# Straightforward Java Persistence Through Checkpointing

Jon Howell*

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
jonh@cs.dartmouth.edu

August 6, 1998

**Abstract**

Several techniques have been proposed for adding persistence to the Java language environment. This paper describes a system we call `icee` that works by checkpointing the Java Virtual Machine. We compare the scheme to other persistent Java 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. We needed persistence for Java objects, and none of the systems we reviewed seemed suitable for our purposes. One important requirement was persistent threads. To make this a simple project, a second requirement was that the implementation did not involve modifying the Java Virtual Machine.

We first discuss the semantics provided by our checkpointer, and then look at its functionality, implementation and performance. We propose future work, and then provide an overview of related persistent Java systems and checkpointing systems.

## 2 Semantics of the checkpointer

Our system provides persistence by checkpointing a running JVM process. This technique gives rise to a unique set of persistence semantics. Moss and Hosking provide a taxonomy for categorizing persistent systems [MH96]. Figure 1 characterizes our checkpointer according to their taxonomy, to put it into the context of the persistent Java literature.

The following semantic consequences arise from our checkpointing persistence scheme:

1. Persistence is type-orthogonal, includes threads, and is determined by reachability.
2. External state such as files and communications links are not transparently restored.

*Supported by a research grant from the USENIX Association.

In Advances in Persistent Object Systems, Ron Morrison, Mick Jordan, and Malcom Atkinson, eds. Pages 322-334. Morgan Kaufmann, 1999.
URL: http://www.cs.dartmouth.edu/~jonh/research/

3. Checkpointing restricts how easily data can migrate to new JVMs or platforms.
4. The size and performance of a checkpointed JVM is restricted by our reliance on a system's paging mechanism.

Each characteristic is developed in the following subsections.

## 2.1 Type orthogonality, thread persistence, and reachability

Because it checkpoints the entire VM, persistence is orthogonal to type. For the same reason, threads are persistent. Furthermore, because every object is persistent, it trivially follows that every object reachable from a persistent object is persistent. Thus programmers need not worry about explicitly making objects persist, which is the point of persistence by reachability. However, the programmer has no way to prevent objects from persisting, which will be inefficient for certain classes of application.

## 2.2 Unrecoverable external state

Of course, any state external to the virtual machine is not recoverable. This state includes open files and communications channels. For example, on recovery we reopen and restore the file pointer of file descriptors pointing to simple files. However, we do not roll back any changes made to the file data itself.

Of greater interest to us are communications links. We do not aim to provide a distributed checkpointing system, so we cannot roll back the state of remote entities, nor can we, in general, even transparently reopen closed links to the remote entity. Therefore, the only alternative is to let the link appear forcefully closed to the persistent process. Applications that handle unexpectedly closed connections should be able to reestablish communications with their remote counterparts. Some applications will share state across virtual machines, so restoring one machine to a previous point would produce an inconsistent state. We presume that such distributed applications will need to provide their own support (possibly by explicitly controlling our single-process checkpointing mechanism) for distributed consistency.

---

**M0: Is persistence by reachability?** Every object is persistent, so there are no distinguished persistent roots.

**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?** Our system implements persistence, not explicit transactions. Nettles and Wing describe how to build transactions out of persistence and 'undoability.' [NW91]

**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 `javai` 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.

---

Figure 1: Our checkpointer categorized according to Moss and Hosking's taxonomy

### 2.3 Data durability

Once data is stored into a VM made persistent with our checkpointer, what sort of changes can it endure? The actual checkpoint data is a flat file, comprising a memory image of the checkpointed process. Embedded in that image is information about the process' state that is hidden in the kernel, including the status of open file descriptors. Included in the image is the application's thread and heap state, the JVM itself, the libraries that were linked against the program, and even the checkpointer itself. The consequences of this are several:

1. The image cannot be transferred between heterogeneous systems. A different CPU architecture, or even a different system call interface, would be incompatible with the process in the image. Therefore, the data (in checkpointed form, at least) cannot migrate off of the platform it was created on.

2. Similarly, because the JVM is frozen, the data cannot be transparently migrated to a higher JVM version.

3. Changing class definitions for the data would be difficult as well, equivalent to modifying classes in a running JVM or in a store that includes class definitions to provide consistency between data representation and method implementation.

Most systems that save the class definitions with the persistent data face the same class-evolution problem, so a solution suitable for other systems may be applicable to ours.

### 2.4 Virtual address space limitations

The data in a checkpointed JVM is limited by the size and performance of virtual memory on the host operating system. On 32-bit systems, often only one or two gigabytes of virtual address space is available for user data. To 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.

## 3 Functionality of the checkpointer

We next outline the procedure used to establish checkpoints and to recover from failures, represented graphically in Figure 6. We also show the user and programmer interfaces to the system.

### 3.1 Checkpointer operation

When some thread invokes the Java native method `Checkpoint.Control.doCheckpoint()`, the checkpointer's native method executes the following procedure:

1. Signals are masked to halt the other threads.

2. Information about kernel-hidden process state and open communications channels is stashed in the data segment.

3. All memory segments are scanned and written to a temporary file along with an index that describes how to reconstruct them. Most of the work is one large `write()` call per contiguous memory region.

4. The temporary file is atomically renamed to replace the previous checkpoint file.

5. Signals are unmasked to resume the other threads.

The atomic replacement of one checkpoint file with the next (a two-phase commit) prevents opening a window of vulnerability during the checkpoint operation.

### 3.2 Checkpointer invocation

Currently, Java applications that require persistence must be started using a specially linked `javai` replacement:

`% java myprog args` becomes `% icee myprog args`

The java program may explicitly call `Checkpoint.Control.doCheckpoint()` to establish a checkpoint. Alternatively, `icee` can be invoked using a wrapper class, `Checkpoint.Auto`, that provides periodic checkpointing and a callback interface to notify interested objects when a recovery has taken place. The command

```
% icee Checkpoint.Auto -period=60 myprog args
```

executes `myprog(args)`, taking a checkpoint every 60 seconds.

### 3.3 Target program recovery

When the program fails, the user may restart it from the most recent checkpoint by invoking:

```
% icee -recover
```

This is implemented as follows:

1. Signals are masked to stop the other threads.

2. To reconstruct the checkpointed memory map, each memory region in the checkpoint file is `mmap()`ed into the appropriate location in the new process.

3. File descriptors are reopened according to the stashed information now available in the data segment.

4. Kernel-hidden process state is restored.

5. Signals are unblocked, causing the thread system to resume. Of course, the threads that resume are those that were encoded in the checkpoint, not those present at step 1.

## 4 Implementation of the checkpointer

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. We describe the basic checkpointing procedure, followed by four challenges we encountered in checkpointing a complex process such as the JVM.

### 4.1 Basic checkpointing

A basic 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.

### 4.2 Challenge: detecting all 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.

It turns out that `/proc` fails to mention *reserved* pages that are mapped but unallocated.[1] These pages are used by thread stacks, and appear, zero-filled, as soon as they are accessed. We interpose on all `mmap` calls to acquire this information.

### 4.3 Challenge: interposing on library functions

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.

Furthermore, we interpose on some system calls to collect information that is not available at checkpoint time from `/proc`. Currently, that means we have to provide a specially-linked `javai` substitute, a simple program that calls the Java Native Interface. We would prefer to have a native class checkpointer that loads entirely from a shared library, but by that point in the process' life, we have missed the opportunity to interpose on library calls.[2]

### 4.4 Challenge: extracting hidden process state from the kernel

The `green_threads` package 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. This process state is hidden in the kernel, not in the process memory map, so we must be careful to save and restore it explicitly, lest `green_threads` become confused at recovery time.

### 4.5 Challenge: handling sockets and other strange file descriptors

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

[1] In Solaris 2.5. Solaris 2.6 does provide information on mapped but unallocated pages.

[2] One possibility would be to use the regular `javai`, but provide a modified `libc.so` via the `$LD_LIBRARY_PATH` environment variable.

the checkpointed process.[3] This technique works as desired for code using the `java.net.Socket` class, including RMI code. 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.

The `awt` package seems to immediately call `exit()` upon discovery of an unexpectedly closed filehandle. We are looking into how we might fully deceive `awt`.

## 5 Performance of the checkpointer

We have just described how the checkpointer is implemented. In this section, we examine the cost of taking checkpoints, and the overhead experienced during normal execution.

### 5.1 Cost of checkpointing

To roughly evaluate the cost of checkpointing, we measured the CPU time and wall-clock time of checkpointing two applications. One is a trivial Java application (denoted trivial in the figure), which produced a 5.3MB checkpoint image. The other was Java compiler (denoted javac in the figure), which produced a 6.0MB image. For each program, we compare synchronous checkpointing (sync), wherein the time includes waiting for the disk write to complete, and asynchronous checkpointing (async), where we write the checkpoint to `/tmp`, which is not backed to nonvolatile disk on Solaris.

The results are presented in Figure 2. The total height of the bars is the wall-clock time, so the white sections represent time the application spends waiting for I/O. The maximum coefficient of variation was 0.08. The async values represent main-memory checkpointing, wherein we defer writing to disk until after the program is allowed to continue. In that case, the latency seen by the target program due to checkpointing is around 0.2 seconds.

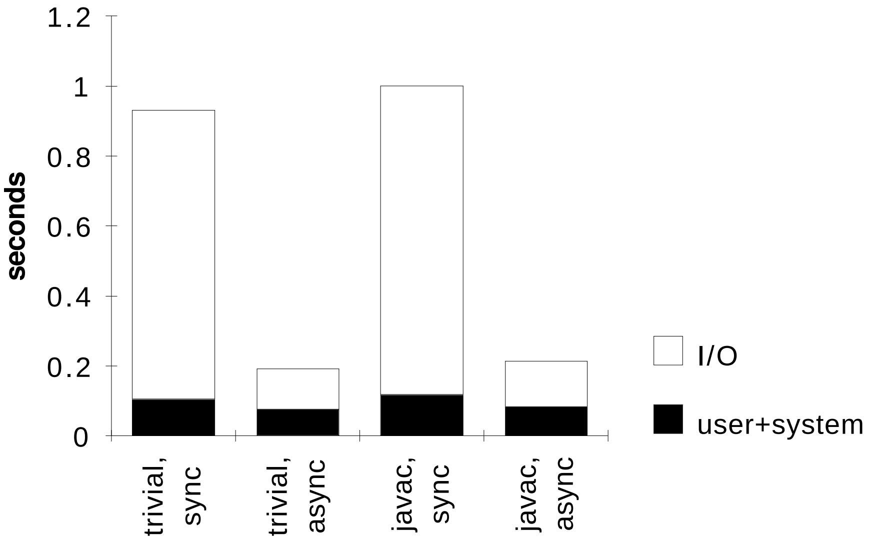


Figure 2: Checkpointer delay is reduced by deferring disk writes.

[3] 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 condition into a `java.io` exception.

## 5.2 Runtime overhead of checkpointing

We expected overhead of checkpointing during normal execution to be low, since currently the only costs are the interposition on the `open()` and `mmap()` system calls. We measured this by invoking the Java compiler on a simple class five different ways. The results presented in Figure 3 are the *user* + *sys* times for one compilation of the simple class. We ran about forty trials of each case, in random order, on an unloaded system, with all jobs fitting in main memory. The maximum coefficient of variation was 0.06.

In the first case, javac, we directly invoked the Java compiler on the target class. For the remaining four cases, we wrote a wrapper class with a `main()` method that invokes `sun.tools.javac.Main.compile()`. The wrapper class allowed us to invoke the compiler from the normal Java interpreter (the java bar), and from our modified interpreter.


Figure 3: The checkpointer introduces negligible runtime overhead.

The icee-nockpt bar represents invoking the wrapper with `icee`, but without taking a checkpoint. The icee-initial bar represents invoking the wrapper with `icee`, and taking a checkpoint before exiting. We are not certain why java and javac took so much longer to start up than icee-nockpt, which amounts to a simple Java Native Interface invocation, but it may be partly due to the shell scripts that wrap the `javai` and `javac` binaries.

The icee-recover bar represents restarting the checkpoint taken by icee-initial to invoke the compiler again. This case shows how taking a snapshot of the Java compiler after it has completed the CPU-intensive task of loading and verifying all of its classes dramatically shortens startup time for future compilations [Jor96]. The same concept has been applied in many contexts; Section 7.2.2 mentions some examples.

# 6 Future work on the checkpointer

We have several ideas for improvements to our checkpointer. Plank's *libckpt* package includes two several common optimizations we would be silly to overlook [PBKL95]. We may exploit shared libraries to save time and disk space. We may simplify the tool by packaging it completely as a native class. We may provide a way for our data to persist beyond class and JVM upgrades. We may modify `icee` to allow the creation of persistent data repositories. And finally, we may port the system to other platforms.

## 6.1 Asynchronous checkpointing

One can reduce the latency of a checkpoint by allowing the target process to proceed as soon as all of its pages have been marked to be checkpointed. This requires marking the pages copy-on-write, to avoid writing an inconsistent checkpoint. Happily, the `fork()` call does just this. We plan to use `fork()` to make writing checkpoint pages asynchronous with program execution.

## 6.2 Incremental checkpointing

On the second and subsequent checkpoints, we can reduce substantially the number of pages we need to write by only writing those that have changed since the prior checkpoint.

## 6.3 Exploiting shared libraries

Currently, we write the image of every memory segment to disk. If we could avoid writing out the program text and the read-only text of shared libraries, the initial checkpoint would both be faster and consume less space on disk. 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.

## 6.4 Improving invocation

In the implementation section, we mentioned how our prototype requires invoking programs with a `javai` substitute that is linked with the checkpointer libraries. This requirement arises because we need to log invocations of two system calls, `mmap()` and `open()`. The need for the former is obviated by improved `/proc` functionality in Solaris 2.6. The latter is required to reopen disk files, since at checkpoint time, `/proc` only reports an open file's inode number, but opening a file requires a pathname. Eliminating the interposition requirement would make the checkpointer much more convenient to use.

Currently, recovery requires the checkpoint image and a copy of the original `icee` binary that generated it. We would like to make the checkpoint image files executable, so they are completely self-contained.

## 6.5 Handling the class and JVM evolution problem

As we mentioned in Section 2.3, data in one of our persistent stores is "stuck" there along with its associated virtual machine and class definitions. When we need to use the same data with evolved classes or an upgraded JVM, we expect to migrate the data out of the old store and into the new using the `Serializable` mechanism.

## 6.6 Creating a persistent repository

It is sometimes desirable to run many independently developed programs against a persistent repository. Here is a proposal that could achieve this goal. First, we would need to return a value from `doCheckpoint()` indicating whether a restore has just occurred. Second, we would need to have the recovery process somehow stash its argument list and pass those in to the Java program as additional input. With these two features, we could create a repository and use it the first time with:

```
% icee RepositoryClass myprog myargs
```

The `RepositoryClass` would invoke `myprog.main(myargs)`. Then `RepositoryClass` would force an immediate checkpoint and exit the JVM. To run another program against the repository, we would invoke:

```
% icee -recover myprog2 myargs
```

The recover operation would reawaken the original invocation of `RepositoryClass`, and inform it of the recovery so it does not merely exit again. The `RepositoryClass` would somehow acquire the new command-line arguments, and invoke `myprog2.main(myargs)`.

### 6.7 Porting to other platforms

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.

## 7 Related Work

Our checkpointer lies at the intersection of the persistent Java world and the checkpointing world. The first subsection categorizes persistent Java approaches according to the layer of software they modify. The second section examines previous uses of checkpointing.

### 7.1 Approaches to persistent Java

Existing approaches alter the Java execution environment at one of three layers: 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 4. In the sections following, we look at each approach in turn. Our model differs in that it inserts a checkpointing layer between the JVM and a traditional OS (in our case, Solaris). See Figure 5.

#### 7.1.1 Library extension

Some systems provide persistence by supplying class libraries to do the job, or by rewriting or specially compiling application class files to implement persistence. See Figure 4(a). An advantage of this approach is a durable implementation that does not modify the JVM and is portable. The disadvantage is that such systems can not support persistence by reachability, type-orthogonality, or thread persistence.

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]. Classes in Persistent

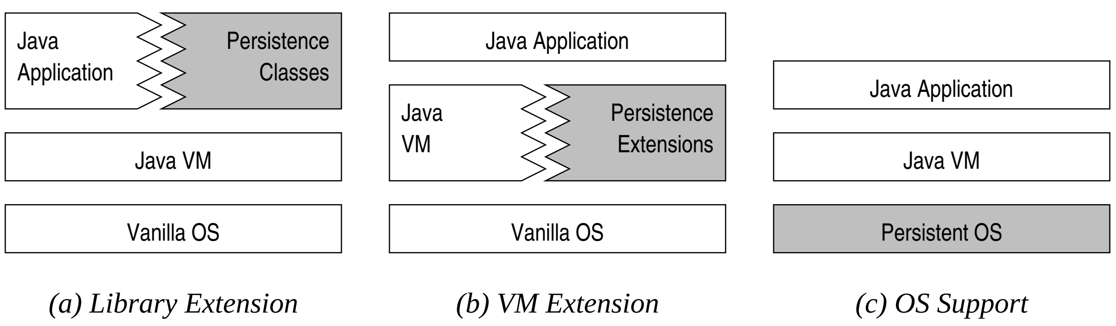


Figure 4: Existing proposals for Java persistence fall into three categories.

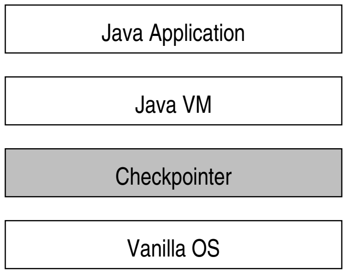


Figure 5: Our checkpointer-based scheme inserts a layer between the Java VM and the operating system.

Java must be declared persistent.

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].

### 7.1.2 Virtual machine extension

A second approach is to modify the Java Virtual Machine to provide type-orthogonal persistence to applications. See Figure 4(b). Such systems are tightly coupled to the JVM, which is both an advantage and a drawback. The advantage is that they may store persistent data in a portable format that can survive JVM updates or even cross-platform migration. The drawback is that the implementation of such a system must be modified to track each change to the JVM implementation. The authors of PJama, for example, lament making "a succession of ports of our technology between different versions of the JVM, an activity that will not diminish" [PAD+97].

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].

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].

### 7.1.3 Operating system support

A third approach to Java persistence is to run the JVM and Java application on top of a persistent operating system. See Figure 4(c). The advantage of such an approach is ubiquitous, orthogonal persistence without modification to the JVM. A drawback is compatibility: most end users do not have the freedom to replace their conventional operating system with a persistent one.

We are aware of only one proposal that would make threads persistent. Dearle *et al.* suggest implementing Java on top of the Grasshopper persistent 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, RDH+96].

## 7.2 Checkpointing

Lee and Anderson provide a thorough introduction to fault tolerance; they cover checkpointing in chapter seven in the context of backward error recovery [LA90]. Cristian gives an overview of fault tolerance [Cri91]. He discusses how checkpointing and rollback provide failure masking for individual servers and server groups. Bowen and Pradhan discuss applications of checkpointing throughout the memory hierarchy [BP93]. The following two sections discuss extensions of checkpointing to distributed systems and some interesting uses of checkpointing.

### 7.2.1 Distributed checkpointing

Strom and Yemini wrote a seminal paper on distributed checkpointing using message logs to recover consistent states from asynchronous checkpoints [SY85]. Li, Naughton and Plank implemented a real-time, concurrent checkpointing system for parallel applications [LNP90].

Chiueh describes a hardware optimization for checkpointing [Chi93]. By preventing checkpoints from slowing down the target process, Chiueh's design enables a checkpoint-per-message approach to consistent distributed checkpoints, which is the most straightforward solution to the nondeterminism and cascaded rollback problems encountered in distributed checkpointing.

POGS is a checkpoint coordinator, a CORBA service that assists a distributed application in taking globally-consistent checkpoints [Zwe97].

### 7.2.2 Uses of checkpointing

Morin and Puaut discuss the application of checkpointing to distributed shared virtual memory systems [MP97]. DSVMs are an interesting design point for checkpointing because of their implicit communication patterns that differ from those of explicit message-based systems. ICARE, for example, is a checkpointed DSM that can improve execution time of an application because the pages replicated to form a checkpoint can serve as read-only replicas [KMB98].

Smith and Ioannidis implemented remote fork, a process migration system for Unix [SI88]. It works by checkpointing, transferring the checkpointed image to another machine, and restarting the image on the destination host. Their migrated processes are "deaf and dumb;" that is, they do not attempt to handle the disconnected file descriptors. The reader interested in process migration will find a number of useful citations in that paper.

Some languages such as Smalltalk and Self use snapshots as a way to save the state of the virtual machine and provide a persistent application programming environment. Emacs and TeX both dump core during their installation process, and postprocess the core file to produce a runnable image that saves some initial processing in future invocations. Perl makes the same dump/reincarnate functionality available to the programmer.

Checkpointing enables an intriguing mode of program debugging known as reverse execution. A sidebar in Bowen's overview describes this process [BP93].

## 8 Conclusions

By checkpointing a Java Virtual Machine, we provide persistence to the Java language environment. Our checkpointer offers a unique set of advantages: it provides type orthogonality, persistence by reachability, and thread persistence; it does not require modifying the JVM (or tracking changes to it); and it runs on a commodity operating system. It suffers some disadvantages: it is difficult to migrate persistent data to new JVMs or platforms; it is inefficient because all data is made persistent; and it does not offer support fine-grained transactions. Overall, it occupies an interesting and useful point in the space of persistent Java solutions.

## Availability

The version of `icee` described in this paper is available at:

`ftp://ftp.cs.dartmouth.edu/pub/jonh/icee-1.0.tgz`

## Acknowledgements

Thanks to my advisor David Kotz for guidance in the project and his always thoughtful comments on writing. Thanks to the anonymous reviewers for their thoughtful suggestions. Thanks also to Sun Microsystems for providing 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.

[BP93] N. S. Bowen and D. K. Pradhan. Processor- and memory-based checkpoint and rollback recovery. *IEEE Computer*, 26(2):22–31, February 1993.

[Chi93] Tzi-Cker Chiueh. Polar: a storage architecture for fast checkpointing. *Journal of Information Science and Engineering*, 9(1):61–80, March 1993.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[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.

[KMB98] A.-M. Kermarrec, C. Morin, and M. Banâtre. Design, implementation and evaluation of ICARE: An efficient recoverable DSM. *Software - Practice and Experience*, 28(9):981–1010, July 1998.

[LA90] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, second revised edition, 1990.

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

[LNP90] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, 1990.

[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.

[MP97] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):959–969, Sept. 1997.

[NW91] S.M. Nettles and J.M. Wing. Persistence+undoability=transactions. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 832–843, January 1991.

[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.

[RDH+96] J. Rosenberg, A. Dearle, D. Hulse, A. Lindstrom, and S. Norris. Operating system support for persistant and recoverable computations. *Communications of the ACM*, 39(9):62–69, Sept. 1996.

[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.

[SI88] Jonathan M. Smith and John Ioannidis. Implementing remote `fork()` with checkpoint/restart. Technical Report CUCS-365-88, Columbia University, 1988. Available at: `ftp://ftp.cs.columbia.edu/pub/reports/reports-1988/cucs-365-88.ps.gz`.

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

[SY85] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug. 1985.

[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 *Proceedings of the First International Workshop on Mobile Agents*, pages 86–97, 1997.

[Zwe97] M. Zweiacker. Making CORBA applications fault tolerant using checkpointing and recovery. *Ada User Journal*, 18(2):78–91, June 1997.

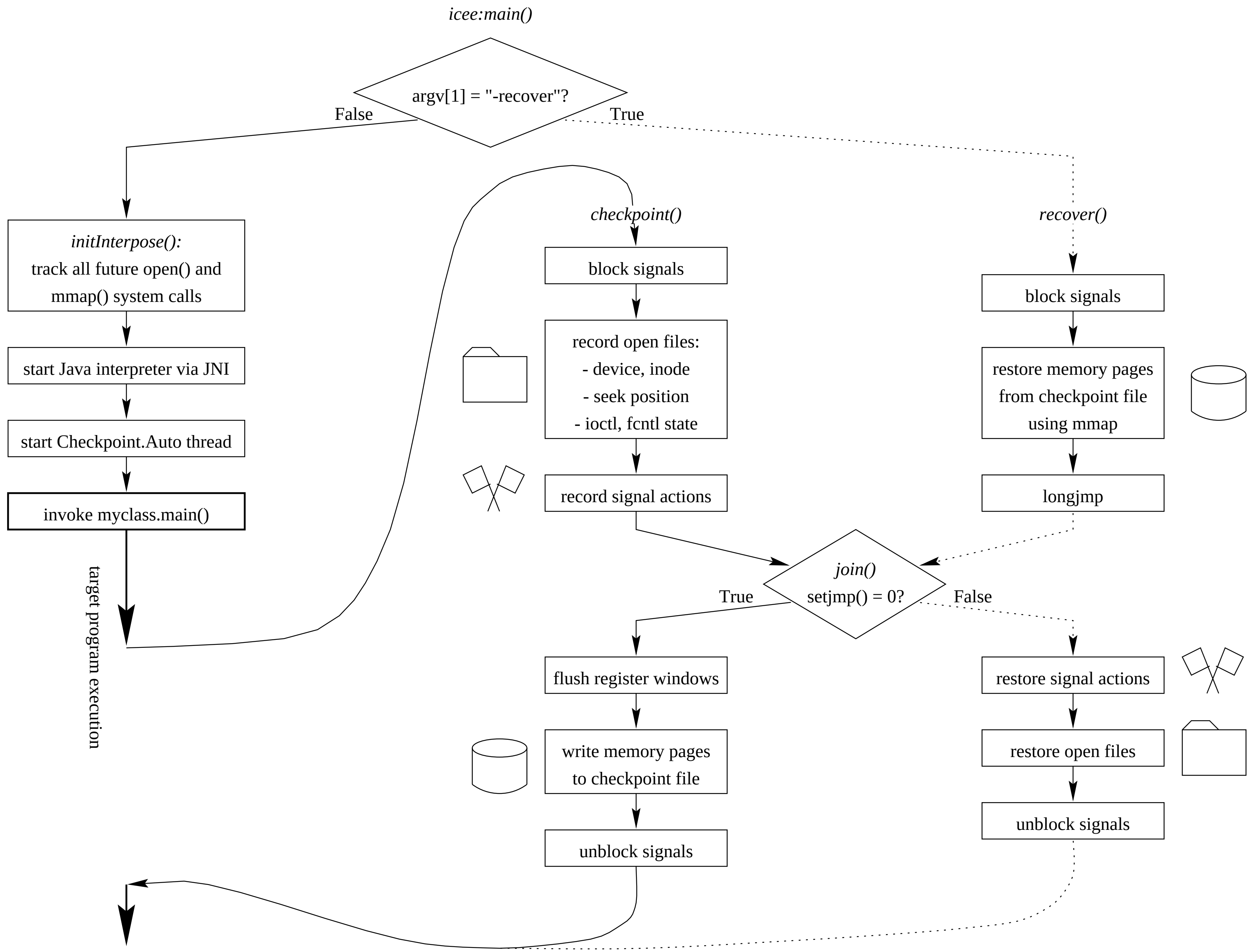


Figure 6: Flow of checkpointing and recovery operations.