White Paper

Best Practices for Native Code Integration Between C or C++ and Java™

May 19, 2009

Best Practices for Native Code Integration Between C or C++ and Java

This white paper describes recommended techniques for integrating native code, typically written in C or C++, with Java code. While generally applicable to any Virtual Machine, examples presented relate to the PERC Ultra virtual machine.

Introduction

Atego® optimizes the PERC Ultra Java Virtual Machine to accelerate the programming of demanding embedded applications, especially in the fields of network infrastructure, industrial automation and telematics. The majority of any of these applications can be developed in the Java® programming language, by using PERC Ultra. Despite this, it is common to also have code written in other programming languages, usually C or C++. PERC Ultra provides several ways to access and interface to this so-called "native code". Note that PERC Ultra compiles Java down to the same kinds of native machine instructions that C or C++ would, and, in this general sense, is just as "native" as C or C++. Conventional usage, however, distinguishes between machine code compiled by a Java compiler, and "native code", compiled by non-Java compilers. This white paper outlines best practices for integrating this native code, and especially with the code generated by the PERC Ultra virtual machine.

JNI

JNI, the Java Native Interface, is the most widely used and understood mechanism for integrating Java code with native code. Source code written with JNI is portable across different VMs that support JNI. Even JNI binaries for a given CPU/RTOS combination are interoperable with any other Java VM supporting JNI for that platform.

JNI, in general, is well documented in the Addison-Wesley book, The Java Native Interface, by Sheng Liang. This book does not cover the details of compiling

and running native code on an embedded system. The PERC Ultra User Manual provides these details, giving the correct (and obscure!) compiler and linker flags and lists example Makefiles, to use JNI on embedded targets. Both of these resources are must-reads before designing or implementing JNI code.

Generally, JNI is the first and best design choice for integration with native code. JNI provides a rich interface between native and Java code, and allows the programmer to implement everything in native code that could be done in Java code. This richness, though, opens up design pitfalls for the unaware — just because something can be done, doesn't mean that it should.

Prefer Java implementations over native ones whenever possible

This is the overarching design rule when using JNI. As with any rule, there are exceptions, but when there is a choice to implement a feature in Java or native code, the former is almost always the better design. Even if it takes awkward measures, the guideline usually holds. For example, if a native method needs synchronization, it might seem natural to use the JNI functions MonitorEnter and MonitorExit, to implement a mutual exclusion. These functions can lead to subtle bugs, though. If an exception is thrown through MonitorExit, the C code won't automatically release the monitor, which can lead to difficult-to-debug livelock and deadlock conditions. Not matching MonitorExit's to MonitorEnter's causes the same problem, and is equally difficult to debug. A clean solution to this is to hoist the monitor up and out of the native code, into the calling Java code, and synchronize the whole of the native method call.

Another example concerns error handling. While it may seem natural to allocate, construct and throw Java exceptions from native code, this is error prone, and difficult to maintain. It is almost always better to have the native code just return an error status, as is the usual practice for C code. The calling Java code can then check that error status, and create and throw any exceptions as needed.

Treat the functions AttachCurrentThread and DetachCurrentThread similarly. These two functions are responsible for more confusion and bugs in deployed JNI code than any other. They should only be used when absolutely necessary. Do not write platform dependent native code to create and manage threads, and then invoke these evil twins to rendezvous with Java code. Instead, it is much cleaner and safer to create a new thread in Java, as a subclass of java.lang.Thread, and enter native code from there. Overall, there is much less native code. If the code needs enhancement later, say, by

adding a thread pooling mechanism, it will be much easier to maintain and understand.

Chapter 10 of The Java Native Interface describes a number of other pitfalls to avoid when writing JNI code. One more that is worth mentioning here has to do with references to Java objects. Automatic garbage collection of unused objects is one of the great strengths of Java, eliminating the risk of both dangling pointers and memory leaks. Both risks reappear when writing native code. When a JNI function acquires a reference to a Java object, the Java Virtual Machine has no way to know that the function is done with the object until the function returns. A long-running function can thus keep an object alive indefinitely; the PushLocalFrame and PopLocalFrame functions can be used to advise the VM that a set of references is no longer needed. Most object references used in JNI functions are “local” to the lifetime of the function call that acquires them; any local reference can be converted to a “global” reference that will remail valid until it is explicitly released. In order to share a reference between calls to the same or different functions, a global reference must be acquired; otherwise the effect is similar to using a pointer that is no longer valid. However, forgetting to eventually delete a global reference will prevent the object from being garbage collected even after the JNI function returns.

JNI DESIGN STRATEGIES

The JNI book warns about the different semantics a native programmer sees when using a VM that uses native threads, as opposed to a “green threads” VM which performs its own management of Java Threads within a single OS-level thread. The PERC Ultra VM uses a hybrid approach, which combines the best features of both models. This strategy allows the JNI programmer to issue calls that might block naturally, without taking any special actions to prevent the other threads from blocking. That is, a JNI call in PERC Ultra may call a blocking I/O function, without linking to any special libraries. If the call blocks, any other runnable threads will run until the I/O returns.

The Liang book suggests three strategies for designing JNI based libraries: One-to-one mapping, shared-stubs, and peers. The first, simply wrapping all C calls one-to-one with Java calls, works very well in most cases. For small interfaces, this is the preferred approach. The second method suggested, shared-stubs, involves writing one assembly language routine which dispatches to a given C routine based on a name lookup. This is an abomination, and should be avoided at all costs. In addition to being extremely non-portable, and difficult to maintain, it introduces new security holes and creates problems that are very tricky to debug. The final approach, using peers, is the way that much of java.awt.* and java.io.* is implemented. It is probably best for larger frameworks, but leads to bad object coupling, and is hard to debug and unit-test.

The JNI design strategy that Atego recommends is most famously implemented

in the SWT windowing toolkit, and documented at the link in the references section. This design philosophy has two main tenets: First, to minimize the amount of native code, as described above. The second is to retain the portability benefits of Java. Developers of Java native code frequently forget this tenet, which leads to great loss. The benefits of Java's portability, especially for an embedded system, are legion. To develop, debug, and unit test on a desktop machine, especially if target hardware is unavailable or over utilized is a tremendous advantage. All too frequently, developers throw away this advantage the moment the first line of native code is written.

> **It is OK (even desirable) to write platform-dependent code in Java, as long as there is an implementation for each platform.**

This maxim is the result of the merger of the above two design tenets. Using this design strategy is straightforward. First, design a clean, pure-Java interface to the needed functionality, using standard OOA&D techniques. Second, discover the minimum set of C or C++ routines that are needed to implement this interface for each of the target platforms. Note that for some platforms, especially simulators, no C code may be needed at all. Wrap each of these C/C++ functions in a simple Java wrapper method, which just calls the function, passes the arguments it requires and returns the return value. Finally, implement in pure Java code the Object Oriented design from the first step in terms of the primitives defined in the second step, for each platform. Use the usual Gang-Of-Four patterns to link the top two layers—typically the factory pattern is the way to go. Careful attention to this approach has many benefits. The greatest of these is portability, for the value of running an embedded application on a desktop simulator cannot be overestimated. Most embedded projects face significant delays in the development of software because of this lack of portability. Ensuring that the bulk of an embedded project's Java code can run on a desktop machine is the single most important thing a team can do to keep its project on schedule and under budget. This also enables the continuous unit testing of the embedded software, running on a desktop host. There are many Java tools to help with unit testing; Atego recommends the JUnit tool.

If an embedded project uses this recommended approach to integrating native code, JUnit can still be used to unit and regression test the bulk of the code. Just as the factory pattern creates platform-specific instances of the native interface, it can also create debug-specific, or testing-specific interfaces. Again, this is an easy way to ensure good code coverage for the platform-independent code without needing to run on a target. Another huge benefit is debugging. Often when there is a bug, the Java and C teams play finger-pointing games, casting blame

at each other. When the C interfaces are small and well defined, it is trivial to write all C, or all Java reproducer cases, which clearly illuminate the problem.

A REAL-WORLD EXAMPLE

A real example best illustrates this design strategy. A large industrial facility has embedded computers distributed widely throughout the plant. A proprietary network connects these computers, sending timing signals, to precisely synchronize the collection of data from each of the computers. The computers run both VxWorks® from Wind River, and embedded Linux®. However, the software interface to the timing network is very different for these two operating systems. On VxWorks, the device driver for the network interface signals a semaphore when a timing event arrives. On Linux, there is a device special file; reading this file blocks until the network receives the timing event, then one byte is readable from the file. As there is no built-in support in Java for this proprietary network, native code is needed to interface the Java code to this network.

To simplify this example, there will be only one timing signal, a start signal, which indicates that the computer should start monitoring. Following our design guidelines, the first step is to create a Java interface for this. This will need just one method:

```
public interface TimingNetwork {
        public static void waitForStartSignal() throws TimingNetworkException;
}
```

Note this is an interface, not a native code entry point. The naïve approach would be to make this a concrete, native method, and to provide multiple native implementations, one for each platform. But that leads to a proliferation of native code, which violates the first tenet stated above.

The second step is to implement the smallest set of native code that the above interface requires, once per platform. For VxWorks, this just boils down to waiting on a semaphore. A reasonable implementation of this second step for VxWorks might look like:

```
public class VxWorks {
        public static native int semTake(int semId, int timeout);
        public static native int semGive(int semID);
}
```

Note that the two methods have the same name as the VxWorks functions they wrap. This underlines that these are not idealized, portable semaphores; rather, they are exactly VxWorks semaphores, with all implied benefits and constraints. For Linux, the minimal set of native code required to implement this interface is more interesting: it is empty! The Linux implementation of TimingNetwork can be finished with pure Java code, and no additional native methods.

The final step is to implement the interface in terms of the primitives just defined. For VxWorks, this implementation gets the id of the semaphore from a configuration setting (perhaps from a file), and calls into the SemTake method. The Linux implementation is pure Java, and just does a blocking read on the device special file the device driver uses.

The most important implementation is the portable implementation, which will run on any platform, even if it doesn't have the special networking hardware. The code for this could just be a no-op. Or, a more sophisticated implementation could simulate the timing network over TCP. Granted, the timing would not be correct for the simulation, but it allows for the code to run on any Java VM on any hardware. This portability, which aids design and development is vitally important to retaining the full benefits of the Java programming environment.

JNI PERFORMANCE

JNI has a reputation for being slow. And indeed, a call to a C function from Java through JNI takes between fifteen and thirty times as long, depending on the number and type of arguments, as a call from Java to Java (assuming AOT or JIT compilation). Keep in mind though, the famous quote from Donald Knuth, "Premature optimization is the root of all evil"[1]. Even though a call to JNI is much slower than a non-native call, it still runs very quickly, in absolute terms. Even a slower PowerPC processor can execute millions of JNI calls per second. Keep this number in mind when designing code that uses JNI — JNI is slower than non-native code, but there is almost always plenty of speed available to accomplish the task at hand. A common mistake is to fear the slow speed of JNI, and break the above design rules, in an attempt to gain more performance. The most common design rule broken is the "leaf function" rule, for many developers will think that, "JNI calls are slow, so I best get as much work done possible per call". Before breaking these design guidelines, first calculate back-of-the-envelope estimation of how many JNI calls the code needs to execute in a given timeframe. Measure the speed of a JNI call on the target platform. In most cases, these measurements reveal that the software design can afford the discipline of clean, maintainable code, as described above. In fact, this is true for most code, not just JNI, for otherwise, we'd be writing all

1. Knuth, Donald: Structured Programming with Goto Statements. Computing Surveys 6:4 (1974), 261–301.

of our code in assembly language.

There are a number of techniques one can use that can improve JNI performance without sacrificing clean design. A common pattern in JNI code is to copy a buffer from a C function into a Java-accessible array or data structure. It is straightforward to allocate this buffer in the C code called from JNI, as often, that is the only place that has the correct size of the buffer. However, when allocating memory from JNI code, the JNI routines must synchronize with the garbage collector and the other running Java threads, which can be expensive. It is usually faster (and more maintainable) to allocate the memory in the Java world, then pass the buffer into the C code. This is true, even if it means preallocating a maximal sized buffer. Optimizing this example one step more takes into consideration the common need to get an internal, "C" pointer out of the allocated Java array or Object. Usually, JNI code will need to call Get<Type>ArrayElements, or similar function to extract a real "C" pointer from the Object. The implementation of Get<Type>Array elements (and similar functions) in the PERC Ultra VM requires the garbage collector to make the object static temporarily. This is so that the object won't get moved by the garbage collector, for this would invalidate the pointer. If the object or array is allocated in the Java side, it can be made static at allocation time by using the AllocateStatic type tag. This will eliminate both the synchronization overhead and any additional copy. Also, if only one buffer is allocated, and reused over the course of many JNI calls, the work of the garbage collector will be reduced, which will result in an added speedup.

A FINAL NOTE ABOUT JNI OPTIMIZATION AND SPEEDUPS:

As with any programming optimization or tuning situation, the first task should be to run a profiler, to verify where the slowdowns are. Time and time again, slowdowns happen in the most unexpected places. Remember—a few days spent profiling a program can often save months of optimizing the wrong parts of it. Atego PERC Ultra integrates with any profiling tool that uses the JVMPI protocol, including the Eclipse Workbench Test and Performance Tools Platform (TPTP) which provides a very dynamic view of memory and CPU usage by the Java application. Unfortunately, while very effective in a desktop setting, the memory requirements for this type of profiling are unacceptably high for many embedded targets.

Atego offers two ways to overcome this limitation. By using a "proxy" profiling agent, the memory load can be placed on the host where the profiling tool is running instead of on the target system. However, this can turn a memory problem into a bandwidth problem since all the data must pass between target and host over the network. Another alternative is the open-source batch-mode HPROF profiler published by Sun

Microsystems and supported by Atego. Its lighter-weight approach provides less detail than interactive profiles, but is usable in most embedded environments.

The pitfall with any profiling tool for Java is that every native method is opaque. That is, once control enters a JNI method, what happens inside the method is invisible to the profiler, until control returns or re-enters Java. This is another good reason to move as much code as possible from JNI up into Java. If there is a small amount of JNI code, and a need to get detailed timing information for short snippets of code, there is an easy answer. Most modern CPUs, including the x86 and PowerPC have a high resolution clock on chip, which is very cheap to read. On the x86, this is at the clock rate, while on the PowerPC, it usually a small multiple of the clock rate. Atego Professional Services can provide examples of inline assemble code, which can read these registers. By scattering these calls throughout the JNI code, it is easy to get very precise timing measurements.

PNI

PNI, the PERC Native Interface, is the native interface of last resort, and comes into play when JNI just isn't fast enough. PNI runs faster than JNI, but requires great care to use it safely.

PNI gains its speed from being tightly coupled to the PERC Ultra VM. While JNI is portable across VMs, and across different versions of the same VM, PNI has changed several times as the internals of the PERC Ultra VM evolve, and will probably continue to do so. JNI cleanly handles most of the underlying VM machinery itself, but PNI does not. For example, with JNI, the programmer does not need to worry about whether a native call will block the calling thread. This is not the case for PNI — if a method called from PNI might block, it must wrap that call with certain PNI functions. These ensure that the whole virtual machine does not get stuck behind the one blocking call. JNI also shields programmers from some details of interacting with Java code, such as the need to yield the CPU periodically in order for high-priority tasks to meet their latency requirements, and the possibility that objects may be relocated by the defragmenting garbage collector. The PNI programmer is responsible for taking these possibilities into account. The PERC Ultra User Manual contains the complete reference and guidelines for using PNI. Be sure to be thoroughly familiar with this reference before writing PNI code. The Atego Professional Services team can also assist with formal inspection and review of customer-written PNI code.

PNI's one benefit is that it does run faster than JNI. How much faster depends, of course

on the platform and the usage. Typically, PNI will run about three times faster than JNI, again depending on the number and type of arguments. One can mix and match PNI code and JNI code, so a common use is to first design and write all native methods in JNI, and later, in the tuning and optimization phases of a project, rewrite those few methods which absolutely need to in PNI.

Direct Memory

The most common need for native access is to read or write to memory outside the Java heap. This external memory may be a device, memory mapped into the address space, memory shared with C threads, or the like. PERC Ultra provides a clean, optimal solution to this common problem with support for DirectMemory. A single class, named COM.newmonics.pvm.DirectMemory[1], accesses this type of memory. DirectMemory has straightforward static methods for reading and writing fundamental types from a given raw memory address. These methods look like normal Java function calls (and are so if called in interpreted mode), but they are handled specially by both the Atego AOT and JIT compilers. Because the compiler understands what these functions do, instead of generating instructions to call the methods, it can generate in-line instructions to access the memory directly, optionally omitting array-bounds checking, and performing processors-pecific optimizations. With these optimizations, the compiler can emit machine instructions similar to what a C compiler would emit. This results in much faster execution than a JNI call, or even a Java-to-Java call, and is by far the fastest way to read and write external memory with the PERC Ultra virtual machine. Even so, it is very easy to maintain this code, as it is trivial for the code maintainer to see the intent of the original programmer. Array-bounds checks can be enabled during the testing and debugging stage to detect errors, and turned off for production use. If embedded code just needs to read or write foreign memory, Atego strongly recommends the use of the DirectMemory class.

Standard Java provides a related facility in the form of “direct byte buffers.” A direct byte buffer can be created by either the java.nio.ByteBuffer.allocateDirec method (which gives no control over the address of the buffer) or with the JNI NewDirectByteBuffer function (which does allow the address to be specified). Like DirectMemory, direct ByteBuffers provide access to arbitrary memory addresses from Java code, but the compiler optimizations described above apply only to DirectMemory. Direct ByteBuffers are accessed through ordinary native method calls, with the associated performance penalty.

1. The naming is historical. Atego acquired the PERC Ultra Virtual Machine technology from NewMonics, Inc. in 2005.

Conclusion

There are many ways to integrate native code with PERC Ultra. The first choice for programmers needing this integration should be the DirectMemory class, if applicable. If not, using JNI, with the design pattern outlined above is the recommended approach. Finally, if after profiling, JNI is too slow, consider using PNI for the most time-critical loops. In any case, the Atego Professional Services team has many years of invaluable experience working with native code, and can help projects meet their complex requirements in this area of Java Programming.

References

The Java Native Interface, by Sheng Liang, ISBN 0-201-32577-2

The PERC Ultra User Manual (PDF file, ships with PERC Ultra; also available for download)

SWT: The Standard Widget Toolkit PART 1: Implementation Strategy for Java™ Natives (available at http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html)

To obtain more information, please contact Atego at **www.atego.com** or call one of our sales offices

North America
Phone: (888) 91-ATEGO
Fax: (858) 824-0212
E-mail: info@atego.com

France
Phone: +33 (0) 1 4146-1999
Fax: +33 (0) 1 4146-1990
E-mail: info@atego.com

United Kingdom
Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 575033
E-mail: info@atego.com

Germany
Phone: +49 7243 5318-0
Fax: +49 7243 5318-78
E-mail: info@atego.com