs by translating PIDs into normal virtual memory addresses in the object cache. The PEVM swizzles a reference when it is first pushed onto a thread stack: if it is a PID, it faults the object in if necessary (it may already be in the object cache), then replaces the PID on the stack and in memory with the object’s address. References are pushed when a value is accessed in an object: an instance (including a class) or an array. The primitive language operations that access object references from other objects are listed in Table 1. Thus, references on the stack are always direct pointers; this includes all references in local variables, temporaries, and method invocation parameters.

Using the terminology of [Whi94], our swizzling strategy is lazy and direct. It is lazy at the granularity of a single reference: references in resident objects stay unswizzled until they are accessed.² This has the advantage that only the references used are swizzled. Our strategy is direct because we replace a PID by the in-memory address of the referenced object. Other persistent systems often use indirect pointers: a swizzled reference points to an intermediate data object (*fault block*) that itself points to the object when it is in

²Another term for this lazy swizzling strategy is “swizzling on discovery.”

memory. Indirect swizzling provides more flexibility in deciding what objects to evict from the cache and when to evict them [DA97]. We chose a direct strategy because it requires fewer source changes: only the implementations of the *getfield/putfield*, *getstatic/putstatic*, and array access bytecodes need to be changed. It also has less CPU overhead since it avoids an extra level of indirection. More detail about our swizzling strategy appears in [LM99].

3.3.4 Persistence read and write barriers

Barriers [HM93] are actions performed during certain object access operations (e.g., reads, writes) to support object caching and checkpointing. The PEVM's persistence read barrier swizzles object references and, when necessary, faults in persistent objects. Because of its swizzling strategy, the PEVM only needs to add read barriers where the three operations in Table 1 are performed and where the result is known to be a reference. As a result, stack operations (the majority of operations) do not require read barriers.

The PEVM's persistence write barrier marks updated objects so that on a later checkpoint (if they are reachable from a persistent root), they will be propagated to the store. We use a relatively expensive but simple mechanism to mark objects dirty: we reset the "isClean" flag in its header. At checkpoint time, we must scan all resident objects in order to discover which ones have been updated. This scheme is adequate today, since writes occur much less frequently than read. However, later, when the number of resident objects becomes large (several million), a more sophisticated technique will be needed to keep checkpointing non-disruptive.

The JIT can implement the persistence write barrier using a single instruction since it already holds the reference in a machine register. It implements the read barrier by seven inline instructions. Three instructions are executed in the common case (already swizzled), while all seven and a procedure call are executed if a PID is encountered:

1. Check if the reference is a PID or an address. If an address (fast path), skip the rest of the barrier.
2. (Slow path) Pass the PID to a procedure that tests whether the PID's object is resident in memory and, if not, faults it in. The procedure returns the object's heap address. The PID in memory is replaced by this address to avoid re-executing the barrier's slow path if a future access is made with the same reference.

Except for JIT-generated code and a few support routines in assembly language, the ResearchVM uniformly uses its *memsys* software interface to access runtime values such as objects, classes, arrays, and primitive values [WG98]. This meant we could easily get nearly total coverage of all access operations in non-JITed code by adding our barriers to the implementation of *memsys*.

3.3.5 Checkpointing

The checkpoint method first invokes any *checkpoint listeners* and then *stabilizes* the state of the program. Special actions that must occur when a checkpoint happens are supported by the event listener interface, *OPCheckpointListener*. A checkpoint listener is an object whose class implements this interface and which is registered using the method *OPRuntime.addListener()*. The *checkpoint* method of each checkpoint listener will be called at the start of each checkpoint.

Stabilization writes all changed and newly persistent objects to the store. The PEVM does not currently stabilize threads: it cannot write the state of each thread and cannot restore the state of each thread after a

restart. It starts a stabilization by stopping all other threads to ensure that it writes a consistent set of objects. As a result, no heap storage can be allocated and so no code written in the Java programming language can run. It then makes two passes: a gather pass to identify and allocate PIDs for the objects to stabilize, and a save pass to write objects to the store. It stabilizes classes before their instances. Objects are written using a breadth-first traversal. Stabilization uses the following algorithm:

1. If there is no store, then first create a store file. Write a store header object with a well known PID, then write each bootstrap class. The bootstrap classes are those needed to support the operation of the PEVM itself: for example, the classes *Object*, *Class*, and *Thread*.
2. *Gather pass*: Create a list all objects and classes to be stabilized and reserve PIDs for them. First, gather all primary promotion objects: iterate over the ROT and add each modified object to the list. This examines all objects in the ROT but the performance to-date is acceptable.
Next, gather other promotion objects: recursively add objects reachable from the primary objects that have not yet been stabilized. The PCD must be updated to reflect each class being stabilized. An array is created in memory to record the PCD updates. Storage for it is allocated by C code since no code written in the Java programming language can execute. The array has an entry for each class being stabilized that contains the class's name, PID, and the PID of its class loader. Space for this PCD change array is reserved in the store at this time; the array will be written when stabilization completes.
3. If the store was just created, write the store header object. This is the first point at which anything is written to the store.
4. *Save pass*: Iterate over all objects to be stabilized, write them to the store, and register them in the ROT. Objects are written in the same order that their PIDs were reserved. For each class being written, add an entry to the PCD change array. At the end of the save pass, write the change array to the store.
5. After the stabilization, or when the program restarts, update the PCD. This is done by code written in the Java programming language that reads and processes the PCD change array written by the last stabilization.

Checkpointing does not need to do a garbage collection, which is crucial to keep latency down. Objects are written using a breadth-first traversal. Objects are unswizzled in the store's buffer pool so that they are left intact in the heap and are immediately usable after a checkpoint, which also minimizes latency.

In the future, when we stabilize larger programs, we may find that scanning all objects in the ROT is too expensive. It may be necessary to use remembered sets or another mechanism to record modified objects.

3.3.6 Object and class faulting

When the PEVM's read barrier faults in an object, it ensures that the object's class is in memory (faulting it in if necessary) and swizzles the object's reference to it. This allows the PEVM to use the ResearchVM's existing procedures without change. It allocates space for the object in the heap and then enters the object into the ROT.

Class faulting uses information recorded in the PCD. The class is loaded using the PID found there for its class file. To use the ResearchVM's standard class loading code, the PEVM first ensures that the class loader for the class is resident and swizzles its reference.

3.3.7 Eviction

The PEVM frees up space in the object cache by *evicting* some persistent objects. We do eviction during garbage collection since that allows us to use the collector's mechanisms to update all references to each object. More specifically, we do eviction during full garbage collections (collections that collect the heap's old generation as well as its young generation) because at that time the entire heap is full, which makes it essential to free space. Because of the complexity required, we do not currently evict dirty objects or objects directly referenced from a thread stack. We also do not evict objects necessary for the PEVM's own operation.

We support multiple eviction strategies. These include the following:

1. Total eviction. This discards all possible persistent objects. Only those objects necessary for the PEVM's operation are kept.
2. Random eviction. This randomly selects objects to evict until it has evicted a percentage of objects that was specified on the PEVM's command line.
3. Second chance eviction. In this strategy, an object marked for eviction is kept if it is referenced before the next old space collection. Second chance eviction keeps the most recently used objects in the hope that they will be used again.

The default eviction strategy we use is second chance since it proved to be the most effective strategy among those we tried. It allows the highest application throughput with reasonably large heaps, which is the normal case. The PEVM is intended to support large applications and to be run on machines large enough to allow the use of large heaps. If the PEVM is run with a heap that is nearly full of transient objects (i.e., a heap that is too small), then total eviction is best since in that case, retaining persistent objects will tend to provoke frequent garbage collections and the cost of these will outweigh any advantage of retaining the objects.

Figure 2 shows the impact of different eviction strategies on the performance of the pBOB benchmark [DAK00] with a 10% populated company when eviction is necessary: when the heap is too small to hold all needed persistent objects. IBM created the pBOB benchmark to measure the performance of programs written for the Java platform that operate on typical business objects stored in object databases. In this figure, the "no evict" column is empty because pBOB cannot run in 20MB unless some persistent objects are evicted. This figure shows that second chance eviction performs better than either total or random eviction.

Figure 3 shows the impact on pBOB of the same eviction strategies with a heap large enough to hold all needed persistent objects. As this figure shows, second chance eviction performs nearly as well as no eviction since it retains necessary objects.

Second chance eviction operates as follows: First, the collector marks the eviction candidates as "victims" and unswizzles references to them. Then, later if the program dereferences a reference to a victim object, that object's victim mark is cleared. On the next old generation collection, all remaining victim objects

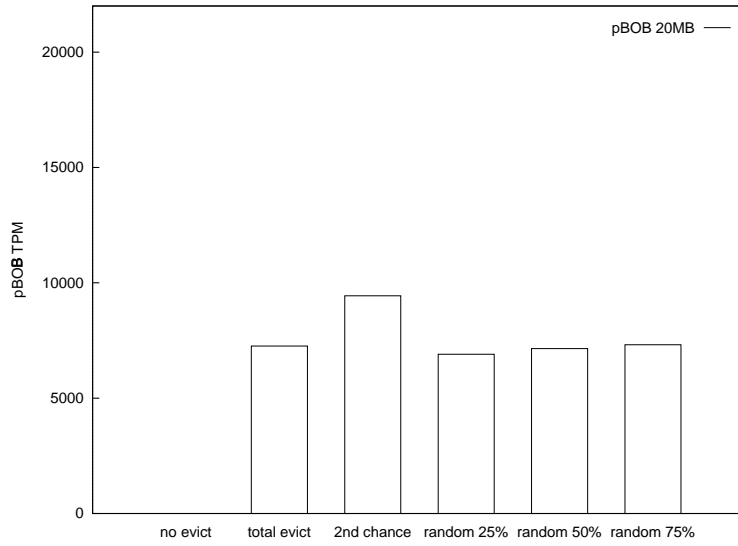


Figure 2: Different eviction strategies: pBOB benchmark, 20MB heap

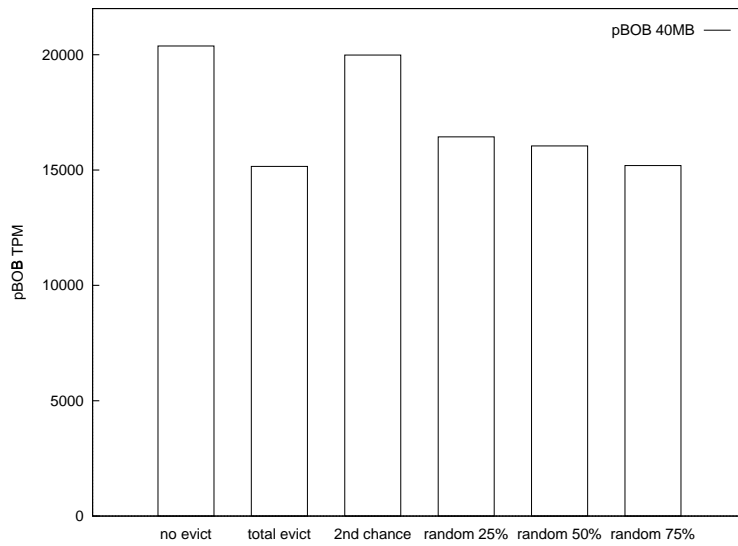


Figure 3: Different eviction strategies: pBOB benchmark, 40MB heap

are removed from the heap. Our second chance eviction strategy is relatively aggressive since it evicts *all* unreferenced objects, including some that might be referenced again in the future. It typically evicts more objects than the second chance strategy of PJama Classic, which evicts no more than a portion of the object cache. As a result, we are investigating modifications of this scheme that will retain more objects.

3.4 Support for external state

There is a limit to orthogonal persistence for the Java platform (OPJ): external state not under control of the OPJ system. For example, open sockets, files, windows, and databases do not have state held completely in objects, and programmer assistance is required to handle them correctly.

The PEVM supports four techniques for handling external state:

- *Transient* fields and program tests: Transient static fields or instance fields marked using *OPTransient.mark()* are reset to the appropriate default value (*NULL* or 0 or *false*) when the program restarts. Such transient fields can be combined with explicit program checks to decide when to recreate or reestablish the external state. For example, a program using a *java.io.File* can mark as transient the field holding that reference. Then the program can test whether the reference is *NULL* and, if so, re-open the file. Note that we cannot use the *transient* modifier for instance fields because it has been given an unfortunate interpretation by JavaTM Object Serialization: transient fields are not written using a call to *defaultWriteObject()* However, such fields can still be written by a *writeObject* method, so such fields may or may not be “transient”. In particular, they are not transient in the sense of orthogonal persistence.
- Callbacks on program restart: Special initialization at program restart is supported by the event listener interface, *OPResumeListener*. An object whose class implements this interface has a *resume* method that will be called on a restart if the object is registered using the method *OPRuntime.addListener()*. For example, the *java.lang.ref.Reference* class uses one of these callbacks to recreate the object it uses as a lock on restart.
- Lazy callbacks on first object access: Objects that require special initialization before first use are supported by the *OPResumable* interface. When an object is faulted in, if its class inherits from the *OPResumable* interface, its method *resume* is called. The method is called “lazily” or just-in-time, at the first use of the object. This technique is used, for example, by *java.awt.Font* to reinitialize its instances as needed. Its callback method invokes native code that reinitializes the font information for an instance.
- Lazy callbacks on first class access: For syntactic reasons, there is a separate mechanism used for classes that behaves like the one above.

If we had been able to support persistent threads, additional mechanisms would have been necessary to manage external state. Because the checkpoint and restart operations modify native state, they would need to be synchronized with any code that uses native state.

3.5 Initialization of the PEVM

When a program restarts, classes needed for PEVM's own operation must be loaded from the store. This includes a large number of classes such as *Class*, *ClassLoader*, and *Exception*, and the classes that implement the PCD and the String intern table. These essential classes are called the *bootstrap classes*. They are discovered on the first run of a program and recorded in a bootstrap list. When a store is created by the first checkpoint, the bootstrap list is written out and its PID recorded in a known place in the store. On a restart, the PEVM loads the bootstrap classes in the list from the store during the early stages of its initialization. There are currently 179 bootstrap classes.

The PEVM must also reinitialize class libraries on a restart. Most of these libraries use native code to implement their functionality. The PEVM reloads each previously loaded native library when restarting. It also reinitializes the JNI (Java Native Interface) information used by native code to access fields and invoke methods: JNI class, field, and method handles. Much of this reinitialization can be done automatically by taking advantage of a common programming idiom used in the implementation of class libraries. Many library classes use a method *initIDs* to initialize their JNI handles. To reinitialize them, when the PEVM loads a class, if it has an *initIDs* method of the correct signature, it invokes that method.

3.6 Store garbage collection

Like the heap, the store can accumulate objects that are no longer reachable. Sphere includes a compacting store ("disk") garbage collector to reclaim space [Pri00]. This collector deals with the large size of stores. It also deals with the slow access time of disks by minimizing the number of disk accesses it does. The store garbage collector operates concurrently with programs and allows concurrent access to the objects it is collecting. It is also fault-tolerant: if a collection crashes, either all updates will be reflected in the store or none will.

3.7 Archiving

A store depends on the version of the PEVM that created it. It may no longer be usable if a change is made to the store format or to a class in the store, even if that class is purely internal to the PEVM's implementation.³ To avoid this problem, we created a portable archive file format that represents the user-visible state of the program.

Creating an archive resembles checkpointing except that objects are written in a portable format. Each object is assigned a unique index and references to it are replaced by that index. The archive format for each object includes its identity hash value. The format for a class instance has its class and the values of all fields, and the format of a class includes its name, class loader, and a list of the names and types of its non-static fields. We record the values of static variables and the class file for *user-defined classes*: those loaded into the application class loader or a user-defined class loader. By only recording such data for these classes, the archive is isolated from most changes to classes implementing the PEVM.

Restoring from an archive recreates the application's state. Classes are loaded and set to the state in the original program (linked, initialized, etc.) without running static initializers. Instances are created without

³However, the PEVM's stores depend much less on the specific version of the PEVM than those of PJama Classic. The store format for PJama Classic was much more intricate and changed more frequently.

Table 2: Transient OO7 operations: warm runs

	Small			Medium			Large		
	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>
Q1	1.00	1	1.00	1.00	1	1.00		1	1.00
Q4	2.00	1	1.00	2.00	1	1.00		1	1.00
T1	15.87	372	0.80	12.49	3389	0.86		3332	0.87
T2a	15.84	373	0.78	13.14	3288	0.88		3371	0.86
T2b	15.78	378	0.78	13.00	3289	0.89		3322	0.88
T2c	16.56	388	0.81	13.78	3441	0.89		3499	0.86
T6	22.41	73	0.40	19.56	75	0.40		79	0.44
mean	4656	273	187	22237	1567	1180		1595	1211

running a constructor, then their identity hash value and field values are set from the archive.

The logical archive and restore utilities are written almost entirely in the Java programming language. To allow this, the PEVM includes reflection-like native methods to read and write private fields, create instances without running constructors, mark a class as already initialized without running a static initializer, and assign identity hash values to objects.

3.8 Evolution

Traditionally, applications are separated from their data: a program loads its data from a file or database, does its work, then writes its results back. Changes to a program, including changes to its internal data structures, do not affect the on-disk data unless the on-disk representation is changed.

Orthogonal persistence saves application developers the difficulty and cost of explicitly managing the persistence of their data. Unfortunately, it breaks the traditional program development model: changing a class can make its instances in a store invalid. To deal with this, we implemented a *program evolution* facility [ADHP00] that allows programmers to simultaneously change programs and their data. The evolution facility determines how each class is being changed and, for many changes, automatically converts its instances. Complex changes that modify the representation of data structures (e.g., changing a point’s representation from Cartesian to polar coordinates) may require help from the programmer.

4 The PEVM’s Performance

Table 2 shows average “warm” execution time results for several OO7 traversals [CDN93] when PJama Classic, the PEVM, and the ResearchVM (labeled “ResVM”) are run transiently. OO7 is a standard benchmark that simulates the behavior of a CAD system traversing graphs of engineering objects. Times for the PEVM are given in milliseconds while the other times are expressed as ratios with respect to the PEVM’s time for the same OO7 configuration. We repeated each operation five times during one execution of the virtual machine. After each operation a transaction commit (in our case a *checkpoint()* call) was performed, and its time was included in the operation’s total time. For these “warm” times, we computed the average

Table 3: Transient OO7 operations: cold runs

	Small			Medium			Large		
	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>
Q1	0.50	4	0.75	0.40	5	0.80		17	0.18
Q4	0.60	5	1.00	0.33	6	0.67		6	0.83
T1	10.85	557	0.58	12.71	3324	0.90		3315	0.89
T2a	15.76	375	0.79	12.63	3355	0.88		3314	0.88
T2b	15.98	374	0.79	12.96	3288	0.90		3348	0.86
T2c	16.51	390	0.77	14.73	3448	0.88		3420	0.88
T6	21.50	75	0.48	18.66	79	0.57		83	0.59
mean	4668	296	198	22462	1585	1287		1599	1296

time for the final four operations (ignoring the initial “cold” operation). The operations were run on a Sun Enterprise 3500 with four 400MHz UltraSPARC-IITM CPUs and 4GB of memory. The OO7 benchmark is single-threaded so the additional CPUs had little effect. Each run was made with a heap large enough to hold the entire OO7 database. The geometric means of the run times in milliseconds of the traversal operations T1-T6 (only) are given at the bottom of the table for each combination of virtual machine and OO7 configuration; the query operations run too fast to give meaningful means. PJama Classic cannot run the large OO7 database, so that column is blank.

These results demonstrate how much faster the PEVM and ResearchVM are compared to PJama Classic. For the medium database, the PEVM and ResearchVM are faster by factors of 13.1 and 14.9, respectively. These results also show the low cost of the PEVM’s persistence barriers: with the medium database, the PEVM’s total runtimes for all operations is just 14% greater than the ResearchVM. One surprise in these results is how much faster the ResearchVM is on traversal T6 than the PEVM. This is due to the PEVM’s more expensive garbage collections: these do additional work necessary for persistent programs even though these runs are transient.

Table 3 shows the average “cold” execution time results for the same operations. The results here are slower than those above but are otherwise similar.

Tables 4 and 5 show the average warm and cold execution time results for the same operations when PJama Classic and the PEVM are run persistently. The column for the ResearchVM is included but left blank to make it easier to compare the two tables. Note that the warm runs of traversals T2a and T2b, which update every atomic part in the OO7 database, are much slower than either T1 (which just visits each atomic part) or T2a (which only updates one atomic part per composite part) since they cause more objects to be written during a checkpoint. We could not run PJama Classic on two update-intensive OO7 traversals (it crashed due to object cache problems) so we did not compute geometric means for those cases.

Experience with several programs shows that the average overhead of the PEVM’s persistence barriers is about 15%. The barrier cost in the interpreted PJama Classic is also about 15% [Jor96] and we expected, when designing the PEVM, that the relative cost for the PEVM’s barriers would be much higher because of the ResearchVM’s greater speed. The execution time overhead for the *pBOB* benchmark is just 9% compared

Table 4: Persistent OO7 operations: warm runs

	Small			Medium			Large		
	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>
Q1	0.83	18		1.17	18			16	
Q4	1.53	19		1.94	17			25	
T1	13.62	411		11.71	3600			3581	
T2a	13.20	428		11.56	3652			3625	
T2b	10.16	606			6523			6508	
T2c	10.85	618			6633			6615	
T6	14.13	109		14.68	105			106	
mean	4578	373			2266			2262	

Table 5: Persistent OO7 operations: cold runs

	Small			Medium			Large		
	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>	<i>Classic</i>	<i>PEVM</i>	<i>ResVM</i>
Q1	0.84	62		2.31	89			249	
Q4	1.27	49		1.34	50			72	
T1	1.16	6473		1.59	66831			66229	
T2a	1.16	6486		1.58	66962			65902	
T2b	1.19	6692			69405			68668	
T2c	1.30	6735			69652			68867	
T6	3.27	616		3.19	658			805	
mean	6024	4104			26957			27804	

to the standard ResearchVM. The SPECjvm98 [SPE98] *db* and *javac* benchmarks are slowed down by 14% and 18%, respectively. Programs that do extensive array computation such as the SPECjvm98 *compress* benchmark are slowed down up to 40%. This is because we needed to disable a ResearchVM optimization that reduces the cost of array bounds checks since it interacts badly with our read barrier.

The overall performance of the PEVM is good when running large persistent programs. As an example, pBOB run with the ResearchVM yields about 27000 pBOB “transactions” per minute when run with a single pBOB thread over a 100% populated “warehouse”. When the PEVM runs that same configuration of pBOB persistently, the result is 23403. This is despite the fact that the PEVM must fault in more than 220MB of objects from the store for this particular benchmark.

The PEVM can scale to *very* large numbers of objects. It easily supports the large OO7 database. In addition, we ran an experiment where we added additional warehouses to pBOB (each containing about 212MB of objects) until it ran over 24 warehouses: a total of more than 5GB of objects. This is significantly more than will fit into our 32 bit virtual memory and made heavy use of the PEVM’s cache management. Furthermore, we were able to run five threads per warehouse, each simulating a different client, for a total of 120 threads accessing those objects.

5 Related Work

This section discusses other orthogonal persistence solutions for the Java platform. These systems differ in the extent of their orthogonality and their scalability.

Serialization: JavaTM Object Serialization [JOS98] encodes object graphs into byte streams, and it supports the corresponding reconstruction of those object graphs. It is the default persistence mechanism for the Java Platform. Serialization is easy to use and, for many classes, automates the serialization/deserialization of their instances. However, it only serializes classes that implement the *java.io.Serializable* interface. Since many core classes do not implement this, Serialization does not support orthogonality and persistence by reachability. It also suffers from severe performance problems. Furthermore, an entire stream must be deserialized before any objects can be used.

ObjectStore PSE Pro for Java: Excelon Corporation’s ObjectStore PSE Pro for Java [LLB⁺97] is a single-user database implemented as a library. It provides automatic support for persistence but it requires that all class files be postprocessed to add annotations: additional calls to library methods to, e.g., fetch objects. This postprocessing complicates the management of class files since it results in two versions of each class file (one for transient and one for persistent use). It also makes it difficult to use dynamic class loading. The additional method calls are expensive and slow program execution significantly.

PJama Classic: Our previous implementation of orthogonal persistence for the Java platform, PJama Classic, is based on the “Classic” VM used in the JavaTM2 SDK, Standard Edition, v 1.2, which is significantly slower than the ResearchVM, primarily because of slower garbage collection and thread synchronization. The Classic VM uses a conservative garbage collector and objects are accessed indirectly through handles, which adds a cost to every object access. It also has a JIT, but PJama Classic does not support it. As a result, PJama Classic is approximately 10 times slower than the PEVM. PJama Classic is also more complex than the PEVM (requires more software patches) because the

Classic VM has no memory access interface that corresponds to the ResearchVM's *memsys*: PJama Classic needs 2.3 times more patches, 1156 versus 501. The object cache in PJama Classic is separate from the heap, which made it possible to manage persistent objects separately from transient objects at the cost of a more complex implementation. Like the PEVM, PJama Classic implements second chance eviction but since the garbage collector in the Classic VM is conservative, it lacks exact information about the location of objects, which significantly complicates eviction and checkpointing, both of which may move objects referenced from a *C* stack. Another limitation of PJama Classic is that its custom-built store is directly addressed (a PID is the offset of an object), which makes it nearly impossible to implement store garbage collection or reclustered that operate concurrently with a persistent program.

Gemstone/J: Gemstone/J [Inc98] is a commercial application server that supports E-commerce components, process automation, and J2EETM services. It includes a high-performance implementation of persistence for the Java platform that approaches orthogonality more closely than most other systems. However, it requires use of a transactional API and does not have complete support for core classes. Gemstone/J uses a modified version of the Java HotSpotTM performance engine. It implements a shared object cache that allows multiple Gemstone/J VMs to concurrently access objects in the cache. When transactions commit, changed objects are visible to other VMs. It does not store the values of static fields persistently since Gemstone felt that this would cause too many third-party libraries to fail. Interestingly, while the PJama system moved towards using a unified heap and object cache, Gemstone/J moved towards a separate cache. Their most recent version, 3.2.1, includes a separate "Pom" region that holds (most) resident persistent objects. More study is needed to understand the advantages and disadvantages of using separate (or combined) object caches.

6 Conclusions

We set out to build a new OPJ implementation with better performance, scalability, and maintainability than PJama Classic. The PEVM's overall performance is good: it is about 10 times faster than PJama Classic with an overhead (compared to an unchanged ResearchVM) of about 15%. This is largely because of our swizzling strategy, which made it simple to modify its optimizing JIT while preserving almost all of the speed of the code it generates. Also, our use of direct object pointers minimizes CPU overhead. It ties eviction to garbage collection, but we have still been able to implement a variety of different eviction schemes. In addition, the PEVM's checkpointing is much faster than that of PJama Classic. We compared the PEVM's performance to several commercial persistence solutions for the Java platform, and found it as fast or faster; however, we cannot quote specific numbers due to licensing restrictions.

We have shown that the PEVM can scale to large numbers of objects. We have tried it with a moderate number of threads (e.g., the 120 mentioned above), but do not yet know how well it scales with many hundreds of threads. The PEVM has longer pause times than the ResearchVM, and we expect to work on improving responsiveness in the future. Responsiveness will become even more important as the scale of our programs and data increase.

We made a first internal release of the PEVM in only half the time needed for PJama Classic. This was due to a combination of factors: our swizzling strategy, the ResearchVM's *memsys* memory interface, and our decision to build a relatively simple system. The PEVM's relative simplicity compared to PJama Classic (501 VM source patches versus 1156) has made it easier for us to maintain.

We have demonstrated that an implementation of orthogonal persistence for the Java platform can perform well, and can preserve most of the performance of the fast JVM upon which it was built.

Acknowledgments

We want to thank the other members of the Forest team for their numerous ideas and contributions. Grzegorz Czajkowski, Laurent Daynès, Mick Jordan, Cristina Cifuentes, Jeanie Treichel, and the anonymous referees provided helpful comments on earlier drafts of this paper.

References

- [ADG⁺99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of OOPSLA '99, Denver, Colorado, USA*, November 1999.
- [ADHP00] M. Atkinson, M. Dmitriev, C. Hamilton, and T. Printezis. Scalable and recoverable implementation of object evolution for the pjama platform. In *Proceedings of the Ninth International Workshop on Persistent Object Systems: Design, Implementation and Use, Lillehammer, Norway*, September 2000.
- [AM95] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3), 1995.
- [CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. *ACM SIGMOD Record*, 22(2):12–21, June 1993.
- [DA97] L. Daynès and M. Atkinson. Main-memory management to support orthogonal persistence for java. In *Proceedings of the 2nd International Workshop on Persistence and Java, Sun Labs Technical Report TR-97-63, Sun Microsystems Laboratories*, August 1997.
- [DA99] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 1628, Springer-Verlag*, June 1999.
- [DAK00] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable jvms. *IBM Systems Journal*, 39(1), 2000.
- [Day00] L. Daynès. Implementation of automated fine-granularity locking in a persistent programming language. *Special Issue on Persistent Object Systems, Software–Practice and Experience*, 30(4), April 2000.
- [HC99] A. Hosking and J. Chen. Pm3: An orthogonally persistent systems programming language – design, implementation, performance. In *Proceedings of the International Conference on Very Large Data Bases, Edinburgh, Scotland*, September 1999.
- [HM93] A. Hosking and J. Elliot Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the 14th Symposium on Operating Systems Principles, Asheville, NC, USA*, December 1993.

- [Inc98] Gemstone Systems Inc. Gemstone/j programming guide, version 1.1, March 1998.
- [JA00] M. Jordan and M. Atkinson. Orthogonal persistence for the java platform—specification and rationale. Sun Labs Technical Report TR-2000-94, Sun Microsystems Laboratories, July 2000.
- [Jor96] M. Jordan. Early experiences with persistent java. In *Proceedings of the First International Workshop on Persistence and Java, Glasgow, Scotland*, September 1996.
- [JOS98] Java object serialization specification, revision 1.43.
<http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html>, November 1998.
- [LLB⁺97] G. Landis, C. Lamb, T. Blackman, S. Haradhvala, M. Noyes, and D. Weinreb. Objectstore pse: a persistent storage engine for java. In *Proceedings of the 2nd International Workshop on Persistence and Java, Sun Labs Technical Report TR-97-63, Sun Microsystems Laboratories*, August 1997.
- [LM99] B. Lewis and B. Mathiske. Efficient barriers for persistent object caching in a high-performance java virtual machine. Sun Labs Technical Report TR-99-81, Sun Microsystems Laboratories, 1999.
- [PAD⁺97] T. Printezis, M. Atkinson, L. Daynès, S. Spence, and P. Bailey. The design of a scalable, flexible, and extensible persistent object store for pjama. In *Proceedings of the 2nd International Workshop on Persistence and Java, Sun Labs Technical Report TR-97-63, Sun Microsystems Laboratories*, August 1997.
- [Pri00] T. Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, University of Glasgow, May 2000.
- [SPE98] The specjvm98 benchmarks.
<http://www.spec.org/osg/jvm98>, August 1998.
- [WG98] D. White and A. Garthwaite. The gc interface in the evm. Sun Labs Technical Report TR-98-67, Sun Microsystems Laboratories, 1998.
- [Whi94] S. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, Department of Computing Science, University of Wisconsin–Madison, Madison, Wisconsin, 1994.

About the Authors

Brian Lewis is a Senior Staff Engineer at Sun Microsystems Laboratories. He is a member of the Forest project, which is developing orthogonal persistence for the Java platform. His interests include programming language implementation, runtime system design, and distributed programming. In previous projects at Sun, Brian implemented the on-the-fly bytecode compiler in Tcl/Tk 8.0 that significantly improves the performance of the Tcl scripting language. He was responsible for the machine-independent, just-in-time compilation architecture used by the Clarity C++ programming system. He also developed monitoring and debugging tools as part of the Spring distributed operating system project. Prior to joining Sun, Brian worked at Olivetti Research and Acorn Research, where he implemented a user interface toolkit and portions of the runtime system for a new operating system. At Xerox, he led the team that designed and developed the Shared Books distributed, multi-user publication management system. He also implemented portions of the Mesa programming system and developed software version management tools. He received the Ph.D. in Computer Science from the University of Washington.

Bernd J.W. Mathiske is a Staff Engineer at Sun Microsystems Laboratories. He is interested in all aspects of the implementation of persistent and distributed programming systems, and is currently focusing on persistent object caching. Prior to Sun, he worked at Sony European Research and Development on wireless mobile computing middleware. He received a Ph.D. in Computing Science from the University of Hamburg where he was one of the key designers and implementors of the Tycoon persistent language system. He also worked as a consultant and software developer in several industries.

Neal M. Gafter is a Staff Engineer at Sun Microsystems Laboratories working in the Forest project. He helped to implement the PEVM and implemented an archive mechanism for the PEVM that creates an architecture-neutral external checkpoint of a running program. He also wrote applications that take advantage of persistence including a Java compiler that caches parse trees and an NNTP (netnews) server. Prior to the Forest project, Neal was the technical lead for Sun's C++ development group and has been a member of the ANSI/ISO C++ Standards Committee since 1992. Neal has been working on compilers since 1981 and received a Ph.D. in Computer Science from the University of Rochester.