5-2-1998

Straightforward Java Persistence Through Checkpointing

Jon Howell
Dartmouth College

Straightforward Java Persistence Through Checkpointing

Jon Howell[*]

Technical Report PCS-TR98-330[†]
Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
jonh@cs.dartmouth.edu

May 2, 1998

Abstract

Several techniques have been proposed for adding persistence to the Java language environment. This paper describes a scheme based on checkpointing the Java Virtual Machine, and compares the scheme to other techniques. Checkpointing offers two unique advantages: first, the implementation is independent of the JVM implementation, and therefore survives JVM updates; second, because checkpointing saves and restores execution state, even threads become persistent entities.

1 Introduction

Previous papers at this workshop have outlined a variety of strategies for providing persistence for Java programs. Moss and Hosking provide a taxonomy for categorizing these proposals according to the persistence programming model and implementation [MH96]. Our proposal is unlike most in that even thread state is made persistent, and it requires no changes to the Java virtual machine.

This paper will describe our scheme in the context of Moss and Hosking's taxonomy, outline existing persistence proposals, and give a status report on the existing implementation.

2 Description

Existing models provide persistence to Java one of three ways: They add persistence with language-level mechanisms, they alter the JVM to add support for persistence, or they run the whole JVM over an operating system that supports persistence. These options are diagrammed in Figure 1.

Our model differs in that it inserts a checkpointing layer between the JVM and a traditional OS (in our case, Solaris). See Figure 2. A library routine, in the form of a native method that commences a checkpoint, is provided to give the application explicit control over checkpointing when desired.

We now analyze our model within the Moss and Hosking framework.

[*]Supported by a research grant from the USENIX Association.

[†]Submitted to the Third International Workshop on Persistence and Java, Tiburon, CA, September 1998. Available at `http://www.cs.dartmouth.edu/reports/abstracts/TR98-330/`

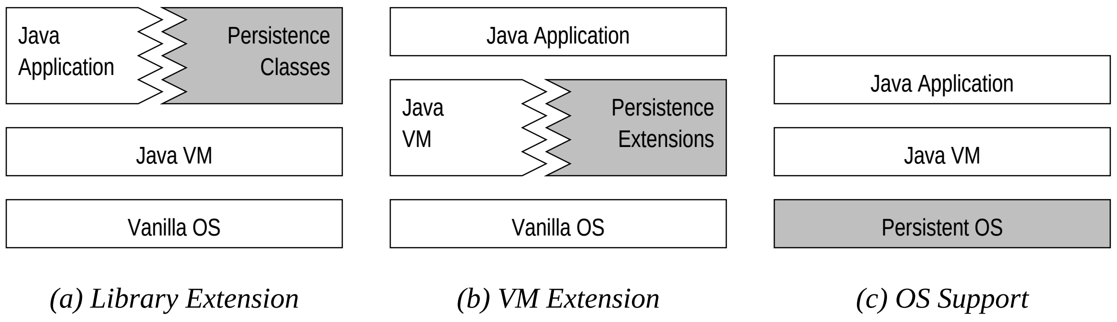

Figure 1: Existing proposals for Java persistence

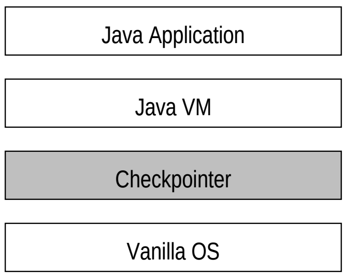

Figure 2: Our checkpointer-based scheme

M0: Is persistence by reachability? Persistence is by existence: the entire VM is checkpointed, therefore every object is persistent. Admittedly, this can be inefficient for certain classes of application with large amounts of temporary state. The *libckpt* package allows programmers to specify application data that need not be made persistent [PBKL95], but it is not clear yet how we might support this idea without modifying the JVM.

M1: Is object code persistent? Yes, since classes are loaded into the heap.

M2: Is program execution state persistent? Yes, thread stacks are included in the checkpoint file.

M3: What is the transaction model? One can emulate transactions in our system by arranging a checkpoint at each transaction commit. To rollback, one would abort the process and restart from the previous checkpoint. Obviously, the system is not designed for fine-grained transactions.

T1: Is the source language changed? No.

I1: Is the Java compiler changed? No.

T2: Is the object language changed? No.

I2: Is the Java interpreter and run-time system modified? Trivially. The wrapper for the Java interpreter *java* is replaced with a new wrapper that is linked with the checkpointing library. A native method is provided that invokes a checkpoint operation in the library.

2.1 Implementation challenges

When we began this checkpointing system, we were hoping we could simply apply an existing generic checkpointer to the JVM. However, it turns out that the Java runtime depends on much more process state than we expected.

2.1.1 Generic checkpointing

A traditional checkpointing system, designed to protect long-running data-crunching applications, needs to save only a few components of process state. Memory is assumed to consist only of the text segment, data segment, heap, and stack. File descriptors are assumed to point to regular files, so it suffices to save the seek pointer of each open file. File contents are synchronized (to flush the buffer cache to disk) at each checkpoint, but are not themselves checkpointed, so this technique handles read-only files and those written sequentially.

2.1.2 Memory segments

The JVM depends on much more process state. First, the memory map is a good deal more complicated than the four basic segments. Each thread is given two memory segments for its Java and native stacks, and each shared library is served by a read-only segment and a read-write segment for data. We use the `/proc` debugging interface to extract the list of segments and their contents.

2.1.3 Fighting with `green_threads`

The `green_threads` package used to implement the JVM overrides many symbol names from *libc* to provide alternative implementations of system functions such as `open`, `read`, and `dup`. Our checkpointer, which is meant to be built as a layer below the entire JVM, must be careful to access the system services directly.

`green_threads` also makes `ioctl` calls on file descriptors to arrange for nonblocking behavior and to request signals to indicate when the descriptors are ready to be accessed. We must be careful to save and restore that information about each file descriptor, lest `green_threads` become confused at recovery time.

2.1.4 Sockets and other strange file descriptors

The most challenging aspect about writing a checkpointer for the JVM is to handle bits of state that are unrestorable, such as filehandles on open sockets.

Because we cannot restore open socket connections, we need a clean way to persuade the JVM (in particular, `green_threads`) that the socket has spontaneously closed. To accomplish this, we restore the application with dead sockets attached to the file descriptors associated with sockets in the checkpointed process.[1] This technique works as desired for code using the `java.net.Socket` class. An obvious consequence is that the application program must be prepared to recover in a useful way from an exception delivered due to a lost socket connection.

So far, however, the `awt` package seems to immediately call `exit()` upon discovery of an unexpectedly closed filehandle, and RMI's behavior is even more mysterious. We are currently looking into why our deception is not fully effective.

[1] Our current recovery strategy is to open a connection to the 'daytime' service, read out the date, and then substitute that nearly-dead socket for the open socket in the original process. The `green_threads` package discovers that the socket is now closed, and the JVM translates that state into a `java.io` exception.

2.2 Comparison to other approaches.

Our checkpointing approach to Java persistence has two significant disadvantages. First, it is limited by the size and performance of virtual memory on the host operating system. On systems with 32-bit pointers, often only one or two gigabytes of virtual address space is available for user data. To significantly relieve the size restriction, our system would need to run on a machine with a 64-bit pointer size, and we would need access to a user-level mechanism for backing memory pages with user-allocated files rather than the system-allocated swap partition.

The second disadvantage is the absence of any mechanism for transactions and logging. If the application uses some library-level transaction mechanism, the persistence mechanism does not violate its atomicity. But to make the soft transactions durable, one must invoke a checkpoint with each transaction commit. One can expect the page-size granularity of the checkpointer to be substantially less efficient than write-logged transactions.

3 Related Work

Several other works have proposed different techniques for providing persistence extensions to Java. Those mentioned here are categorized according to Figure 1.

3.1 Library Extension

Some systems are able to avoid any modification to the JVM, but they sacrifice some measure of transparency. Classes must be explicitly declared potentially persistent. Persistent Java maps Java objects to a database via JDBC. The system is implemented as a Java class library, and involves no changes to the compiler or the JVM [dST96].

The JSPIN system provides persistence through a mapping to an object-oriented database. JSPIN also avoids modifying the JVM, but requires processing any potentially persistent classes through a modified compiler [RTW97, WKMR96].

The ObjectStore PSE system uses a bytecode postprocessor to insert residency and update checks into methods for potentially-persistent classes [O'B96, LLOW91].

Concordia, an infrastructure for mobile agents, employs Java serialization facilities to provide persistence for agent code and data [WPW$^+$97].

3.2 Virtual Machine Extension

The PJama project extends the JVM by adding a persistent object pool alongside the original transient object heap. Objects are moved out to a buffer pool of storage made persistent using the Recoverable Virtual Memory (RVM) package [Spe96, SA97, Jor96, ADJ$^+$96, PAD$^+$97]. The PJama authors lament making "a succession of ports of our technology between different versions of the JVM, an activity that will not diminish." [PAD$^+$97]

Moss and Hosking describe a new Java interpreter that they expect to base on their Persistent Smalltalk system [MH96].

Transactions for Java extends the JVM to log changes to the heap, and back them to stable storage using RVM [GN96].

The Persistent Java project from IBM modifies a JVM to simulate a large address architecture, even on 32-bit hardware. That address space is made persistent by shared address space subsystem [Mal96, JMN$^+$97].

3.3 Operating System Support

We are aware of only one proposal that would make threads persistent. Dearle et. al. suggest implementing Java on top of the persistence provided by the Grasshopper operating system. They describe how Grasshopper could be used below a JVM (or other language) to provide transparent persistence support, without modifying the runtime language system at all [DHF96, Dea97].

Libckpt is a portable checkpointing package, but is only aware of a three-segment memory map and simple files [PBKL95]. It was the inspiration for the checkpointer described in this paper, but because the JVM depends upon much more detailed information in the process state, our checkpointer is more complex and currently not portable.

4 Current Status

The current system can checkpoint and restore the JDK 1.1.5 JVM on SPARC hardware running Solaris 2.5.1. It successfully checkpoints programs running multiple threads. A Java program that has open sockets will receive an exception on those sockets at recovery time, so that application-specific code can rebuild an appropriate network connection.

Our goal in implementing the checkpointer was to provide a persistent object repository reachable by RMI. As we mentioned above, our technique for restoring file descriptors that once pointed to open sockets is not capable of convincing `awt` or RMI that it has simply encountered a mysteriously closed file descriptor. We continue to search for ways to convince RMI and `awt` to handle the recovery gracefully.

Figure 3 gives the cost of checkpointing a trivial Java program and the `javac` compiler. The former produced a 5.3MB checkpoint image, the latter produced a 6.0MB image. In the *sync* case, the checkpoint image is flushed to disk before continuing; in the *swap* case, the checkpoint image is written to `/tmp`, which is not backed to nonvolatile disk on Solaris. The maximum coefficient of variation was 0.08.

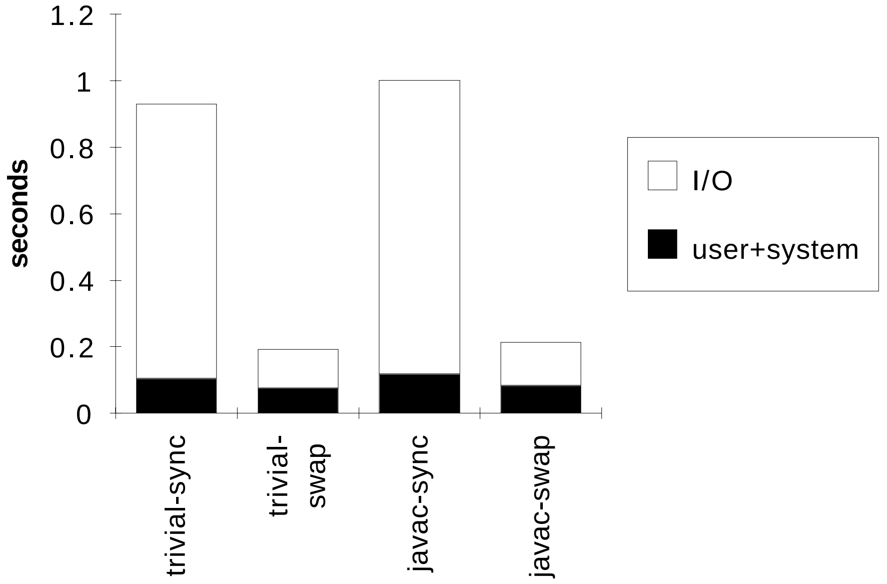

Figure 3: Checkpointer performance

5 Future Work

The *libckpt* package employs several common optimizations such as incremental and main-memory checkpointing, which should be fairly easy to incorporate into our system. It also introduces *user-directed checkpointing*, which allows the programmer to specify regions of memory that do not need persistence [PBKL95]. This latter feature is impractical in our system because it would violate the safety of Java, and an implementation would almost certainly require modifying the JVM. Also, the technique is most effective for scientific systems, where the programmer can easily make statements about the usefulness of large arrays of simple datatypes.

Currently, we write the image of every memory segment to disk. One space optimization would be to avoid writing read-only segments, such as the program text and the read-only segment of shared libraries. However, it appears that the runtime link editor modifies the code segments of shared libraries, so that they do not match the `.so` file they were loaded from. Perhaps there is a way to avoid saving the link-edited versions of the libraries, knowing that they could be recovered automatically later.

The prototype uses the Java Native Interface to attach the checkpointing code to an invocation of the Java Virtual Machine. It is conceivable that we could load the checkpointing library dynamically (as a native method), allowing the end-user to invoke even persistent programs with just the `java` command, rather than our custom-linked wrapper.

The current system depends on several features of Solaris. We expect the amount of effort involved in a port to another operating system to be commensurate with the amount of effort involved in porting `green_threads`, simply because we need to checkpoint any OS-specific state that `green_threads` depends upon.

6 Conclusions

By checkpointing a Java Virtual Machine, we provide persistence to the Java language environment. Unlike other persistent Java systems, our scheme has the advantages of being able to run on a commodity operating system, making thread state persistent, and not requiring any modifications to the Java VM. The latter feature is important because it means that our scheme should work without change when future versions of the JDK are released.

Availability

When the RMI problem is solved to our satisfaction, we plan to distribute the package freely. Interested parties may contact the author via e-mail to inquire about the status of the package.

Acknowledgements

Thanks to my advisor David Kotz for guidance in the project and his always thoughtful comments on writing. Sun Microsystems provided software used in this project.

References

[ADJ+96] Malcom Atkinson, Laurent Daynes, Mick Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.

[Dea97] Alan Dearle. Persistent servers + ephemeral clients = user mobility. In *Proceedings of the Second International Workshop on Persistence and Java*, August 1997.

[DHF96] Alan Dearle, David Hulse, and Alex Farkas. Persistent operating system support for Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[dST96] C. Souza dos Santos and E. Theroude. Persistent Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[GN96] Alex Garthwaite and Scott Nettles. Transactions for Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[JMN$^+$97] Maynard P. Johnson, Steven J. Munroe, John G. Nistler, James W. Stopyro, and Ashok Malhotra. Java(tm) persistence via persistent virtual storage. In *Proceedings of the Second International Workshop on Persistence and Java*, August 1997.

[Jor96] Mick Jordan. Early experiences with persistent Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[Mal96] Ashok Malhotra. Persistent Java objects: A proposal. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[MH96] J. Eliot B. Moss and Tony L. Hosking. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[O'B96] Patrick O'Brien. Java data management using ObjectStore and PSE. Object Design, Inc. white paper, November 1996. Available at `http://www.odi.com/content/white_papers/javawp1.html`.

[PAD$^+$97] Tony Printezis, Malcolm Atkinson, Laurent Daynes, Susan Spence, and Pete Bailey. The design of a new persistent object store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java*, August 1997.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 USENIX Technical Conference*, pages 213–224, January 1995.

[RTW97] John V. E. Ridgway, Craig Thrall, and Jack C. Wileden. Toward assessing approaches to persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java*, August 1997.

[SA97] Susan Spence and Malcolm Atkinson. A scalable model of distribution promoting autonomy of and cooperation between PJava object stores. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences*, pages 513–522, 1997.

[Spe96] Susan Spence. Distribution strategies for Persistent Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[WKMR96] Jack C. Wileden, Alan Kaplan, Geir A Myrestrand, and John V.E. Ridgway. Our SPIN on persistent Java: The JavaSPIN approach. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996.

[WPW$^+$97] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. *Concordia:* an infrastructure for collaborating mobile agents. In *Mobile Agents. First International Workshop, MA '97 Proceedings*, pages 86–97, 1997.