Using Inferno ™ to Execute Java ™ on Small Devices

C. F. Yurkoski
L. R. Rau
B. K. Ellis
Bell Labs
600 Mountain Avenue
Murray Hill, NJ
{yurkoski, larryr, brucee}@bell-labs.com
http://www.chunder.com/

Abstract. This paper describes an implementation of Java [1] on the Inferno operating system.

There are applications for which object oriented designs are the appropriate solution. Java is an object oriented programming language in which these solutions can be written. But as always there are tradeoffs, and O-O designs in Java are not without theirs. Among the costs of using Java is that the memory and permanent storage required to run applications tends to be large, resulting in them not fitting on devices with limited capacity.

Inferno is a network operating system that was created to allow applications to be easily and dynamically distributed across networks. With Inferno, applications can easily take advantage of resources in the network such as persistent storage, memory, devices, cpu, server processes etc. as if they were local.

Here, we describe an implementation of Java on Inferno that minimizes the amount of local storage needed on a small device at the cost of a small increase in Inferno's text size. This work is explained and its performance characteristics are reported.

1. Introduction

First some background on Inferno. Inferno is an operating system designed for building distributed services. [2] It has three design principles. First, all resources are named and accessed like files in hierarchical file systems. Second, disjoint resource hierarchies can be joined together into a single, private hierarchical *namespace*. Third, a single communication protocol, *styx*, is used to access all resources, whether local or remote.

Inferno is small, less than 1 megabyte, including all of the kernel, graphics libraries, virtual machine and security code. Inferno is also highly portable running on many systems and processors (x86, sparc, mips, arm, powerpc and more).

Inferno applications have been written in a programming language called Limbo. [3] Limbo has a number of features that make it well suited for a heterogeneous network environment. It has a rich set of basic types, strong typing, garbage collection, concurrency, communications and runtime loadable modules. Limbo may be interpreted or compiled *just-in-time* for efficient and portable execution. Originally, Limbo was the only language available to application programmers under Inferno.

Object oriented designs can be the best way to solve some programming problems, but as with all engineering decisions there are trade-offs. O-O is no exception; the cost of the programming convenience of inheritance, polymorphism and information hiding can be increased program size. [4] On machines with ample memory and disk space, this trade-off may be worth it in terms of overall development costs. Even PCs these days often have over 100 megabytes of RAM and gigabytes of disk space; and workstations and servers have even more. However there are many devices on which you might like to run your applications that are equipped with significantly less. These devices include screen phones, nomadic and wireless computers and several flavours of "set top" boxes (web browsing set-top boxes, video on demand set-top boxes, next generation cable boxes, etc.). Typically these devices have four to eight megabytes of RAM and some flash storage. The effort described here addresses the problem of running on these small devices. To do so, two issues must be addressed: the device must have enough storage for the program text and enough memory in which to execute the application. Java is becoming the programming language of choice for object-oriented designs, but Java is not small. It is effected by both of these issues.

Start with storage. On a sparc, the total size of the 1.1.3 Java shared libraries from Javasoft needed to run the virtual machine is 1.6 megabytes. Under Windows 95, for the 1.1.4 version the size of the java.exe, DLLs and libs is 0.96 megabytes. Besides this, another 9.29 megabytes are needed to store the Java core classes in the *classes.zip* file. Then there is the operating system itself that is needed to execute this code. Together these numbers represent the static storage required by the Java run time environment.

A solution to the first issue, storage for the text for the Java runtime, could be to store it remotely and not on the device. The problem with this approach is that most of these small devices do not have high bandwidth network connections. Most will be equipped with at most a 56K modem. It takes too long to download the ten megabytes of code required to begin to run a Java application.

Second is the issue of the execution footprint needed to run a Java program, that is the memory needed to load the Java virtual machine, possibly just in time compile the classes, create the objects, etc. This, of course, will vary between programs. Even a minimal Java "hello world" on a sparc needs 4.3 megabytes in which to run (not counting any of the operating system). Any Java programs that use the graphical interface (AWT) are considerably larger. A trivial graphical clock

program needs 1032 8K pages (8.24 megabytes) of user space memory in which to run. Devices that have only a total of 8 megabytes of memory cannot run such programs.

The small appliance world of screen phones, set top boxes and wireless computing is considerable different from the internet-centric desk top world in which Java grew up. What is good in one realm is not necessarily viable in the other. In an attempt to address this, Javasoft recently released its *Personal Java* [5] specification which is a slightly reduced subset of the original desk top Java. Implementations of Personal Java are scheduled to be available early in 1998. This slightly limited API reduces the size of the mandatory Java core classes somewhat. But it is still large and reducing the number or complexity of the required system classes has no effect on the size of the native shared object libraries needed to run the virtual machine. Nor does it have any effect on the run time footprint that is required.

The size of code needed for the Java run time makes it impossible to store it on these small devices and the network connections that they have make it impractical to download all the code to the devices on demand. A much smaller system is required, preferably a smaller system that can be easily segmented between the most important bits that could be stored locally on the device and the less important parts that have be to downloaded on demand. Ideally, it should be easy to distribute an application dynamically (what is where) to take advantage of different hardware configurations and network capacity. Inferno is designed to be a system that makes it easy to build such dynamically distributable services.

You might ask then why not just develop the application in a language that was designed for this environment - a language like Limbo. Given a clean sheet, that may often be the best solution. But there will be times when that is not practical. First, you might have already written the application in Java and if it is large enough, it may not be worth rewriting. Secondly, there may be externally produced programs that are written in Java and available to you on the web only as *.class* files that you would like to run on small devices. Third, there will be reasons, even on small devices, why writing a particular application in Java makes sense. Finally, programmers will program in whatever language they prefer. It is of no avail to tell them what you think is good for them.

2. Design Goals

Our goal was to allow Java to run on Inferno while requiring as little local persistent storage and memory as practical. We imposed the following design constraints:

- Adding Java support to Inferno should have no detrimental effect on Inferno. In particular, any increase in the operating system text size required to support Java system should be almost zero. Also any increase in OS data space when Java was not running had to be zero, and no changes required for Java in the

operating system should make it run slower.

- The system should support Javasoft's *Personal Java* API.
- When running Java the system should use as little memory as possible and run as fast as practical.
- The resulting code must be completely portable, running everywhere Inferno runs. It could not be dependent on any platform specific features.

3. Implementation

The Inferno operating system has at its core a virtual machine (VM) called *Dis.* Dis is a memory-to-memory VM that allows small, simple just-in-time compilers to generate efficient code. The Limbo compiler produces virtual instructions for execution on the *Dis* VM.

Dis and the above considerations precipitated a design in which Limbo programs translate, resolve and provide the run time support needed for Java programs. Limbo modules provide the implementations of the native methods on which Java programs depend. Only some minor modifications were made to the Inferno kernel for Java.

The major components of this architecture are:

- A translator called *j2d,* written in limbo that converts the Java bytes codes in *.class* files into *Dis* instructions.
- A *loader,* also written in Limbo, which resolves the class hierarchy, provides a variety of run time support for the Java classes and uses *j2d* to translate the *.class* files as needed.

  To facilitate faster loading, system core classes can be pre-translated from Java byte codes into Dis instructions.
- Most Java implementations rely on a large body of native methods written in C/C++. In fact, the Personal Java proposal defines its JNI, which is in C/C++, as part of its standard. Although Inferno supports C code as drivers, in Inferno, user level code is written in Limbo. So we wrote our native methods in Limbo. This makes them smaller than if they were in C by a factor of 3 though it does make them slower.

  Also two particularly huge Java base classes were re-written in Limbo dramatically reducing their size. By rewriting it in Limbo, the memory needed for *Character.class* was shrunk from 139 K bytes to 48 Kbytes. The requirement for CharacterEncoding and character to byte conversion was reduce from 65K to 18K.

- Only nine simple instructions that required just 36 lines of C code to implement had to be added to the Dis virtual machine. Four of the new instructions support new type conversions and two provide previously unsupported shifts. The last three allocate new structures. These nine are:

```
cvtrf    convert real to float
cvtfr    convert float to real
cvtws    convert word to short
cvtsw    convert short to word
lsrw     logical shift right word
lsrl     logical shift right long
mnewz    allocate class
newaz    allocate (and zero) an array
newz     allocate object
```

  It should be noted that none of these new instructions were required; the translation can be accomplished without them. However, they make the generated code more efficient. These new instructions are also not Java specific; the Limbo compiler can now generate them too.

- Finally, some minor modifications were required in the kernel. These exposed to user level code some interfaces that had previously only existed internal to the operating system. In general, these allow the loader to resolve modules and to create objects that the garbage collector can properly redeem. They were developed as a new *built-in* module that provides these interfaces:

```
ifetch   get a module's instructions
tdesc    get a module's type descriptors
newmod   create a new module
tnew     create a new type descriptor
link     create a new module link
ext      install links
dnew     create module data
compile  just in time compile a module
```

  Several alternatives to this scheme were considered. The first was incorporating a Java virtual machine into Inferno. This was rejected not only because it would increase the size of the system by too much but because it would also result in too many duplicated sub-systems. There would be two virtual machines, two garbage collectors and two sets of just in time compilers. Getting them to all cooperate would be a challenge.

  Another idea that was considered, was the creation of a hybrid virtual machine that could execute either Dis instructions or Java byte codes. Although this

would not increase the size of the system by as much, it was would still make it much larger. Furthermore, since the two machines would now be integrated that cost would always have to be incurred, even when the Java functionality was not needed. The Java functionality could not be easily loaded only when it was needed.

Finally, we considered using Inferno's network capabilities and transparently use a "java server" in the network as if it is a local resource. This can be done, and in situations where there is sufficient bandwidth (for example, cable systems) can be the appropriate solution. But, as stated above, often there may only be a 56K connection into the device. If the application is graphics intensive this will not be enough. Also such a system does not scale well, so a solution that can also execute Java byte codes directly on the edge device needs to be also provided.

During this effort several problems encountered. The first was Java's "class spaghetti". We discovered that Java's system classes are not at all hierarchical. If you select any basic class and find all the classes it references, and find all those classes recursively, you will find yourself referencing every class in the system. This required us to abandon the idea of speeding execution by completely resolving the class when it is loaded. Instead the resolution is done during execution, resolving only what is needed.

A second problem is that translated text size of the *.dis* code that is the output of *j2d* process is larger than the *.class* code that was its input. This is because j2d does not have as much information available to it from the .class files as the Java compiler had at its disposal from the .java files. The best that j2d can do is one-to-one and the code can grow by as much as 100%. An alternative would be to provide a Java compiler that produced Dis instructions directly, instead of producing Java byte code, but that only helps if the Java source code is available, often only the class files will be. It turns out that for most Java programs running on Inferno this is not a problem because the total memory footprint required to execute the Java program including not only the program text itself but all supporting operating system and virtual machine code is smaller using Inferno than it would be using another operating system and its virtual machine. For a sufficiently large Java program this would not be the case. How large the Java program has to be depends on how small the alternative system is, but it is on the order of several megabytes of class file text.

This system does not execute finalization routines. In general, this is not a problem because most of the functionality that the programmer might put in a finalization routine, such as closing open files is finessed by the garbage collector, the Inferno operating system takes care of them. Also the programmer cannot according to the Java language specification depend upon the finalization routine being executed at any specific point in time. The

specification is nebulous enough in this respect that its letter if not its spirit can be met without ever executing these routines.

4. Results

The implementation that we selected resulted in a system that has the following characteristics:

- Kernel text size was increased by less than 11 Kbytes. This includes all the code for the new instructions and the new module built-in into the kernel. This adds less than 2% to the size of an Inferno kernel.
- Text space for the modules required to run Java on Inferno: j2d, the loader and assisting modules is just 176 Kbytes. This is distributed as follows:

105K bytes	for the Java to Dis translator
41K bytes	for the loader
3K bytes	dis assembly language utilities
27K bytes	miscellaneous

  This 176K of code encompasses the functionality of about a megabyte of DLL or shared libraries. Several factors are responsible for this dramatic decrease in size. First, the Dis instructions are more concise than native machine code. In general Dis instructions sequences are a third the size of the corresponding native code. Second Limbo is an efficient language in which to develop an application like this. Third, doing this work on Inferno, allows the leveraging of much of the Inferno functionality. For example, the Inferno virtual machine and garbage collector can be used. A virtual machine or garbage collector need not be rewritten. This is significant savings.

  If some local storage exists, using 176K of be practical in many situations where using several megabytes to store the shared libraries or DLLs would not be.
- Through the use of the Inferno namespace none of these modules need be resident on the small device. If hardware does not have this amount of local storage the modules can be remote but bound into the namespace of the application and be treated as if they are local. Of course, something analogous could be attempted through NFS, by mounting a remote file system that contains the Java virtual machine and its shared libraries and system classes. If your small device is also bandwidth poor using NFS would not be practical. Downloading the Java virtual machine, across a 33Kbit per second link, when you want to run a Java application would take too long. But downloading the entire 176K, even at that speed, is possible.

Of course it might not be possible to fit the code for NFS onto the device in the first place (whereas Inferno has its networking code built into it).

Furthermore, because Limbo modules are only loaded when the execution path of the program first requires them, often much of the 176K need not be loaded at all.

The same can be done for any pre-translated core system classes that are stored as .dis files. The entire contents of the *classes.zip* file need not traverse the link but only the classes that are used.

Also since the Inferno namespace allows disjoint hierarchies to be unioned into a single private hierarchy, some important classes can be local why others are in the network. Since the application, here the Java program, never sees the network, the distribution of what is where can vary between platforms, or even between executions on the same platform without requiring any change in the application.

- Both Limbo and Java allow modules to be "just in time" compiled on the target hardware. This *jitting* translates the machine independent codes into native machine instructions just before the code is executed. The goal of this late stage compiling is to maintain the portability of the code without incurring all the speed disadvantage of interpreted code. Jitting has two disadvantages: it increases the text size by a factor of about three and it incurs the execution time cost of the translation to native instructions. If the translated code is to be repeatedly executed this tradeoff may be worth it. Originally jitting Java code was an all or nothing decision, either the virtual machine compiled all the code it ran or none of it. More recently "hotspot" jitting has been proposed in which the virtual machine attempts to determine which pieces are most worthy of compiling. This scheme promises to address the all or nothing problem but further complicates and increases the size of the Java virtual machine. Inferno, in contrast, has always allowed the application module developer three choices on a per module basis:

  1. always compile the module,
  2. never compile the module, or
  3. let the virtual machine take its default. (The VM's default is an execution time option.)

  These choices allow the developer to indicate to the system, considering speed and size constraints, exactly which modules should be compiled.

- Inferno has a hybrid garbage collecting algorithm that reference counts most data. [6] A background concurrent and incremental mark and sweep collector recovers any cyclic data that cannot be collected by reference counting. [7] This has several advantageous over stop the world memory recovery. Most important to us here is that on average less memory is

needed since it is recovered as soon as it goes out of scope instead of having to wait for the next epoch.

Inferno is small even with the additional code for Java. Here is the size of Inferno operating system on several different platforms:

processor	text	data	bss	total (bytes)
386	466593	86944	62020	615557
arm	955168	99920	54200	1109288
sparc	911304	96496	45128	1052928

These numbers vary some with what device drivers are present, but these are typical values for actual systems.

The amount of memory, beyond that needed for the OS and virtual machine, required to run the application varies greatly with the application. But a growing number of small Inferno enabled devices have been built. From these we can see how much memory typical applications on Inferno require. These Inferno devices run as video-on-demand set top boxes, web browsing set top boxes, screen phones and network computers. They have between 2 and 8 meg of RAM. Running Java on them requires at least 8 megabytes, but for Java that is small. Tiny Java applications can use that much user space, without considering all the memory for the operating system support that they require. (For example, a minimally equipped Javastation, a sparc, comes with 32 Meg of memory.)

As stated at the beginning of this paper there are always tradeoffs. This design attempts to minimize memory usage and maximize portability. It runs on sparc, x86, arm and other processors. It runs on screen phones with as little as a total of 8 Megabytes of RAM. An implementation targeted for a specific processor and with no memory limit can always be made to out perform a system designed to run everywhere in limited memory. Here are some benchmarks for this system. As always mileage will vary, but to examine the performance of the system we used Pendragon's Embedded Caffeinemark ™ 3.0 [8] This benchmark is widely accepted as a measure of Java speed and is readily available on the web. It uses six sub-tests to quantify aspects of the Java virtual machine speed. The following is a brief description of each sub-test:

1. A sieve of Eratosthenes finds prime numbers.
2. The loop test measures compiler optimized loops.
3. The logic test tests the speed of decision making instructions.
4. The string test manipulates strings.
5. A float test simulates a 3D rotation around a point. And

6. The method tests recursive function call speed.

The following two tables show results for several specific platforms.

The first column of the first table is for a screen phone equipped with a 200 MHz strongARM processor, 8 megabytes of RAM and 2 megabytes of flash memory that appears as a disk. (In these tables larger values are better.)

Table 1 Inferno Caffeinemarks

	screen phone	set top box
Sieve score	83	22
Loop score	80	20
Logic score	80	58
String score	45	3
Float score	13	0
Method score	67	7
Overall score	329	0

The results for the screen phone running Inferno, are no where near the best Caffeinemarks in the world, but for a machine with just 8 megabytes of ram and 2 meg of flash they are impressive.

It would be interesting to compare these results with published reports of other Java virtual machines running on similar platforms. But such data does not yet exist.

For comparison, the first column second table column shows the results of the same test on a 200 megahertz Pentium running Javasoft's version 1.1.4 and the second column shows the results on the same machine running Inferno with the jit enabled.

Table 2 Pentium Caffeinemarks

	Javasoft jvm	Inferno
Sieve score	364	679
Loop score	278	992
Logic score	343	2676
String score	533	48
Float score	298	222
Method score	315	266
Overall score	346	415

As you can see Inferno's speed here is slightly better than Javasoft's JVM.

As a more extremely limited example, the second column of the first table contains the results on a set-top (web-browsing) box equiped with a 40 MHz 386 with 8 megabytes of memory, 8 megabyte of flash and a 28.8 hardware modem.

In this case the low score on the float test makes the overall score zero, but it is unlikely that another JVM would do better on this platform. The Pendragon web site that publishes the Caffeinemark results does not list any results for a 386 processor. This example shows that by using Inferno, Java can be executed even on extremely limited platforms.

5. Summary

The Inferno operating system is designed to allow distributed network applications to be easily developed and efficiently run on small devices. By taking advantage of Inferno's features we were able to develop a Java implementation that can run on small devices. This scheme is not the fastest Java engine but it is small both statically and dynamically. This allows Java to be used in places where it otherwise could not be.

References

1. Gosling: The Feel of Java. Computer IEEE V30, 6 (1997) 53-&
2. Dorward, Pike, Presotto et al.: Inferno. Proceedings IEEE COMPCOM (1997)
3. Dorward, Pike, Winterbottom: Programming in Limbo. Proceeding IEEE COMPCOM (1997)
4. Vazquez: Selecting A Software Development Process. Communications of the ACM (1994)
5. http://www.javasoft.com/products/personaljava/spec-1-0-0/personalJavaSpec.ht
6. Richard E. Jones and Rafel D. Lins: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley (1996)
7. Kafura, Mukherji and Washabaugh: Concurrent and Distributed Garbage Collection of Active Objects. IEEE Transactions on Parallel and Distributed Systems Vol 6. No 4 (1995)
8. http://www.pendragon-software.com/pendragon/cm3/index.html