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

Brian Lewis, Bernd Mathiske, Neal Gafter
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
{brian.lewis,bernd.mathiske,neal.gafter}@sun.com

Abstract

This paper outlines 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, which features an optimizing Just-In-Time compiler, exact generational garbage collection, and fast thread synchronization. The PEVM also uses a new, scalable persistent object store designed to manage 80GB of objects. It is approximately ten times faster than previous OPJ implementations and can run significantly larger programs. Despite its greater speed and scalability, the PEVM's implementation is much simpler (e.g., just 43% of the VM source patches needed by our previous OPJ implementation). This is largely due to the pointer swizzling strategy we chose, the ResearchVM's exact memory management, and simple but effective mechanisms. For example, we implement some key data structures in the Java programming language since this automatically makes them persistent.

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) [Atkinson, Morrison 95]. 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 [Jordan, Atkinson 99].

The PEVM is our most recent OPJ implementation. We learned a great deal from our previous system ("PJama Classic"), but we wanted an OPJ system with more performance, scalability, and maintainability. 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 [Detlefs, Agesen 99], fast thread synchronization [Agesen et al. 99], and exact generational garbage collection. It is also based on the Sphere recoverable persistent object store from Glasgow University [Printezis 00] [Printezis et al. 97]. Sphere is intended specifically to overcome the store-related limitations we experienced with PJama Classic. It is designed to scale well up to 80GB.

*SunLabs document number 2000-0189.

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

[1] The ResearchVM is embedded in Sun's Java 2 SDK Production Release for the Solaris™ Operating Environment, available at http://www.sun.com/solaris/java/.

This paper describes the design and implementation of the PEVM. Our design goals are listed in Section 2. Then Section 3 first outlines and then 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.

2 Design Goals

This section discusses our goals for the PEVM's design. Our primary goals were to provide high performance and scalability. The performance criteria were the following:

- ▷ Overall execution speed of long-running persistent applications should be as high as possible.
- ▷ Object caching should be efficient. It should also be non-disruptive: the pauses it produces should be small.
- ▷ Checkpointing latency should be dominated by the performance of the object store, not by the object cache.

Our scalability goals were the following:

- ▷ Sphere is designed to manage stores of approximately 80GB of objects. The PEVM must be able to operate with the same large number of objects.
- ▷ 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.

3 Architecture of the PEVM

This section describes the PEVM's architecture. It starts with an overview that lists key design points, then discusses specific parts of the design in more detail.

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 PEVM's 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.
- ▷ 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 3.4).
- ▷ The *evolution* facility supports application change while preserving the existing data in a store (see section 3.5).

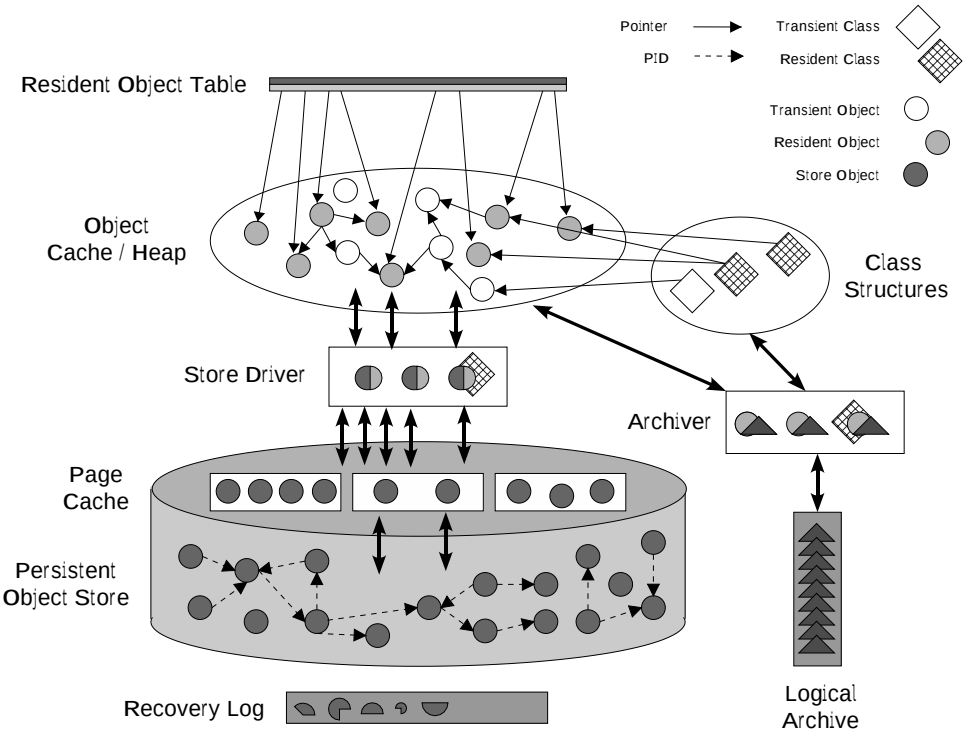

Figure 1: Architectural Components

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. 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.

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, 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 reinitializion 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 second, simpler store implementation to isolate problems[2]. A store driver insulates the PEVM from 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

[2] We are just beginning to experiment with a third store that is log-structured.

between the heap and store representations of objects, and is the only system component that understands both representations. To adapt the PEVM to an additional store architecture, it is enough to implement a new store driver.

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 other 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).

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.

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 because they can not be used until enough of the VM is running to support programs.

To ensure type safety, the store must include the class for each object and 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 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.

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 [White 94], 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. This has the advantage that only references used by the programs 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

Operation	Example
instance field access	*instance.field*
static field access	*Clazz.field*
array element access	*array[index]*

Table 1: Language operations that are sources of object references

itself points to the object when it is in memory. Indirect swizzling provides more flexibility in deciding what objects to evict from the cache and when to evict them [Daynès, Atkinson 97]. We chose this 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 [Lewis, Mathiske 99].

3.3.4 Persistence read and write barriers

Barriers [Hosking, Moss 93] are actions performed during certain object access operations (e.g., reads, writes) to support object caching and checkpointing. The PEVM's persistence read barrier is responsible for swizzling object references and, when necessary, faulting 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. Since writes occur much less frequently than reads, the PEVM's write barrier simply marks the object dirty by resetting the "isClean" flag in its header.

The JIT can implement the persistence write barrier using a single instruction since it already holds the reference in a machine register. Similarly, the JIT 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 whether the resulting reference is a PID or an address. If it is an address (fast path), skip the rest of the barrier.
2. (Slow path) Pass the PID to a runtime procedure that tests whether the PID's object is resident in memory and, if not, faults it in. In any case, 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 [White, Garthwaite 98]. 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

Checkpointing writes all changed and newly persistent objects to the store. The PEVM does not currently checkpoint native threads. It starts a checkpoint by stopping all other threads to ensure that it writes a consistent set of objects. It then makes two passes: a gather pass to identify and allocate PIDs for the objects to checkpoint, and a save pass to write objects to the store. It does not need to do a garbage collection, which is crucial to keep latency down. Objects are written using a breadth-first traversal that yields good clustering [Schkolnick 77]. 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.

3.3.6 Object and class faulting

When the PEVM's read barrier faults in a 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. This is done during garbage collection because the PEVM uses direct pointers to objects and only the collector has support to update all references to an object. Because of the complexity required, the PEVM does not currently evict dirty objects or objects directly referenced from a thread stack. Eviction is performed during a collection of the heap's mark-compact old generation. We support multiple eviction strategies, but the default is *second chance* since it proved to be the most effective strategy among those we tried. 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. 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 are removed from the heap. This strategy typically evicts more objects than the second chance strategy of PJama Classic [Daynès, Atkinson 97]. As a result, we are investigating other schemes that might retain more objects.

3.4 Archiving

A store depends on the particular version of the PEVM that created it. The store may no longer be usable if a change is made to the store format or to a class represented in the store, even if that class is purely internal to the PEVM 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 is much like checkpointing except that the objects are written in a portable format. Each object is assigned a unique index and all references to it are replaced by that index. The archive format for an object includes its identity hash value. The format for a class instance has its class and the values of all public and private fields. The representation of a class includes its name, its class loader, and a list of the names and types of its non-static fields. We also 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 user-defined classes the archived data is isolated from most changes to classes that implement the PEVM.

Restoring from a archive recreates the application's state without running any user-defined code. Classes are loaded and then set to the same state in the original program (linked, initialized, etc.) without running static initializers. Instances are created and given field values from the archive without running a constructor. Objects created from an archive are assigned the same identity hash value as the original object represented in 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 "empty" instances (without running constructors), mark a class as already initialized (without running static initializers), and assign an object's identity hash value.

3.5 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.

[3] However, 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.

Traversal	Small			Medium			Large		
	Classic	*PEVM*	*ResVM*	*Classic*	*PEVM*	*ResVM*	*Classic*	*PEVM*	*ResVM*
T1	7458	497	397	53178	4023	3522		3978	3546
T2a	7542	453	374	53385	3956	3745		3977	3513
T2b	7429	454	373	57849	3931	3508		3940	3516
T2c	7742	468	388	60169	4108	3666		4103	3656
T6	1963	100	67	1844	107	70		110	79

Table 2: Transient OO7 traversals

Traversal	Small			Medium			Large		
	Classic	*PEVM*	*ResVM*	*Classic*	*PEVM*	*ResVM*	*Classic*	*PEVM*	*ResVM*
T1	7912	513		52894	4260			4203	
T2a	7489	527		52872	4259			4257	
T2b	8061	762			8238			8279	
T2c	8739	789			8310			8582	
T6	1981	151		1867	134			131	

Table 3: Persistent OO7 traversals

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 often makes its instances in a store invalid. To deal with this, we implemented a *program evolution* facility [Dmitriev et al. 00] 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 some assistance from the programmer.

4 The PEVM's Performance

Table 2 shows the average run times in milliseconds for several OO7 traversals 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. We ran five "hot" traversals (after an initial "cold" traversal) and computed the average of the remaining run times. The traversals were run on a Sun Ultra 10 workstation with a 333MHz SPARC CPU and 1GB of memory (more than was used by any traversal). PJama Classic can not 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 14.0 and 15.6 respectively. They also show how little the additional persistence barriers used by the PEVM cost: 10.0% for the medium database.

Table 3 shows the average times for the same traversals 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. We were unable to run PJama Classic on two of the OO7 traversals.

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% [Jordan 96] and we expected, when designing the PEVM, that the relative cost for the PEVM's barriers would be higher because of the ResearchVM's greater speed. The execution time overhead for the *pBOB* benchmark [Dimpsey et al. 00] is just 9% compared to the standard ResearchVM. IBM created this benchmark to measure the performance of server programs written for the Java platform that operate on business objects stored in object databases. The SPECjvm98 [SPEC 98] *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 had 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 additional, 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.

[[We plan to include a comparison between the performance of the PEVM and PM3 [Hosking, Chen 99] in the final paper. PM3 is a state-of-the-art high-performance implementation of orthogonal persistence for the systems programming language Modula-3.]]

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: Java Object Serialization [JOS 98] 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 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 [Landis et al. 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 Java 2 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 though 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 specially 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 reclustering that operate concurrently with a persistent program.

Gemstone/J: Gemstone/J [Gemstone Systems 98] is a commercial application server that supports E-commerce components, process automation, and J2EE™ 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 HotSpot™ 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 was concerned that this would cause too many third-part libraries to fail. Interestingly, while PJama moved towards using a unified heap and object cache, Gemstone/J has moved towards a separate cache. Their most recent version, 3.2.1, includes a separate "Pom" region that holds (most) resident persistent objects. Apparently, Gemstone found that heap garbage collections shortened the lifetime of persistent objects and so reduced throughput. More study is needed to understand the advantages and tradeoffs 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. Finally, the PEVM's checkpointing is much faster than that of PJama Classic.

We have shown that the PEVM can scale to large numbers of objects. We do not yet know how well it scales with many hundreds of threads. We have tried a moderate number of threads (e.g., the 120 mentioned above), but we need to do further experiments. 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, our decision to build a relatively simple system, and the quality of the system we started with (the ResearchVM). 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, and Mick Jordan helped us by reviewing this paper and giving us their comments. We also want to thank the Sun Labs Java Technology Research Group and the Sun Solaris Software Java Topics Group for the ResearchVM and for their help during our development of the PEVM.

References

Agesen et al. 99: Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y. S., and White, D. "An Efficient Meta-Lock for Implementing Ubiquitous Synchronization". In: *Proceedings of OOPSLA'99, Denver, Colorado, USA*, November 1999.

Atkinson, Morrison 95: Atkinson, M. and Morrison, R. "Orthogonally Persistent Object Systems". *VLDB Journal*, 4(3), 1995.

Daynès, Atkinson 97: Daynès, L. and Atkinson, M. "Main-Memory Management to Support Orthogonal Persistence for Java". In: *Proceedings of the 2nd International Workshop on Persistence and Java (PJW2), Half Moon Bay, CA, USA*, August 1997.

Detlefs, Agesen 99: Detlefs, D. and Agesen, O. "Inlining of Virtual Methods". In: *Proceedings of ECOOP'99, Lisbon, Portugal*, June 1999.

Dimpsey et al. 00: Dimpsey, R., Arora, R., and Kuiper, K. "Java Server Performance: A Case Study of Building Efficient, Scalable Jvms". *IBM Systems Journal*, 39(1), 2000.

Dmitriev et al. 00: Dmitriev, M., Hamilton, C., and Atkinson, M. "Scalable and Recoverable Implementation of Object Evolution for the PJama Platform". Submitted to the Ninth International Workshop on Persistent Object Systems: Design, Implementation and Use, Lillehammer, Norway, September 2000.

Gemstone Systems 98: Gemstone Systems, Inc. "Gemstone/J Programming Guide, Version 1.1", March 1998.

Hosking, Chen 99: Hosking, A. and Chen, J. "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.

Hosking, Moss 93: Hosking, A. and Moss, J. Elliot. "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.

Jordan, Atkinson 99: Jordan, M. and Atkinson, M. "Orthogonal Persistence for the Java Platform—Draft Specification".
http://java.sun.com/aboutJava/communityprocess/jsr/jsr_020_opj.html, June 1999.

Jordan 96: Jordan, M. "Early Experiences with Persistent Java". In: *Proceedings of the First International Workshop on Persistence and Java (PJ1), Glasgow, Scotland*, September 1996.

JOS 98: "Java Object Serialization Specification, Revision 1.43".
http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html, November 1998.

Landis et al. 97: Landis, G., Lamb, C., Blackman, T., Haradhvala, S., Noyes, M., and Weinreb, D. "ObjectStore PSE: a Persistent Storage Engine for Java". In: *Proceedings of the 2nd International Workshop on Persistence and Java (PJW2), Half Moon Bay, CA, USA*, August 1997.

Lewis, Mathiske 99: Lewis, B. and Mathiske, B. "Efficient Barriers for Persistent Object Caching in a High-Performance Java Virtual Machine". Sun Labs Technical Report TR-99-81, Sun Microsystems Laboratories, 1999.

Printezis et al. 97: Printezis, T., Atkinson, M., Daynès, L., Spence, S., and Bailey, P. "The Design of a Scalable, Flexible, and Extensible Persistent Object Store for PJama". In: *Proceedings of the 2nd International Workshop on Persistence and Java (PJW2), Half Moon Bay, CA, USA*, August 1997.

Printezis 00: Printezis, T. *Management of Long-Running High-Performance Persistent Object Stores.* PhD thesis, University of Glasgow, May 2000.

Schkolnick 77: Schkolnick, M. "A Clustering Algorithm for Hierarchical Structures". *ACM Transactions on Database Systems*, 2(1):27–44, May 1977.

SPEC 98: "The SPECjvm98 Benchmarks".
http://www.spec.org/osg/jvm98, August 1998.

White, Garthwaite 98: White, D. and Garthwaite, A. "The GC Interface in the EVM". Sun Labs Technical Report TR-98-67, Sun Microsystems Laboratories, 1998.

White 94: White, S. *Pointer Swizzling Techniques for Object-Oriented Database Systems.* PhD thesis, Department of Computing Science, University of Wisconsin–Madison, Madison, Wisconsin, 1994.