Application Objects in Jikes RVM—Separating the Wheat from the Chaff

Yuval Yarom

16 December 2005

Abstract

Jikes RVM is designed for a shared memory architecture. Adapting the design to a distributed architecture using distributed shared memory involves several decisions, the first of relates to the objects to be shared.

This paper presents the problem, argues that only application objects should be shared, and describes the modification required to be able to recognise application objects.

1 Introduction

The Memory Manager of a runtime system handles the mapping from the abstract object space of the application to the memory architecture. Objects must be mapped to the memory architecture throughout their lifetime, i.e. between the first and the last time the application accesses them. But as the memory is limited, and as the object space may be very large, it is desirable not to keep dead objects mapped.

Most modern memory managers use *Garbage Collection* to determine the lifetime of objects. The garbage collector scans the application's objects graph, and determines which objects are reachable. Reachable objects are those that the application can access either directly (e.g. the local variables of an active function) or by following a pointer from a reachable object. While not all reachable objects are alive, to be alive, an object must be reachable. Garbage (or unreachable) objects, are dead, and can be unmapped safely.

Distributed Shared Memory (DSM) is a memory architecture that hides the complexity of communication in a distributed system by presenting applications with the semantics of a region of memory shared by all the nodes in the system. To provide the sharing, the DSM intercepts the applications' access to the memory, and transfers the data using a dedicated communication protocol—the *Coherency Protocol.*

As the DSM presents the semantics of a shared region of memory, any garbage collection algorithm designed for a shared memory multiprocessor can work on a DSM. Nevertheless, it seems reasonable to assume that the performance of the system can be improved by adapting the garbage collection algorithm to the DSM architecture and by sharing information between the two. Published garbage collection algorithms for DSMs [8, 9, 10] include some level of information exchange between the memory manager and the DSM, however, no study of the extent of sharing required and the expected benefits has been published.

To conduct such a study a flexible DSM system is required. Such a system should be easily configured for multiple DSM architectures and allow easy prototyping of garbage collection algorithms. It is also desired that the system will be able to execute the commonly used garbage collection benchmarks and do that with a reasonably high performance.

IBM's Jikes RVM [1] seems to meet most of the requirements. MMTk [2], Jikes RVM's memory manager, is designed for rapid prototyping of garbage collection algorithms. Jikes RVM has been used for comparative studies of garbage collection algorithms for several memory architectures [3,4,6]. It does not, however, support DSM.

One of the first problems encountered when extending Jikes RVM to provide DSM is how to split the object space between the shared and non-shared regions. The shared region should not contain any application-visible references to objects in the non-shared region. Running the application inside an *Isolate* [5, 7] would have provided an easy way of identifying shared objects. Jikes RVM, however, does not support Isolates, and does not maintain any distinction between shareable and non-shareable objects.

This paper suggests a possible solution to the problem. It argues that only application objects should be shared, and offers a way to modify Jikes RVM to distinguish between application and VM objects.

2 Jikes RVM

Jikes RVM is a Java Virtual Machine written almost entirely in Java with the core Java classes provided by the Gnu Classpath package. The VM is based on a byte code to machine code compiler, which is used to compile both the application code and the VM code. This allows Jikes RVM to execute itself, eliminating the need for a "host" virtual machine that executes Jikes RVM.

Using the same compiler for compiling both the VM code and the application code allows Jikes RVM to remove all the boundaries between the VM and the application. Having no boundaries implies less need for data conversion and higher interoperability. In particular, having no boundaries between the VM and the application allows Jikes RVM to inline VM code in the application's methods, and optimise the VM code for the specific uses by the application. However, having no boundaries between the application code and the VM code means that it may be hard to distinguish between application objects and VM objects.

Some of the operations required for a JVM, such as accessing the host memory architecture or executing compiled code, are not supported by the Java language. These operations are abstracted in Jikes RVM by *magic* classes. When the compiler compiles a method that accesses a magic class, it inserts the code to do the abstracted operation instead of invoking the method of the magic class. Another form of magic supported is compiler directives (pragmas). These are indicating on a per class basis by declaring the class to implement a magic interface, or on a per method basis by declaring the method as throwing a magic exception. Pragmas are used to modify the way the compiler generates the code of the affected methods. Uses of pragmas include control of inlining of methods and to disallow thread switches when executing uninterruptible methods.

Objects in jikes RVM are composed of two sections: a header and a body. The body contains the values of the instance fields of the object. The header contains meta data about the object, including the default hash code for the object, garbage collection information, optional profiling information, and a pointer to the Type Information Block (TIB) of the type.

The TIB holds information related to all the objects of the type. This information comprises a pointer to an object representing the type, a virtual methods table that points to the compiled code of the instance methods of the class, and information used to facilitate invocation of interface methods and dynamic type checking.

Pointers to static methods and members are stored in a global data structure—the Jikes RVM Table Of Contents (JTOC). The JTOC also stores references to the TIBs of the loaded types.

3 Splitting the Object Space

In order to extend the Jikes RVM to provide a DSM architecture, it must firstly be decided which objects are to be shared (reside in the shared region) and which are local.

Two requirements guide this decision. The first is that, as the memory manager and the DSM should share information, the DSM should be implemented inside the Jikes RVM. It should be noted that if the DSM is implemented inside the VM, some objects must be local to the node. These objects include, for example, the implementation of the coherency protocol and objects accessed by uninterruptible methods.

The second requirement is that references to a local object are only followed on the node in which the local object resides. As the VM has little control over the application, this requirement implies that all the objects used by the application should be shared. This includes objects of the application specific classes and objects belonging to core Java classes that are created by the application.

Two strategies can be used by the VM to avoid accessing local objects from remote nodes. The first can be used when the system guarantees that equivalent objects must exist on every node. (For example, the TIBs of each type must exist on every node, and all the TIBs of a given class are equivalent.) In such a case, pointers to the object can be converted between external and internal representation (swizzled) when the pointers are exported or imported. This ensures that at each node the pointer references the local representation of the object. For other objects, remote procedure calls can be used for accessing the objects.

Sharing the application objects while keeping the VM objects local, meets both requirements.

While Jikes RVM does not distinguish between the application objects and the VM objects, it does maintain partial distinction between some application classes and VM classes. Application classes are loaded by the application classloader, whereas VM classes are loaded by the bootstrap classloader. However, this distinction is not sufficient to meet the second requirement because core classes are also loaded by the bootstrap classloader and some objects (e.g. the hash table used to implement `String.intern()`) are used by both the VM code and the application code.

To meet the second requirement, another classloader—the VM classloader—is added to Jikes RVM. The VM classloader is used to load the VM classes and the core classes for the VM use, while the bootstrap classloader is used for loading core classes for the application. As Java classes are uniquely identified by the class name and the defining classloader, the use of a VM classloader, which is not accessible by the application, maintains a complete separation between application classes and VM classes and precludes the creation of references from application objects to VM objects.

A complete separation does have its drawbacks. The application does need a way to communicate with the VM. To allow such communication, VM specific classes are always loaded by the VM classloader. The application code can access these VM specific classes, but is limited to using only primitive types and VM specific classes. Core classes cannot be used. The glue layer between Classpath and the VM is now being rewritten to use only this limited interface.

The separation described above handles sharing of objects. It does not, however, allow sharing of static members of the application classes. To support such sharing, the JTOC will have to be divided into a shared region containing the static fields of the application classes, and a local region containing all the information related to the VM classes as well as the TIB pointers and the pointers to the code of static methods of the application classes.

4 Conclusion

This paper presented some of the effort that is required for providing DSM support in Jikes RVM. While the work is far from being complete it seems that the major problems have been addressed, and that there will be no significant impediments to the implementation.

References

[1] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb. 2000.

[2] S. M. Blackburn, P. Cheng, and K. S. McKinley. A garbage collection design and bakeoff in JMTk: An efficient extensible java memory management toolkit. Technical Report TR-CS-03-02, The Australian National University, Sept. 2003.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[4] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *International Symposium on Memory Management*, Berlin, Germany, June 2002.

[5] G. Czajkowski. Application isolation in the Java Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 354–366, Minneapolis, Minnesota, United States, Oct. 2000.

[6] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM Press.

[7] Java Community Process. *JSR 121: Application Isolation API Specification*, Dec. 2005.

[8] D. S. Munro, K. E. Falkner, M. C. Lowry, and F. A. Vaughn. Mosaic: A non-intrusive complete garbage collector for DSM systems. In *Proceedings of the First IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pages 539–546, Brisbane, May 2001. IEEE Computer Society.

[9] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Distributed shared memory management for Java. In *Proceedings of the Sixth Annual Conference of the Advanced School for Computing and Imaging (ASCI 2000)*, pages 256–264, Lommel, Belgium, June 2000.

[10] W. Yu and A. Cox. Conservative garbage collection on distributed shared memory systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 402–410, Hong Kong, May 1996.