Resource Use in the Interaction of Managed and Unmanaged Code

Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth
Intel Corporation, Microprocessor Research Lab
michal.cierniak@intel.com

Abstract

Mobile and resource aware systems are often implemented using managed runtime environments (MREs) because most of the code is managed by a virtual machine. The virtual machine can keep track of and control the use of resources, which provides a good basis for resource control mechanisms. However, some code is not managed, in particular, the native code underlying native methods.

We discuss the transition from managed to native code in our MRE implementation. In particular, we discuss two issues in carrying out these transitions: implementing all the details for the transfer, and passing reference parameters. Also discussed are two approaches to each issue having different performance and resource requirements. First, we discuss the implementation of the transition from managed to native code using customized stubs versus using a generic *stub interpreter*, and show that there is a memory versus performance tradeoff for these two approaches. Second, we describe two ways of implementing JNI object handles and show that a slightly more complex approach is much better than a simple one in terms of both memory usage and performance. These demonstrate the importance of choosing the right mechanisms for managed to native transitions in both resource limited and high performance virtual machines.

1. Introduction

Mobile systems are often implemented in managed runtime environments like the JVM [3] or CLI [1]. One advantage of using an MRE is that all managed code is executed under the control of a virtual machine (VM). The VM interprets bytecodes, or generates code using a just-in-time (JIT) compiler, and that allows monitoring of all actions that may put a strain on system resources, such as memory or threads. Many techniques for enforcing resource management policies have been developed for MREs.

Since not all code in an MRE application is managed, MREs include a mechanism for invoking native code. In the JVM, this mechanism is the Java Native Interface (JNI, [4]); in CLI, it is the Platform Invoke mechanism (PInvoke, [5]). In general, resource use by native code is not under the control of the VM, and native code can use unbounded resources or even crash the MRE by misusing resources. However, the interface between managed and native code is under the control of the VM. In this paper, we explore resource issues in this interface. In particular, we study the resource use of crossing from the JVM through JNI to native code in our MRE, the Open Runtime Platform (ORP, [2]).

ORP is an MRE that implements both the JVM and CLI specifications. ORP's goal is to implement features that are (or will become) common in the marketplace, to assist CPU designers in evaluating future processor architectures. Performance is a key goal; in addition, we are studying other aspects such as security, reliability, and resource control. Resource control is particularly important to embedded and mobile devices such as PDAs, cell phones, and mobile computers, where resources such as power and memory are severely limited compared to desktops and servers. However, careful management of resources can also benefit high-performance systems.

This paper is preliminary – in particular, we do not describe a resource control mechanism for JNI. Instead, we discuss two implementation issues having resource implications. We then describe two strategies for addressing each issue, and measure their impact on memory usage and performance.

The first issue concerns how to transfer control between managed and native code. For example, when calling managed code on IA32, arguments are pushed in left-to-right order and objects are passed using direct references, while calling JNI native methods requires that arguments be pushed right-to-left and

objects be passed using *handles* (indirect references). There are two principal schemes for implementing this. One is to use a machine code *stub* or *wrapper* that is called by the MRE to translate between the calling conventions and object representations and then call the native code. Because these stubs can be highly customized for each method, their performance is good and this has been the scheme used by ORP in the past. The other scheme makes use of a generic *stub interpreter*, which trades performance for memory use but provides a useful alternative in some circumstances. These two schemes are described in Section 2.

The other issue concerns how to implement JNI object handles. These are indirect object references that allow the VM to provide garbage collection (GC) safety for native code. The simplest implementation has one structure per handle with all handles linked into a list. Another implementation aggregates handles together reducing the overhead per handle. These tradeoffs are discussed in Section 3.

Stub interpreters have benefits beyond those described in this paper, including the ability to impose resource control. As an example, in the case of JNI object handles, the stub interpreter decreased the amount of time required to implement the aggregate handles and try new allocation heuristics. We speculate on this possibility and future work at in Section 4.

1.1. Note on Performance Comparisons

In this paper, we present measurements and performance comparisons of our different approaches. These measurements were taken on a system with four 2 GHz Xeon processors and 4GB of RAM running Windows 2000 Advanced Server. The measurements are of SPEC benchmarks, but due to limitations in our system we do not follow the run rules specified for SPECjvm98 [1] and SPECjbb2000 [2] and no comparison with official scores for these benchmarks can be drawn from the results presented in this paper. We used a heap size of 96MB for SPECjvm98 and 1GB for SPECjbb2000. ORP is a well-tuned MRE implementation with performance competitive with commercial products [2], so we believe that the results are applicable to other implementations.

2. Stubs and Stub Interpreters

A JNI stub must convert between the conventions for managed code and those for JNI. For ORP on IA32, this means pushing a managed to native frame (a structure used by the VM to enumerate all managed frames on the stack), reversing the argument order, converting object references into handles, enabling GC, synchronizing (for synchronized methods), converting the return value from a handle to an object reference if the return type is a reference type, and throwing any exception raised by the native code to the managed code. Exactly how much and what kind of work is required depends upon the method being calling, and in particular how many arguments it takes, which are references, whether it returns a reference, and whether it is synchronized.

One approach is to construct a customized sequence of machine instructions, a stub, for each JNI native method. This sequence can be constructed at load time – any time between when the class is loaded and the native method is first invoked. Since the generated sequence does precisely the work needed for that method it potentially has excellent performance. However, custom stubs have to be generated for each native method potentially leading to high memory usage. Another disadvantage of custom stubs is the need to generate machine code from within the VM, customizing and piecing together various code templates. This generation is a complex process, and is error prone in our experience.

An alternative is to generate a very short machine code sequence that calls a generic function we term a "stub interpreter". The short sequence passes a data structure describing the method in question to the generic stub. The generic stub pushes the managed to native frame (this cannot be done in C), and then calls a C function passing the method's data structure. The C function then uses the method data structure to guide its actions in converting the arguments, synchronization, and converting the result. To actually

[1] We followed the guidelines for research use as outlined in Section 4.1 of the SPECjvm98 run rules: http://www.spec.org/jvm98/rules/runrules-20000427.html#Research.

[2] We followed the guidelines for research use as outlined in Section 5 of the SPECjbb run rules: http://www.spec.org/jbb2000/docs/runrules.html#Research%20Use.

call the native method it needs to call one more generic stub to copy the converted arguments onto the stack in the right locations. In this approach, each native method requires a stub of only two instructions; the rest is done by generic code. Thus memory requirements are modest. Additionally these two-instruction stubs are easy to generate, while the generic stubs are short and fixed. Hence this approach is easier to debug and significantly easier to port to a new platform. However, because many decisions are made each time the native method is invoked rather than at custom stub generation time, performance might suffer.

As an aside, ORP uses stubs for many other managed-to-native code transitions and similar tradeoffs exist for some of these other uses. For example, CLI delegates require stubs to create and invoke the delegate. A delegate is essentially a type-safe method pointer that can encapsulate either a static or instance method. They are used, e.g., to implement CLI threads. The stubs created for a delegate do the actual invocation of the delegate's method on the object that the delegate represents (if the underlying method is an instance method). CLI also has multidimensional arrays, each of which have a get and put method that depends upon the number of dimensions and array element type. Custom stubs could be generated for delegates and the array access methods to get high performance, or an interpreter could be used to reduce memory usage.

Stub interpreters also offer greater implementation flexibility and simplify prototyping. It is straightforward to add new functionality to a stub interpreter while generating the equivalent machine code is difficult and error-prone.

ORP implements both custom stubs and a stub interpreter for JNI methods, selected by a command-line option. We measured the memory usage and execution time of the two approaches and observed a trade off between memory usage and the execution time performance.

Table 1 shows the amount of memory allocated (in bytes) for JNI stubs with the two approaches. (Aggregate handles are described in Section 3.) For custom stubs, we show the amount of memory for each JNI handle scheme described in the next section. These results show that custom stubs require an order of magnitude more memory than the interpreter for JNI stubs. Also, there is little difference between the different JNI handle schemes.

	Custom Stub, Simple Handles	Custom Stub, Aggregate Handles	Stub Interpreter
Compress	5,703	5,282	432
Jess	5,943	5,500	444
Db	5,703	5,282	432
Mpegaudio	5,918	5,493	444
Mtrt	6,451	5,997	489
Jack	5,868	5,443	447
Javac	6,704	6,227	507
SPECjbb2000	10,875	10,119	819

Table 1: Bytes allocated for JNI stubs

Figure 1 shows the performance of the two approaches normalized to the performance of custom stubs. Higher is better. In five of the benchmarks, the stub interpreter shows a performance degradation and in three of the benchmarks the degradation is at least a factor of three. Custom stubs clearly have better performance in general.

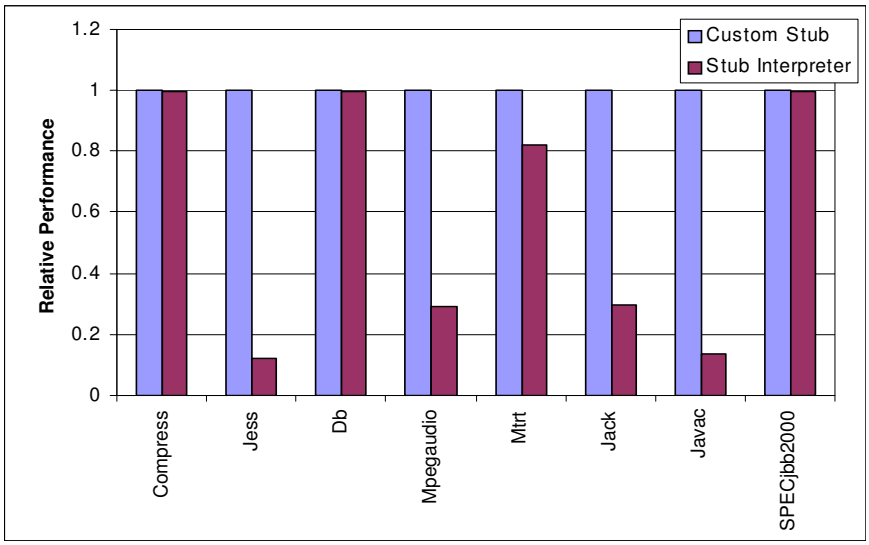

Figure 1: Performance Comparison of Custom Stubs versus Stub Interpreter

3. JNI Object Handles

Objects are passed to JNI native methods as handles. These handles are opaque to native code, which cannot directly manipulate objects. Instead, the VM provides functions for such operations as accessing fields and invoking methods. Because native methods must go through the VM to get all object handles, the VM can use handles to provide GC safety to native code. The handles insulate native code from concerns about garbage collection and whether objects are moved. The VM keeps a list of all handles it passes to native code, and at GC time enumerates all handles as roots. When a native method completes, all handles given to it are freed. (Native methods can also keep objects across invocations using global handles, but we do not describe this here.)

The simplest scheme for implementing handles, and the one ORP used initially, is to place each handle in a structure of its own and to link these structures into a list. In this scheme, ORP maintains a doubly linked list of these handles. Each handle requires sixteen bytes on IA32: four each for the object reference, two list pointers, and a flag to indicate whether the handle was allocated on the stack or heap. At entry to a native method, all reference arguments are converted into handles that are allocated on the stack. Any additional handles needed during the method's execution to, say, access a field, are allocated using malloc.

Another scheme, which ORP also implements, is to aggregate multiple handles into the nodes of a singly linked list. In this scheme, each node in the list has an array of object references, variables to indicate the maximum capacity and current size of the array, and a pointer to the next node. The current size reflects the number of references currently being used as handles. Any excess capacity over the current size is used to allocate new handles, but once the capacity is exhausted, a new list node must be allocated. On entry to a native method, ORP allocates a single list node on the stack with capacity eight plus the number of reference arguments. Thus up to eight object handles can be allocated during native method execution before a new list node must be created. Once the original eight are used, a node is allocated in the heap with capacity ten. These heap nodes have an overhead of eight bytes for a total of 48 bytes for ten handles, or an average of just fewer than five bytes per handle as compared with sixteen in the simple scheme. Furthermore, the number of calls to malloc and free is reduced in the aggregate scheme, the handles have better locality, and native methods requiring fewer than eight handles need no mallocs or frees at all. Thus the aggregate method should have better memory usage and probably better performance.

Table 2 shows the total number of handles and total memory size of the handles for the two schemes. These results show that the aggregate scheme uses much less memory. The reduction is dramatic for both Javac and SPECjbb2000.

	Number Handles, Aggregate Scheme	Handle Bytes, Aggregate Scheme	Number Handles, Simple Scheme	Handle Bytes, Simple Scheme
Compress	20	960	493	7,888
Jess	23	1,104	830	13,280
Db	21	1,008	370	5,920
Mpegaudio	20	960	349	5,584
Mtrt	20	960	26,271	420,336
Jack	71	3,408	972	15,552
Javac	20	960	784,638	12,554,208
SPECjbb2000	7,361	353,328	1,473,371	23,573,936

Table 2: Number and size of handles allocated on heap

Figure 2 shows the performance of the two schemes, normalized to the execution time of the simple scheme and factoring out the compilation and class loading time. As with Figure 1, higher is better. Aggregate handles improve performance on most of the benchmarks, and by as much as 11% for javac. It wasn't obvious that the new aggregate scheme world work better since the old scheme had never itself shown up as a performance problem in our tuning work.

Neither the simple nor the aggregate scheme frees handles while the native method is executing. Future work, involving a mechanism for deallocating these handles, might further reduce the memory footprints of both schemes, although such a mechanism will not decrease the total memory allocated to the handles.

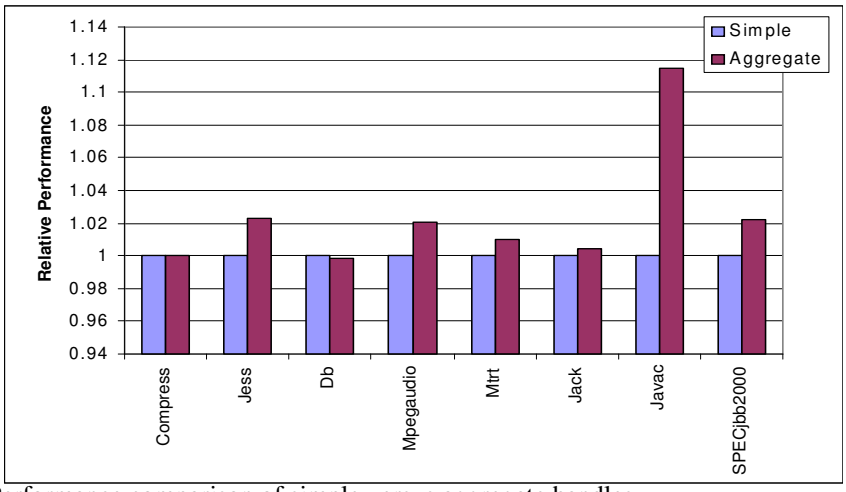

Figure 2: Performance comparison of simple versus aggregate handles

4. Conclusion and Future Work

The managed to native interface is important both in terms of resource usage and performance. This paper has discussed two implementation issues in the managed-to-native interface and presented two approaches for each issue. For JNI handles, one approach—the aggregate handle scheme—is better in terms of both memory usage and execution speed. For stubs, there is a tradeoff between the memory required and performance achieved for the two approaches. ORP does not do this today, but it could be modified to dynamically select between the different stub approaches depending on the amount of memory available and the need for greater performance.

ORP uses stubs for many other purposes, and some of these are likely to have similar resource considerations. We conjecture that these stubs could be used to implement a resource control mechanism such as limiting the number of handles allocated to native code. Stub interpreters would simplify prototyping this in a future version of ORP. We envisage controlling other resources as well, such as the

stack space and maximum execution time that may be used by native code, as the managed-to-native transition stubs delimit the start of the stack space and the start of execution. Timers and guard pages could detect when the resources limits are reached. In this way, some control over native code could be achieved. All of this is future work.

References

[1] Common Language Infrastructure, ISO/IEC 23271. ISO/IEC, 2002.

[2] Cierniak, M., Eng, M., Glew, N., Lewis, B.T. and Stichnoth, J.M., The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment, *Intel Technology Journal*, 7 (2003).

[3] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley, 1999.

[4] Liang, S. *The Java Native Interface.* Addison-Wesley, June 1999.

[5] Platform Invoke Tutorial, C# Programmer's Reference, *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcwlkplatforminvoketutorial.asp*