Java Finalize Method, Orthogonal Persistence and Transactions

John N. Zigman and Stephen M. Blackburn*

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia
{John.Zigman,Steve.Blackburn}@cs.anu.edu.au

Abstract

Java is a popular, object oriented language that is runtime type safe. As such, it has been seen as an attractive basis for the implementation of orthogonally persistent systems by several research groups. Transactions are widely used as a means of enforcing consistency of the stable image in the face of concurrency, and have been adopted by most groups developing persistent Java systems. However, Java has a user definable *finalize* method which provides an asynchronous cleanup mechanism. The strict temporal semantics of transactions and the asynchrony of the finalize method seem at odds. This paper describes this conflict and provides a strategy for resolving the problem.

1 Introduction

The explicit management of memory becomes an increasingly difficult task as the size and complexity of a program grows. To ease the programming burden and to allow the programmer to be more productive, many automatic memory reclamation mechanisms have been developed. *Garbage collection* is the term given to the mechanism for automatically reclaiming previously allocated memory that is no longer in use. A previously allocated section of memory is considered to be unused when it is no longer reachable (via any chain of references) from any potential computation.

The Java language and the Java Virtual Machine (JVM) incorporate an automatic memory reclamation mechanism which frees the programmer from having to perform explicit memory management [Gosling et al. 1996]. Java also provides a mechanism, know as finalization, which enables the programmer to associate a cleanup method with an object type. These methods are invoked automatically at some time prior to the reclamation of an object's space. When Java is used as the basis for a persistent programming language (e.g. PJama [Atkinson et al. 1996]), the correctness implications of interactions between store and JVM level garbage collection need to be considered. The system model consists of one or more JVMs operating on a single persistent store image (see figure 1). Garbage collection is done in each JVM and the persistent store separately.

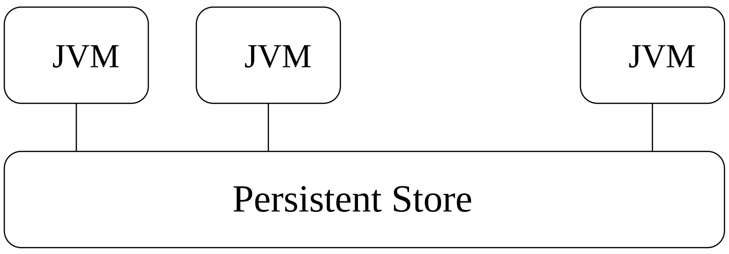

Figure 1: The JVM and Persistent Store System

Transactions are introduced to maintain coherent and consist manipulation of the persistent store in the face of concurrency. When a transactional model is introduced into the system, the interaction of

*The authors wish to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government's Cooperative Research Centers Program.

the transaction model and the cleanup mechanisms must be understood. This is particularly important given the serious issues of correctness that arise as a result of the mismatch between the arbitrariness of *finalize* invocation semantics and strict temporal notions of atomicity and serializability that are associated with transactions. In this paper we consider the interrelation of the JVM and store levels in the context of a transaction model and the cleanup mechanisms provided.

In section 2 the motivation for the *finalize* mechanism and the rules governing the mechanism are covered. Section 3 describes the conflicts that arise from the mixing of transactions and the *finalize* mechanism and section 4 describes an approach to resolve the conflict in a flexible and efficient way. Finally a brief conclusion is given.

2 Program Level Resource Management

When constructing a software system there is often a need for explicit resource management. Such resource management may, for example, take the form of a server that issues resources to many simultaneous clients. The resources provided by the server (possibly threads, class instances or computational information) may be limited for some reason. By managing a shared pool of such resources, with clients acquiring the resource only on a temporary basis, the total number of resources used can be limited.

When constructing libraries which manage a resource for a third party, it is wise not to rely on the library users to correctly release the associated resources. The provision of a method which is automatically called just prior to an object being reclaimed would allow resources to be safely released, making the system more robust.

2.1 What is *finalize* and Why use It?

Java finalize methods provide a mechanism for performing resource management operations, and so are superficially similar to C++ destructor methods. This similarity is due to the fact that both C++ destructors and Java finalize methods are invoked as part of the process of deallocating memory. The difference lies in how the invocation of the methods come about.

C++ requires programs to explicitly deallocate objects. Consequently, the invocation of C++ destructors is synchronous to a programs thread of execution. Java, however, does not require explicit deallocation of objects. Rather, it uses an independent garbage collection thread which is responsible for finding objects which are candidates for garbage collection, invoking their finalize methods and deallocating their memory. In effect, the invocation of a Java finalize method is asynchronous to the execution of the user code.

The strongest statement that can be made about the timing of finalize method invokations is that, at some stage after an object becomes a candidate for garbage collection, and prior to it being collected, the finalize method will be invoked. This unpredictable invocation time and order, means that the authors of finalize methods must take great care to ensure correct program behavior. Ill-considered use of finalize can lead to unpredictable results, such as race conditions.

2.2 When *finalize* is Invoked

The JVM is responsible for the execution of Java programs. It also encompasses the loading and unloading of classes used by the program as well as instantiation and reclamation of object instances. The JVM determines when it can invoke the *finalize* method of an object based upon the state of the object. The state of the object can be described as the cross product of reachability (*reachable*, *f-reachable*, *unreachable*) and finalization (*unfinalized*, *finalizable*, *finalized*). Figure 2 shows all the possible states of an object, and the valid transitions between those states (figure from [Gosling et al. 1996]).

The state describing the reachability of an object falls into one of three categories, these are:

- *reachable*: An object is reachable if it can potentially be accessed from any live thread.
- *f-reachable*: An object is f-reachable if it can be reached by a chain of references from some finalizable object (possibly including itself if it is finalizable), and it is not reachable.
- *unreachable*: An object is unreachable if it is not reachable and it is not f-reachable.

The automatic invocation of the finalize method and the readiness of the object to have its finalize method invoked is characterized by the finalization state of the object. There are three possible states:

- *unfinalized*: An object is unfinalized in it has not had its *finalize* method automatically invoke.

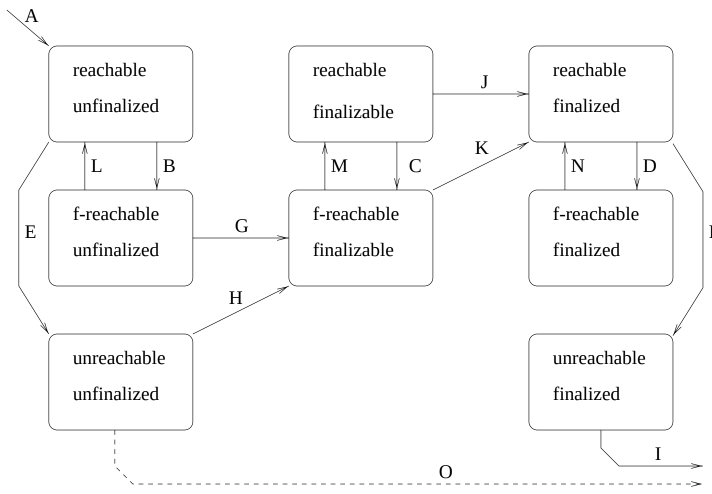

Figure 2: The object state transitions for reachability and finalizability.

- *finalizable*: An object that has not had its *finalize* method automatically invoked, but the JVM may eventually automatically invoke its finalizer.
- *finalized*: An object which has had its *finalize* method automatically invoked.

Figure 2 shows the possible state transitions. More specifically, the various transitions can occur under the following situations:

A When an object is first created it is reachable and unfinalized.

B,C,D The object ceases to be referenced by any potential calculation but is f-reachable.

E,F The object ceases to be referenced by any potential calculation and is not f-reachable.

G An unfinalized object that is f-reachable can be promoted to f-reachable and finalizable by the JVM.

H An object which is not reachable from any object, but does have a finalize method, is promoted to f-reachable and finalizable (enabling its finalize method to be invoked later).

I An object that is finalized and unreachable is no longer of any use and can be reclaimed.

J,K An object that is finalizable can have its *finalize* method run any time, at which point it becomes reachable and finalized.

L,M,N An object that is f-reachable could at any time be made reachable again by some *finalize* method executing and promoting the object to reachable.

O An object which is unreachable and does not have a *finalize* method defines is of no further use and can be collected.

In general, once an object has ceased being reachable and it either has a *finalize* method, or it can be reached from some f-reachable object, then it will slowly migrate across the diagram to the reachable and finalized state and eventually it will be collected. Beyond this constraint it is generally not possible to predict when a *finalize* method will be run; Java does not even guarantee the order in which *finalize* methods are invoked.

2.3 Transactions, Garbage Collection and Persistent Java

The context for this discussion of transactions and Java *finalizers* is the development of orthogonally persistent Java. As the concurrency control model of choice among persistent Java implementation efforts, transactions [Härder and Reuter 1983] are important.

As a practical consequence of the 'principles of persistence' for orthogonally persistent systems [Atkinson and Morrison 1995], persistence by reachability (PBR) is also important. Under PBR, the persistence of an object is a function of its reachability from some *persistent root* (rather than being a function of the object's type, for example). Garbage collection is therefore central to the efficient implementation of orthogonally persistent systems—objects not reachable from the persistent root can be reclaimed. The implementation of PBR in the context of Java is well documented by Atkinson and Morrison [1995].

3 Transactions and *finalize*

Finalization is triggered by the act of severing a reference to some object so that the object ceases to be reachable from some root. In an orthogonally persistent system such as PJama, all computation must occur in the context of a transaction [Daynès et al. 1997], therefore a finalization 'trigger' will inevitably occur within a transaction. This inevitability, the indeterminate temporal semantics of *finalize* , and the stipulation that *all* computation must occur within transactional context raises difficult questions about the semantics of *finalize* in the context of a transactional orthogonally persistent Java.

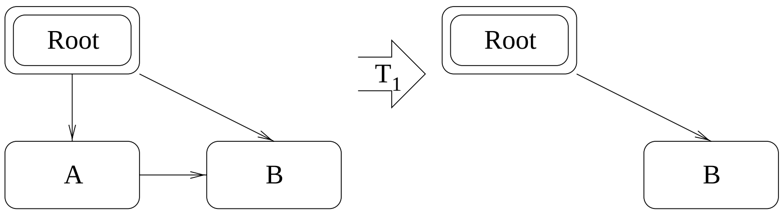

Figure 3: Example 1: Modified store

In the example illustrated in figure 3, an object that was previously held in the store (A) becomes unreachable from any persistent root, and therefore finalizable (note that it is also unreachable from any potential computation, including any current transaction). The sequence of events that perform this transformation is encapsulated in transaction T_1 (which reads the root object and then writes it back having severed the link from the root to object A). As a result, object A is been made unreachable from all persistent roots by T_1. We note however, that object A retains a reference to object B.

Significantly, T_1 has caused object A to become *finalizable* even though object A was not accessed as part of the transaction. Consequentially the *finalize* method for object A must be run, with possible side effects on other objects within the store (including object B, to which A retains a reference). However, in general it is not possible to determine when the *finalize* method will be run and so it is not possible to determine whether it will be executed within the scope of T_1.

This problem of indeterminism can be characterized as a *temporal conflict between the semantics of transactions and those of the* finalize *method invocation*. How is this situation dealt with? Does this mean that the side effects of the transaction must be dealt with immediately? Or does it mean that the side effects of the transaction can be postponed in some way? In the following section, some strategies for dealing with this situation are explored.

4 Accommodating Transactions and *finalize* Methods

We now present two possible strategies for dealing with the temporal conflict between transactions and *finalize* methods:

1. Immediate evaluation of all side effects (section 4.1).

2. A transactional *finalize* method (section 4.2).

4.1 Immediate Side Effect Evaluation

One solution to the temporal conflict is to require that the evaluation of any side effects that result from a transaction be run as part of the transaction itself. While perhaps the most semantically elegant solution, it depends on the difficult task of determining the transaction's side effects. In the case of finalization, this involves establishing whether the transaction has, through the severing of references, made some object finalizable. It is not possible to determine this except by conducting a complete garbage collection of the persistent space just prior to committing the transaction. This is therefore clearly an impractical solution.

4.2 Transactional *finalize* Method

The alternative to executing *finalize* within the scope of the transaction that triggered it is to execute it in some other transactional scope. This is not as unreasonable as it might at first seem. The 'trigger' for finalization is, after all, an act of *omission* rather than *commission*, and in general, it is not easy to determine which object was the last to effect reachability of the candidate object. The question then remains: In what context should the *finalize* be executed?

Transactions are introduced into the system to enable the maintenance of a stable and consistent view of the store in the face of concurrency. Given this, and the fact that the JVM does not guarantee when *finalize* methods will be invoked, we conclude that: **If each of the *finalize* methods is executed as a separate transaction then store consistency will be maintained without violating the operational characteristics of the Java Virtual Machine.**

Revisiting example 1, and accounting for the behavior of transactional finalization (as given above), the effects of transaction T_1 are shown in figure 4. At some point after T_1 has completed, object A can be

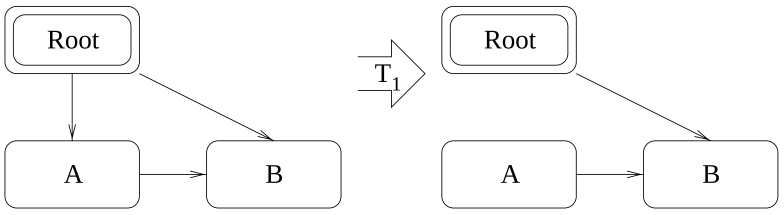

Figure 4: Example 1a: Modified store with transactional finalization

detected as having ceased being reachable from any potential computation and from any persistent root. Subsequently the JVM may run a separate transaction to execute the *finalize* method of object A. This transaction may perform some operations on object B (which is reachable from A). When the *finalize* method returns, the *finalize* transaction attempts to commit its changes[1]. If the transaction commits successfully then object A can be marked as having been *finalized*. As with any transaction, it may fail. If the *finalize* transaction does fail, then the object will remain unfinalized and at some later stage the JVM will again attempt to run that transaction. See section 4.3.2 for a discussion of abnormal *finalize* methods.

4.3 Implications for Persistent Java Systems

The transactional *finalize* model may lead to changes in the structure of the JVM and store interaction, and to the determination of which objects must be maintained in the store. The model requires two garbage collection levels: one at the level of the JVM; the second at the level of the store[2].

JVM and Store Interaction The store level garbage collection mechanism must be aware of and operate according to the various states of the JVM collector (as presented in fig. 2). When the store level garbage collector finds a finalizable object it must inform the JVM to execute a *finalize* transaction thread for the object. This behavior infers two way communication between the JVM and store.

Redefining Reachability It is clear from example 1a (fig. 4), that object A must be retained in the store pending the execution of its *finalize* method. Upon determining the unreachability of A, the store level garbage collector can mark A as *finalizable*, and pass it to the JVM for the execution of the *finalize* transaction.

Only retaining objects that were previously in the store, (even though they are no longer reachable from a persistent root), is clearly not adequate. This is illustrated by example 2, (fig. 5).

The transaction T_2 in figure 5 executes the following events:

- Object C is created and is made to reference object B, (object B is now reachable from object A).
- The reference from the root to A is removed.

[1]The finalize transaction may, of course, be an updating transaction. This should not be surprising, after all, the triggering transaction must have also been an updating transaction.

[2]A persistent Java system could, in principle, be implemented with a single level of garbage collection

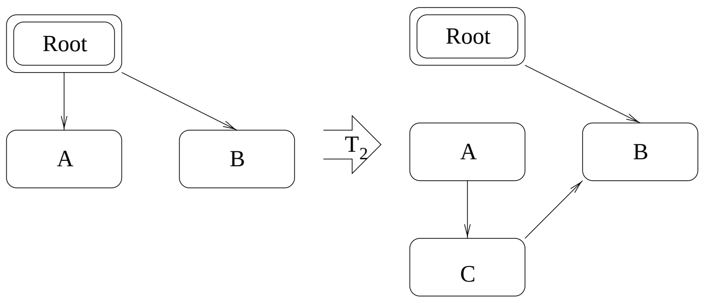

Figure 5: Example 2: Additional Objects

In this example we again deal with the situation where object A is not reachable (from any object or potential computation). It is clear that for object A to be able to execute its *finalize* method then it may need to be able to access object C. Therefore object C must also be held in the store, even though neither C nor A are reachable from the root.

Example 2 illustrates that the set of objects that must be maintained in the store should be extended to include objects that are f-reachable from any existing objects within the store, including any objects that were promoted to the store.

4.3.1 The context for *finalize* methods execution

The store level garbage collector will at some stage determine that object A should execute its *finalize* method. Having detected which object is to have its *finalize* method invoked, the store must inform an appropriate JVM to execute the *finalize* transaction.

Given the extension of persistence by reachability to include f-reachability, the execution of a *finalize* method only ever involves information that is held in the store (see section 4.3). This being the case, the implementer is given the choice of running the *finalize* transactions on the same JVM, a separate JVM or even a dedicated JVM.

4.3.2 Abnormal *finalize* Methods

Although a mechanism for sensibly managing resources is provided, we have no guarantee that it will be used reasonably—a programmer can easily write pathological code. This raises the following questions: Under what conditions should a *finalize* transaction succeed or fail? How do we deal with the abnormal cases?

To determine the conditions under which a *finalize* transaction should succeed or fail, we revise the principal reason for introducing transactions, that is to maintain consistency of the stable image in the face of concurrency. The basis for determining the success of a *finalize* transaction should therefore be only whether a resource conflict exists, not whether the *finalize* code is sensible or otherwise.

An abnormal *finalize* transaction could still cause problems if it is not dealt with in some way. Two examples of abnormal behavior are:

1. Calling **System.exit**.
2. Throwing an Exception.

If the transaction executing the *finalize* caught any possible exception from the *finalize* method being called then this would further limit the damage that could be done. But this is still not adequate. One mechanism introduced to restrict the environment in which programs (or applets), can run is the sand-box. The same mechanism with different restrictions could be introduced to govern the *finalize* transaction execution environment. Within the sand-box the **System.exit** method would terminate the *finalize* method, thus finishing the *finalize* transaction. Other detrimental functions/methods can be dealt with by the same sand-box.

5 Conclusion

This paper has identified a mismatch between the temporal semantics of transactions and the arbitrary timing of Java *finalize* methods. The problem can be dealt with through the use of additional background transactions that execute the finalize methods of objects that have become finalizable with respect to the store. A consequence of the *finalize* transaction is that the notion of reachability so important to orthogonal persistence must be extended to include f-reachability.

Bibliography

ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent java. *SIGMOD Record 25*, 4 (Dec.), 86–75.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4*, 3 (July), 319–402.

DAYNÈS, L., ATKINSON, M. P., AND VALDURIEZ, P. 1997. Customizable concurrency control for Persistent Java. In S. JAJODIA AND L. KERSCHBER Eds., *Advanced Transaction Models and Architectures*, Chapter 7. Kluwer.

GOSLING, J., JOY, B., AND STEEL, G. L., JR. 1996. *The Java Language Specification*. Addision Wesley.

HÄRDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (Dec.), 287–317.