

ABSTRACT

Title of Thesis: A RETARGETABLE OPTIMIZING JAVA-TO-C
COMPILER FOR EMBEDDED SYSTEMS

Ankush Varma, M.S., 2003

Thesis directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering

The Java programming language is achieving greater acceptance in high-end embedded systems such as cellphones and PDAs. However, low-end embedded platforms, such as DSPs or microcontrollers, often have no more than a C compiler, and this prevents Java applications from being run on such systems. Applications must either be re-written in C, or a Java Virtual Machine must be ported to each such system.

This paper discusses a compiler that converts portable Java bytecode to C code, allowing applications written in Java to run on embedded systems which may lack a Java Virtual Machine. This is also applicable to bare-bones embedded systems running without an operating system. We briefly describe code generation strategies, run-time data structures and optimization algorithms used to generate efficient C code. The code size and execution time of the C code were compared with interpreted Java, just-in-time compiled Java, and executables generated directly from Java.

On an average, we found the size of the generated stand-alone executable to be over 25 times smaller than that generated by a cutting-edge Java-to-native-code compiler, while providing performance comparable to the best of various Java implementation strategies.

A RETARGETABLE OPTIMIZING JAVA-TO-C COMPILER
FOR EMBEDDED SYSTEMS

by

Ankush Varma

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2003

Advisory Committee:

Professor Shuvra Bhattacharyya, Chair
Professor Bruce Jacob
Professor Manoj Franklin

ACKNOWLEDGEMENTS

This thesis has been possible because of the support and encouragement of the following people:

Professor Shuvra Bhattacharyya, under whose supervision I chose this topic and performed the work that went into this thesis. This project was rather daunting at first glance, and his help and guidance has been invaluable.

Christopher Hylands of the EECS department at Berkeley, whose technical finesse and awe-inspiring programming and debugging skills helped set up much of the testing framework and Ptolemy interfacing, in addition to getting me out of many a tight spot.

Steve Neuendorffer of the EECS department at Berkeley, whose help with troubleshooting has been most helpful.

Ming-Yung, for patiently helping me get set up and started.

My parents, for being there. And for getting me started on the path that has led me here.

Thanks are also due to Mainak, for playing a mean game of racquetball, Brinda, Zoltan and Radost for being great hiking partners, and Suvarcha, Sada, Pushkin and IyerB, for keeping life fun.

This research was supported by the DARPA MoBIES program, through U.C. Berkeley.

TABLE OF CONTENTS

1.	Introduction.....	1
1.1	Embedded Systems.....	2
1.2	C	3
1.3	Java.....	5
2.	Related Work	9
2.1	JVM-based execution	9
2.2	Stripped-Down Java	10
2.3	Mixed-Mode execution	11
2.4	Java-to-Native Compilation.....	12
3.	Statement of Problem.....	14
4.	Implementation	15
4.1	Runtime Data Structures.....	15
4.1.1	Naming Conventions.....	15
4.1.2	Data and Code Layout.....	16
4.1.3	Referencing Objects, Methods and Fields.	18
4.1.4	Arrays.....	19
4.2	Code Generation.....	20
4.2.1	Interfaces.....	21
4.2.2	Exception Handling.....	21
4.2.3	Native Methods	23
4.2.4	User-Defined Code.....	24
4.2.5	Garbage Collection	24
4.2.6	Start-up.....	26
4.3	Code Pruning Strategy.....	27
4.3.1	Analysis.....	28
4.3.2	Computing the Set of Required Entities	29
4.3.3	Pruning and Code Generation.....	30
4.3.4	Compilation Time	31
5.	Limitations	33
5.1	Dynamic Loading	33
5.2	Reflection	34
5.3	Security.....	34
5.4	Threads	35
5.5	Portability	35
6.	Performance Studies	37
6.1	Benchmarks	37
6.2	Methodology	38

6.3	Results	39
6.3.1	Performance	39
6.3.2	Code Size	41
7.	Conclusion and Future Work	45
7.1	Contributions	45
7.2	Future Work	45
8.	Appendices.....	47
8.1	Example of Generated Code.....	47
	References.....	50

LIST OF TABLES

TABLE 1.	Naming conventions for prefix characters.....	16
TABLE 2.	Structure of Class Descriptor Table	17
TABLE 3.	C equivalent of Java references	19
TABLE 4.	Percentage improvement in performance of Java-to-C strategy over any other approach. The table shows the amount by which the performance improvement of a Java-to-C implementation over the highest-performing of the other strategies.	41
TABLE 5.	Sizes of generated executables for various applications. (Kilobytes).....	43

LIST OF FIGURES

Figure 1.	Performance for various benchmarks.	41
Figure 2.	Sizes of generated executables for various applications (Kilobytes).....	43

1. Introduction

The Java programming language is achieving greater acceptance in high-end embedded systems such as mobile phones and PDAs. However, low-end embedded platforms, such as DSPs or microcontrollers, often have no more than a C compiler, and this prevents Java applications from being run on such systems. Applications must either be re-written in C, or a Java Virtual Machine must be ported to each such system.

This document discusses a compiler that converts portable Java bytecode to C code, allowing applications written in Java to run on embedded systems which may lack a Java Virtual Machine. This is also applicable to bare-bones embedded systems running without an operating system. We briefly describe code generation strategies, run-time data structures and optimization algorithms used to generate efficient C code. The code size and execution time of the C code were compared with interpreted Java, just-in-time compiled Java, and executables generated directly from Java.

On an average, we found the size of the generated stand-alone executable to be over 25 times smaller than that generated by a cutting-edge Java-to-native-code compiler, while providing performance comparable to the best of various Java implementation strategies.

1.1 Embedded Systems

An embedded system is a special-purpose computer system built into a larger device. Embedded systems are typically required to meet requirements that are very different from the requirements of general-purpose personal computers.

Two major areas of difference are cost and power consumption. Many embedded systems are produced in the range of tens of thousands to millions of units, making cost reduction a major concern. Embedded systems often use a (relatively) slow processor and small memory size to minimize costs. The system architecture is often intentionally simplified to lower costs. For example, embedded systems often use peripherals controlled by synchronous serial interfaces, which are ten to hundreds of times slower than comparable peripherals used in PCs.

In addition, programs on an embedded system are often required to satisfy certain real-time constraints. Embedded systems also often lack basic components that are ubiquitous in desktops, such as disk drives, operating systems, keyboards and display screens.

There are a host of different microprocessor architectures used in embedded designs. This in contrast to the desktop computer market, which is limited to just a few competing architectures, chiefly Intel's x86, AMD's Athlon/Duron and the Apple/Motorola/IBM PowerPC, used in the Apple Macintosh.

Although early embedded systems were programmed in assembly language, the focus soon shifted to C, which allowed faster programming and easier debugging. The move towards higher-level languages allows more functionality to be developed in same amount of developer time. The trend towards programming in higher-level languages has continued, with C++ compilers being made available for some systems. Java is also gaining popularity amongst high-end embedded systems such as PDAs and cellphones.

1.2 C

The C programming language [1] was developed at Bell Laboratories in 1972 by Dennis Ritchie. Its power and flexibility became apparent soon after its invention, and caused the Unix operating system to be almost immediately re-written from assembly language to C (except for a few lines of “bootstrap” assembly code). During the rest of the 1970's, C spread to many colleges and universities because of it's close ties to Unix and the availability of C compilers. Soon, many different organizations began using their own C versions, causing compatibility problems. In 1983, the American National Standards Institute (ANSI) responded to this by forming a committee to establish a standard definition of C which became known as ANSI Standard C [1]. More functionality was added to this by the GNU project [2], and GNU C exists as a superset of ANSI C.

C is a powerful, flexible language that provides fast program execution and imposes few constraints on the programmer. It allows low level access to information and commands while retaining the portability and syntax of a high level language. These qualities make it a useful language for systems programming as well as general purpose programs.

C's flexibility stems from the multiple approaches available to the programmer to accomplish the same tasks. It contains richly expressive constructs such as bitwise operators, pointer manipulations, multi-dimensional arrays and function pointers. C imposes few constraints on the programmer. The main area this shows up is in C's lack of type checking. This can be a powerful advantage to an experienced programmer but a dangerous disadvantage because of the lack of safety features.

Another strong point of C is its modularity. Sections of object code can be stored in libraries for re-use in future programs. This concept of modularity also helps with C's portability and execution speed. The core C language excludes many features included in the core of other languages. These functions are instead stored in the C Standard Library where they can be called on as needed. An example of this is C's lack of built in I/O capabilities. I/O functions tend to slow down program execution, and they are platform-dependent when optimally imple-

mented. For these reasons, they are stored in a separate library, and are included only when necessary.

However, there have been other advances in programming languages since the invention of C. Many powerful concepts such as threads, object-oriented programming, type-polymorphism and automatic garbage collection are not fully or directly supported in C. In addition, even strict standards-compliant C code is not fully portable.

Despite these drawbacks, its popularity, flexibility and ease of low-level access have made C the current language of choice for a variety of applications, including operating systems and embedded applications.

1.3 Java

The Java programming language [3] was originally called Oak, and was designed for use in embedded consumer-electronic applications by James Gosling. After several years of experience with the language and significant modifications, it was retargeted to the Internet, renamed, and substantially revised to be the language now called Java.

The Java programming language is a strictly typed, general-purpose, concurrent, class-based, object-oriented programming language, specifically designed to

have as few implementation dependencies as possible. It allows application developers to write a program once and then be able to run it everywhere on the Internet.

Java was designed to be a system that could be programmed easily without a lot of esoteric training, and which leveraged standard practices. One of the most popular programming languages is C, and many programmers doing object-oriented programming used C++ when Java was introduced. So even though the Java designers found that C++ was unsuitable, they designed Java as closely to C++ as possible in order to make the system more comprehensible [27].

Java omits many features of C++ that the designers of Java considered to be rarely used, poorly understood or confusing [27]. These omitted features primarily consist of operator overloading (although the Java language does have method overloading), multiple inheritance, extensive automatic coercions and pointer arithmetic.

Java also has automatic garbage collection, thereby simplifying the task of Java programming but making the system implementation somewhat more complicated. A common source of complexity (and therefore errors) in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection (periodic freeing of memory not being referenced) the Java language not only makes the programming task easier, it also cuts down on coding errors¹.

There are two conventional approaches to executing programs: compilation and interpretation. The standard Java implementation uses both. In compilation a compiler converts the program's source to into machine code that can be directly executed on the computer. This is the approach taken by C [1], C++ [34][35] and Pascal [36]. In an interpreted language, an interpreter parses and executes the source code in a line-by-line fashion. This is the approach taken by Basic, Visual Basic and most shell scripts.

Compiled programs often run faster because they use machine code instructions that can directly run on a computer. Interpreted programs are run on top of the interpreting program are hence much slower. However, an interpreter acts like a buffer between the program and the computer and shields the computer from erroneous instructions that could affect the operation of the computer.

To enable platform-independence and enforce security policies, Java programs are compiled into an interpretable platform-independent form. This compiled form is called *byte code*. Bytecode and other information for each class is stored in a *class file*, which can be loaded and executed in an interpreter called a *Java Virtual*

1. This assertion is made in [27], but we found that it is indeed verified by our experience. All 8000 lines of Java code in the Java-to-C compiler were debugged with print statements alone. Java's built-in exception mechanism and lack of pointer arithmetic obviated the need for using a complex Java debugger. By contrast, the 500 lines of runtime C code required debugging with gdb for errors, which were primarily related to memory allocation, memory alignment and pointers to memory locations. Java's automatic memory management and lack of pointer arithmetic ensures that none of these classes of bugs can occur in a Java program.

Machine (JVM). This enables a Java program to be executed on any platform (architecture and operating system) that supports a JVM.

A virtual machine is any machine written in software, that provides an abstract layer to applications. In a layered application the virtual machine acts as the lowest layer, providing abstraction over the native environment. If an application is be ported to another environment only the virtual machine needs to be re-implemented. The application layers written above it remain unchanged, as they use the same abstract interface to the virtual machine.

2. Related Work

Over time, a number of implementations for Java have been explored. This section discusses various approaches to running Java code on a variety of computer systems.

2.1 JVM-based execution

Full-featured Java Virtual Machines are the standard Java implementation [4]. A Java compiler converts each Java class to a class file. The JVM loads and runs bytecode instructions from class files. Examples of such implementations are the standard Java Virtual Machine from Sun Microsystems [28], the Jikes Research Virtual Machine [29] [30] developed at IBM and the Kaffe Virtual Machine [31]. All of these support the Java Virtual Machine Specifications [4] and support all Java features and a full Java class library.

Early JVMs were simple interpreters, and simply executed each bytecode instruction sequentially. This involved a computational overhead, since the code was not running directly on the microprocessor. Current JVMs improve execution efficiency by employing just-in-time (JIT) compilation [32][33]. JIT compilation involves converting the bytecode for each method to machine code immediately prior to its first execution. This removes the interpretation overhead and results in execution speedup, at the cost of having a more complex JVM. However, a JIT

approach precludes global or expensive optimizations, and is restricted to performing local “peephole” optimizations.

2.2 Stripped-Down Java

A full-featured Java Virtual Machine and class library is appropriate for desktop systems, but constraints on code size, execution speed and power make this approach inappropriate for many embedded systems. The more stringent resource requirements of embedded systems can often be met by:

- Supporting a subset of Java, and excluding features such as dynamic loading, reflection, proxies, multithreading and certain data types that may not be required for the target application(s).
- Using a minimal class library instead of the complex and large Java class library. The Java class library provides a large amount of functionality to the programmer for ease of development, but a much smaller library may be more appropriate for embedded systems.

This approach does not allow standard Java code to run on an embedded platform. Rather, it allows code to be written in Java with a particular embedded implementation in mind.

Sun’s KVM (K Virtual Machine) [5] is part of the Java 2 Micro Edition (J2ME) targeted at resource-constrained and embedded systems. It does not support floating-point data types, reflection and object finalization methods, and it places some limitations on threads. It supports a minimal class library and has a

code size² of ~100KB. The small code size constraint does not allow JIT-compilation. Sun's CLDC HotSpot Virtual Machine [6] (also part of J2ME) enables JIT-compilation, resulting in significant speedup. However, it increases the code size to ~500KB.

WabaSoft's Waba Virtual Machine [25] is another pruned-down Virtual Machine that supports a strict subset of Java. It excludes long data types, double data types, support for exceptions and threads. It provides a small specialized library, so programs need to be written specially for Waba. The code size of a Waba implementation is ~100KB. A successor to Waba is SuperWaba [26] which provides more complete support while increasing code size to ~300KB.

2.3 Mixed-Mode execution

JIT-enabled JVMs cannot perform the kind of global optimizations that can be performed by an ahead-of-time compiler. However, a statically generated executable may not support the full dynamic nature of the Java language (features such as proxies, reflection and dynamic loading). A solution to improve Java performance is to statically convert some of the bytecode to machine code, but also have a JVM to perform certain tasks. Part of an application may run as bytecode on the JVM, while other parts of the application run directly as machine code.

2. Code size here is defined as the combined size of the Virtual Machine and class library.

TurboJ [7][8] compiles certain methods into machine code, and uses special class files called “interludes” that allow methods that have been converted into machine code to call methods that exist as bytecode. It relies on the JVM for object management, garbage collection, thread management and class and library loading. Since it has a full JVM at hand, TurboJ meets all Java specifications.

Harissa [9][10] (originally named Salsa) is a bytecode to native code compiler that uses C as an intermediate language. It links in a bytecode-interpreting JVM into the generated executable to enable dynamic loading of bytecode. It does not support current or recent Java versions.

It must be noted that the mixed-mode execution improves performance, but it does not necessarily reduce code size, because it still requires a JVM and class library.

2.4 Java-to-Native Compilation

It is also possible to convert bytecode or Java source code to platform-specific machine code. Static compilation makes extensive optimizations that cannot be made by a just-in-time compiler, and thus generates more efficient machine code. This necessarily involves some loss of portability, since the generated machine code is platform-specific. There may also be some restrictions on functionality. However, this strategy is a useful option for embedded systems, where code efficiency in terms of both performance and size may be extremely important.

The `Vortex` compiler infrastructure [11] provides `Cecil`, `C++`, `Modula-3` and `Java` front-ends. It allows `Java` code to be compiled into executable form, and serves as a test-bed for various compiler optimizations. `Caffeine` [12] is also a preliminary prototype of a `Java` to native code compiler.

`gcj` [13] provides a highly sophisticated and standardized method for compiling `Java` source code or bytecode into native executable form. It provides a complete runtime environment for `Java`, and is a cutting-edge `Java`-to-Native code compiler.

`Toba` [14] is a well-designed `Java` compiler that compiles `Java` to `C` and then uses a platform-dependant `C` compiler to generate executable code. It does not support current `Java` versions. We build on some of the concepts described in [14] further, implementing features and optimizations specific to embedded systems.

3. Statement of Problem

Is it possible to achieve a Java implementation via C that meets the low memory-size footprint requirements of embedded applications? What is the impact of such a scheme on execution speed? How much of Java functionality can be supported by such a scheme?

We designed an elegant Java-to-C code-translation scheme, and coupled it with a sophisticated code pruning algorithm. We treated this as an example of a C-based static compilation strategy, and compared both performance and code size with other Java implementations. In this document, we also discuss the potential limitations of such a scheme in terms of functionality, based on our experience in the design process.

4. Implementation

4.1 Runtime Data Structures

The Java object model provides a rich set of features to describe object types, methods and fields. We provide data structures that emulate this functionality within C. This data layout strategy is a direct adaptation of the strategies used in many virtual machines, and is similar to that described in Toba [14].

4.1.1 Naming Conventions

Java allows identifier names to be unlimited strings of unicode characters, whereas C requires all identifiers to be ASCII characters of 63 or fewer characters. Further, merely the name of a Java class, method or field does not uniquely identify it. A Java class is uniquely identified by its package and name. Similarly, a Java method is uniquely described by its class and signature, since Java methods may be overloaded within a class, or among different classes.

To prevent two distinct Java entities from being mapped to the same C name, we generate C names by removing characters not permitted in C identifiers, and adding a unique hash-code prefix. This enables all methods, classes and fields to share a global namespace.

The C name corresponding to a class, method or field consists of prefix characters, followed by a numeric hashcode, and then then its Java name. The name may be *sanitized* (illegal characters may be removed and the resulting string may be truncated) to make it a legal C identifier. The hashcode is generated using a robust hashing scheme that is consistent across different runs of the compiler and across different platforms to ensure that a Java entity always corresponds to the same C name.

The prefix characters are required because C names cannot start with a numeric character. We use different prefix characters for different types of named objects. These are specified in Table 1.

TABLE 1. Naming conventions for prefix characters.

Type of named object	Prefix characters
Instance Structure corresponding to a Java class	<i>Vi</i>
Pointer to instance structure	<i>i</i>
Class Structure corresponding to a Java class.	<i>C</i>
C function corresponding to non-native Java method	<i>f</i>
C function corresponding to native Java method	<i>n</i>
Function pointer within a class structure	<i>m</i>

4.1.2 Data and Code Layout

Java primitive types are mapped to primitive C types of the appropriate size. Java objects are *reference types* which extend `java.lang.Object`. Reference types are translated into C pointer types. Each reference points to an *instance*

structure in C. The instance structure contains all instance-specific information (such as non-static fields), and a pointer to a common *class structure*. There is a single class structure corresponding to each class, which contains three sub-structures: the *class descriptor table*, the *methods table*, and the *class variables table*.

The class descriptor table contains information that is needed across all classes, such as the name of the class, a pointer to the superclass etc. The major fields of the class descriptor table are shown in Table 2.

TABLE 2. Structure of Class Descriptor Table

char* name	Character String containing the name of the class.
int instance_size	The number of bytes in instance structures of this class.
void* superclass	Pointer to class structure of parent class.
short Array	Indicates whether the class is an array.
void* (*lookup) (int)	Pointer to function that resolves polymorphic interface method invocations at runtime.
short (*instanceOf) (void*, long)	Pointer to function that resolves “instanceof” queries at runtime.

The method table contains a table of pointers to functions that implement the various methods of the class. The entries for methods present in the parent class come first, followed by methods present in this class, and absent in the parent class. The ordering of methods is maintained from class to subclass. This precise ordering enables type polymorphism, by allowing a class to be treated as any of its superclasses. This is because the entry corresponding to a given method will occupy the same location in the class structure of all subclasses of any given class,

and its location will be invariant when a class structure is cast and indexed as the class structure of a superclass.

The class variable table contains class variables, such as static fields. Note that all non-static fields are members of the instance structure, not of the class structure.

4.1.3 Referencing Objects, Methods and Fields.

References to Java Objects are translated into pointers to the corresponding instance structures. Method references are changed into the appropriate function pointers, and field references become pointers to fields of the corresponding structure.

The code below shows a sample Java class “Circle”, with an instance “c”. Examples of C equivalents of references to members of c are illustrated in Table 3.

```
public class Circle implements SomeInterface {
//Field
int radius;
// Method.
int getRadius();
// Static method.
static String getType();
// Method from SomeInterface.
void move(int x, int y);
}
...
```

Circle c;

TABLE 3. C equivalent of Java references

Reference Type	Java Code	C Code ^a
field	c.radius	c->radius
instance method	c.getRadius()	c->class->getRadius(c)
Static method	c.getType()	c->class->getType()
Interface method	c.move(x, y)	c->class->lookup(9721)(c, x, y)

a. Hashcode prefixes to the names of C identifiers are omitted for clarity.

4.1.4 Arrays

Java arrays are objects, are dynamically created, and may be assigned to variables of type `Object`. All methods of class `Object` may be invoked on an array.

An array object contains a number of variables. The number of variables may be zero, in which case the array is said to be *empty*. The variables contained in an array have no names; instead they are referenced by array access expressions that use nonnegative integer index values. These variables are called the *components* of the array. If an array has n components, n is the *length* of the array; the components of the array are referenced using integer indices from 0 to $n - 1$, inclusive.

We treat arrays as special objects. Their class descriptors can be set at runtime, and all arrays of a given type share the same class descriptor. As with normal classes, the instance structure of an array contains a pointer to the class structure.

Java allows elements of an array to be arrays. Multi-dimensional arrays are treated as arrays of arrays. Unlike the handling of multidimensional arrays in C, this allows non-rectangular arrays to be created. We created C functions and macros to emulate all Java functionality for array initialization, array access and array declarations

4.2 Code Generation

The Java-to-C translation framework is written entirely in Java. It takes Java class files as input. These contain Java *bytecode*, which is a stack-based low-level description, and is not suitable for direct translation to C. For analysis of the bytecode, we make extensive use of Soot [15][16][17], a sophisticated Java bytecode analysis and optimization framework developed at McGill University. Soot allows bytecode to be transformed into *Jimple* [18], a typed 3-address intermediate representation designed to simplify analysis and transformation of Java bytecode. C code is then generated based on the Jimple representation.

For each class required by the application, the compiler generates a C file containing all required methods, including implicit initialization methods described in [3] and [4]. We also generate a header file containing various declarations and type definitions of the class structure and the instance structure. A makefile tailored to the target system is also generated for allowing the platform-specific C compiler to create an executable.

4.2.1 Interfaces

Type polymorphism arising from class-subclass relationships is easily resolved via the structure of the method table discussed in Section 4.1.2 on page 16 because each class (except `java.lang.Object`) has exactly one parent. Java does not allow multiple inheritance. However a class may implement an arbitrary number of *interfaces*, as described in [3]. Calls to methods defined in interfaces are also polymorphic in nature, i.e. there is a set of methods that are possible targets of the method call, and there may not be a single statically known target.

Such invocations are resolved by a per-class lookup method that takes the hashcode of the interface method called as an argument and returns a pointer to the appropriate function by performing a table-lookup operation at runtime.

4.2.2 Exception Handling

According to the Java Language Specification [3],

“When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an *exception*.

An example of such a violation is an attempt to index outside the bounds of an array. Some programming languages and their implementations react to such errors by peremptorily terminating the program; other programming languages allow an implementation to react in an arbitrary or unpredictable way. Neither of these approaches is compatible with the design goals of the Java platform: to pro-

vide portability and robustness. Instead, the Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be *thrown* from the point where it occurred and is said to be *caught* at the point to which control is transferred. Programs can also throw exceptions explicitly, using `throw` statements.

Explicit use of `throw` statements provides an alternative to the old-fashioned style of handling error conditions by returning funny values, such as the integer value `-1` where a negative value would not normally be expected. Experience shows that too often such funny values are ignored or not checked for by callers, leading to programs that are not robust, exhibit undesirable behavior, or both.

Every exception is represented by an instance of the class `Throwable` or one of its subclasses; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by `catch` clauses of `try` statements. During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the

class of the exception. If no such handler is found, then the method `uncaughtException` is invoked for the `ThreadGroup` that is the parent of the current thread-thus every effort is made to avoid letting an exception go unhandled.”

Exception handling in the JVM uses a program counter to keep track of the point at which an exception was thrown. We emulate this by using a global exceptional program counter (*epc*). The *epc* is changed every time a *trap* (a range of instructions corresponding to an exception) is entered or exited.

We use the *setjmp* and *longjmp* routines to handle the non-local jumps and call-stack unwinding associated with exception handling. When a function is entered, the global *jmpbuf* and *epc variables* are stored to local variables and a *setjmp* call is made. When an exception is thrown, *longjmp* is called to return control to this *setjmp* instruction, and a table-lookup is done to find the point within the function that catches the type of exception thrown. If the function does not catch the type of exception thrown, the previous *epc* and *jmpbuf* are restored and the exception is re-thrown to transfer control to the caller.

4.2.3 Native Methods

Java requires certain *native* methods, which are methods implemented in platform-dependent code, typically written in another programming language such as C.

This compiler allows the user to specify C code for the body of any native method. At compile-time, this is integrated with the generated C code, allowing any C native methods to be fully supported. We created a short C file for the body of each required native method.

4.2.4 User-Defined Code

All standard Java library classes have a predefined behaviour that cannot be easily modified by the user. This may be too restrictive for embedded systems designers, since it precludes hand-optimization of critical code, or easy use of specialized I/O. We relax this restriction by allowing the user to define the C code for the body of *any* method, not merely native methods.

This allows easy adaptation of standard Java classes to embedded-system-specific uses. For example, the code for the method `PrintStream.print(boolean)` can be defined to turn an LED on the embedded board on or off, while printing “true” or “false” to a screen on a desktop. This allows a developer to specify platform-specific or optimized code for any method.

4.2.5 Garbage Collection

One of the key features of Java is its heap is automatically garbage-collected. Dynamically allocated memory that is no longer referenced is freed automatically. The JVM's heap stores all objects created by an executing Java program. Objects

are created by Java's "new" operator, and heap memory for new objects is allocated at run time. Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. Objects that are no longer in use are destroyed and memory freed up without any explicit programming directive (by contrast, C requires memory allocated with *malloc()* to be explicitly deallocated with *free()*).

A garbage collector may also combat heap fragmentation, which occurs during normal program execution. When referenced objects are freed and new objects are allocated, free blocks of heap memory may be left in between blocks occupied by live objects. New allocations may have to be serviced by increasing the heap size, although enough total unused space is available in the existing heap. This will happen if the size of the allocated object exceeds the size of the largest contiguous block of free heap memory. On virtual memory systems, the extra paging required to service increasing heap sizes can cause performance degradation. On memory-poor embedded systems, such fragmentation can leave the system out of memory.

The JVM specification [4] specifies that the heap of the Java virtual machine must be garbage collected. It does not define how the garbage collector must work. The designer of each JVM must decide how to implement the garbage-collected heap.

Our implementation uses the publicly available Boehm-Demers-Weiser conservative garbage collector [20][21]. A conservative collector checks every register and all allocated memory for potential pointers, and traces the transitive closure of all memory reachable from these pointers. It does not require type information for managed memory, and thus memory management is transparent to the programmer.

4.2.6 Start-up

The Java virtual machine starts execution by invoking the method `main` of some specified class and passing it a single argument, which is an array of strings (usually the command-line arguments). This causes the specified class to be loaded, linked to other types that it uses, and initialized.

Our approach uses a static executable created from C code, so dynamic loading/linking is not involved. However, the `main` C function takes command-line arguments in a manner very different from the way the `main` Java method takes arguments. The start-up steps in the generated C `main` function are:

1. Set up the class structures for all required classes.
2. Call the `java.lang.System.initializeSystemClass()` method. In a standard implementation, this would be implicitly called by the JVM.
3. Parse the command-line arguments (using the `argc` and `argv` C constructs) into the C equivalent of an array of Java strings.

4. Invoke the method corresponding to the Java main method, and pass it this array of strings as an argument.

This makes the Java application aware of all comand-line arguments, and performs the initializations that allow it to execute correctly.

4.3 Code Pruning Strategy

Java classes tend to derive a lot of functionality from other Java classes, in a highly interlinked manner. The simplest Java class can require over 250 other classes for execution³.

All Java classes are subclasses of `java.lang.Object`. In addition, classes reference fields and methods in other classes, throw exceptions (all exceptions are Java classes) and have local variables that may be objects belonging to other classes. C code needs to be generated for all classes, methods and fields that may be accessed.

Simply translating all classes that are referenced by the main class into C fails. This is because each of these classes will have methods or fields that are not used by the main class. These unnecessary methods and fields can reference additional classes, so all those classes will also need to be compiled unnecessarily.

3. This can be seen by running "`java -verbose`" on a standard "Hello World" program.

A simple solution is to compile *all* Java library classes into a library and load the required ones at runtime. This is the approach used by Toba [14] and `gcj` [13]. This simplifies compilation and linking, but considerably increases code size because the size of this library can be of the order of megabytes, and this may be too costly to implement on embedded systems.

We make reductions in code size by analyzing all relevant files, and discarding not only unnecessary classes, but also unnecessary methods and fields. This leads to generation of highly optimized C code, which compiles into an executable with a small footprint.

4.3.1 Analysis

We use the Soot framework to create a *Method Call Graph* of the application. This is a graph with methods as the nodes, and calls from one method to another as directed edges.

At first glance, it seems that the transitive closure⁴ of the main method should represent all methods that can be called. However, this is not so, because the first time the field or method of a class is referenced, its *class initialization method* [3][4] is also invoked, and this can reference other methods or fields in turn.

4. The transitive closure of $G = (V,E)$ is a graph $G^+ = (V,E^+)$ such that for all v,w in V there is an edge (v,w) in E^+ if and only if there is a non-null path from v to w in G . In this case, the transitive closure of the main method refers to all methods which for which the call graph contains a path to them from the main method.

The method call graph contains an edge from each method to every possible target of method calls in it. The number of such targets can be large for polymorphic method calls. A more sophisticated analysis can trim the method call graph by removing some of the edges corresponding to polymorphic invocations.

We use *Variable Type Analysis* (VTA) [22][23][24] to perform this trimming of the call graph. This analysis computes the possible runtime types of each variable using a reaching type analysis, and uses this information to remove spurious edges.

The method call graph contains information about which methods can possibly be called by each method. Information on required fields is obtained by analysis of each method body.

4.3.2 Computing the Set of Required Entities

From the analysis mentioned above, the set of all possible required classes, methods and fields (collectively grouped as *entities*) can be computed statically. We use a set of rules to determine which classes are required.

1. A set of compulsory entities is always required. This includes the `System.initializeSystemClass()` method, and all methods and fields of the `java.lang.Object` class.
2. The main method of the main class to be compiled is required.

3. If a method m is required, the following also become required: the class declaring m , all methods that may possibly be called by m , all fields accessed in the body of m , the classes of all local variables and arguments of m , the classes corresponding to all exceptions that may be caught or thrown by m , and the method corresponding to m in all required subclasses of the class declaring m .
4. If a field f is required, the following also become required: the class declaring f , the class corresponding to the type of f if f is a reference type (not a primitive type) and the field corresponding to f in all required subclasses of the class declaring it.
5. If a class c is required, the following also become required: all superclasses of c , the class initialization method of c , and the instance initialization method of c .

Interfaces are treated as classes. A simple worklist-based algorithm can be used to add to the set of required entities until no additional entities can be found by application of these rules. Together, rules 3, 4 and 5 encapsulate all possible dependencies and references between entities, making the set of required entities self-contained.

4.3.3 Pruning and Code Generation

The algorithm described above performs a form of interprocedural dead-code elimination. The Code Generator generates code only for required entities. Not only does this remove classes that are never used, but also methods that are never

called and fields that are never referenced. The code size reduction thus achieved leads to executables with code footprints within the levels acceptable for embedded systems.

4.3.4 Compilation Time

In the pruning algorithm, all methods called by a given method in the call graph are needed. We observed that setting up the call graph and computing all targets of a given method is the most computation-intensive part of the algorithm. It took an average of 150 seconds⁵ to perform this computation. The total code generation time was then ~156 seconds.

We reduced this overhead by computing a full call graph for all Java library classes, and storing a disk file called the *invoke cache* containing the names of the targets of each method in these classes. Since all classes here are standard Java library classes, the information in this file changes only with the Java version number (currently 1.4.1). This process takes 150 seconds.

At the time of compilation, a *partial call graph* is set up, consisting only of methods in classes that are not Java library classes. Methods of Java library classes are treated as terminal nodes in this graph, since their targets are already known from the invoke cache. The methods called by a non-library method are its targets

5. All times were measured on a 1.5GHz Intel Pentium 4 workstation with 1GB RAM running Windows 2000.

in the call graph, and the methods called by a library method are its targets read from the invoke cache.

The invoke cache is stored once and loaded every time the compiler is called. Since it does not have to be re-computed each time, it significantly lowers the code-generation time. The code generation time we measured with the invoke cache enabled is 6-8 seconds for each program.

5. Limitations

In a C-based compilation strategy, most (but not all) Java features can be implemented as-is, with little deviation from the behavior described in the Java Language Specifications [3]. This section describes Java features that we found to be difficult or inelegant to implement in our C-based, static Java compilation framework.

5.1 Dynamic Loading

Loading, as defined in the Java Language Specification [3], refers to the process of finding the binary form of a class or interface type with a particular name, perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed from source code by a compiler, and constructing, from that binary form, a `Class` object to represent the class or interface.

Linking is the process of taking a binary form of a class or interface type and combining it into the runtime state of the Java virtual machine, so that it can be executed. A class or interface type is always loaded before it is linked.

Modern Operating Systems, such as Solaris, Linux, AIX and all recent windows versions, support *dynamic loading*. They allow executables to link to libraries which are loaded only at runtime. However, most embedded systems do not

support such advanced features. In our design, we did not assume the presence of a system capable of dynamic linking/loading, and thus this feature of Java cannot be supported.

5.2 Reflection

Reflection (the `java.lang.reflect` package) is the ability to discover information about the fields, methods and constructors of loaded classes and to dynamically invoke them. Reflection is useful for very generic programs such as database browsers, visual code editors or web components such as Java Beans.

However, reflection is not supported by C/C++, and hence our scheme does not fully support it. C programs, ubiquitous in embedded systems today, do not utilize reflection. Therefore, we do not consider this to be a major restriction.

5.3 Security

An executable generated with a C-based static compilation strategy will run as a user task on the underlying platform. Therefore, applications that rely on a JVM as a buffer between them and the platform for security cannot be guaranteed to run correctly.

Also, the compiler assumes that the Java bytecode provided to it as input is correct. It does not include a bytecode verifier.

5.4 Threads

A *thread* is a single sequential flow of control within a program. Java allows programs to have multiple threads. An underlying platform-dependant native thread library facilitates threads and is often provided through system calls.

Our current implementation does not support threads, since thread libraries are platform dependant and we favored portability while making design decisions. Future releases may support threads and care has been taken to avoid programming that would preclude multithreading in future versions. There are no theoretical reasons why Java threads cannot be implemented in a C-based compilation framework⁶, and the main issue is mapping Java thread operations to operations on the underlying native threads. Java-to-C compilers such as Toba [14] have successfully implemented Java threads on a Solaris platform.

5.5 Portability

The generated C code may not be fully portable, even if it is strictly ANSI-compliant C. This occurs when different platforms implement various functions slightly differently, have library files in different locations, or have different C compilers. In our experience, retargetability was not particularly challenging, because the compiler framework can easily generate different C code depending

6. JVMs frequently use native thread libraries directly.

on the platform. A small number of code generation rules translated into differences in C code at a large number of sites. So while porting the generated C code by hand would have been challenging, it is extremely straightforward when code generation is automated.

In addition, some native or user-defined methods may need to be modified for a new target. This is also true for the garbage collection library. This needs to be done in any Java implementation, so our strategy does not seem to have any additional portability considerations.

6. Performance Studies

6.1 Benchmarks

To measure floating-point and arithmetic performance, we used the Java version of the **Linpack** benchmark. Linpack is a collection of subroutines that analyze and solve linear equations and linear least-squares problems. The benchmark solves a dense 500 x 500 system of linear equations with one right-hand side, $Ax = b$. The matrix is generated randomly and the right-hand side is constructed so the solution has all components equal to one. The method of solution is based on Gaussian elimination with partial pivoting. The benchmark score is a number indicative of the speed at which the system can execute floating point operations.

The Embedded CaffeineMark⁷ benchmark suite uses 6 tests to measure various aspects of Java performance. The score for each test is a number proportional to the number of times the test was executed divided by the execution time.

The following is a brief description of what each CaffeineMark test does:

- **Sieve:** The classic sieve of Eratosthenes finds prime numbers.

7. Pendragon Software's CaffeineMark(tm) ver. 3.0 was used. The test was performed without independent verification by Pendragon Software and Pendragon Software makes no representations or warranties as to the result of the test. CaffeineMark is a trademark of Pendragon Software.

- **Loop:** Uses sorting and sequence generation to measure compiler optimization of loops.
- **Logic:** Tests the speed with which the virtual machine executes decision-making instructions.
- **Method:** Executes recursive function calls to see how well the VM handles method calls.
- **String:** Performs basic string manipulations.
- **Float:** Simulates a 3D rotation of objects around a point.

In addition to these, we also estimate code size by compiling additional programs that perform extensive tests of specific functionality (HashSets, LinkedLists etc.).

6.2 Methodology

We obtained the benchmark scores for interpreted Java by using Sun standard JVM with the `-Xint` flag. Sun Microsystems' JVM (with default flags) was used as an example of a JIT-enabled Virtual Machine. GNU *gcj* was used as an example of a standard Java-to-native-code compiler. *gcj* was used with the flags `-fno-bounds-check -fno-store-check -static -s -O2`. These options gave both the smallest and the fastest static native code. Turning on additional optimization (upto `-O99`) did not lead to a significant impact on either performance or code size. The C code generated by the Java-to-C compiler was compiled with `gcc` using the flags `-O2 -static -static-libgcc -s -Wall -`

pedantic, with bounds-checking turned off. These flags ensure ANSI C compliance, perform only basic optimizations and generate a static executable.

The tests were performed on a 1.5GHz Pentium 4 running Cygwin on Windows 2000, since this was a platform which allowed us to run the Java Virtual Machine, *gcj*, *gcc*, and the Java-to-C compiler. All benchmark scores shown are the average of 20 runs.

The Java-to-C compiler is highly retargetable. C code generation has been extensively tested for Cygwin on a Windows platform, a Sparc Ultra 5 running Solaris 5.7, and a TMS320C6711 DSP platform. The latter is an 8-way VLIW, floating-point DSP running at 200MHz. It provides an example of a target embedded system on which it has not been possible to run Java code so far, but on which we were able to run Java applications using a Java-to-C compilation strategy. The DSP's C compiler serves as the back end for compilation of C code to native code. Generated code ran correctly on the system, and we are now working on porting the garbage collection library to the platform.

6.3 Results

6.3.1 Performance

The performance of various benchmarks across a number of possible execution strategies is shown in [1]

We compare the performance of a JVM running interpreted Java (no JIT), Java on a JIT-enabled JVM, Java-to-native-code conversion with the *gcj* front-end to the GNU compiler suite, and of using C as intermediate language for final compilation to native code.

We see that interpreted Java runs an order of magnitude slower than any other strategy. This is because a JVM that is running Java without JIT compilation incurs a recurring overhead of converting bytecode to machine code. This overhead is reduced by JVMs using a Just-In-Time compilation strategy, in which machine code for a method is generated from its bytecode when the method is invoked for the first time. *gcj* performs faster than JIT-compiled Java (slower on *String* and *Sieve*, but faster on the other benchmarks). The Java-to-C compilation strategy performs the fastest on all benchmarks except *String*. This can readily be remedied by providing user-defined code for common string operations, but we omitted such hand-optimization because that would no longer provide a useful comparison.

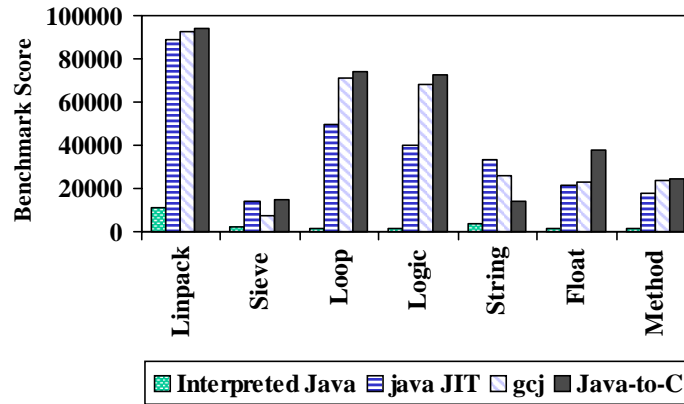


Figure 1. Performance for various benchmarks.

TABLE 4. Percentage improvement in performance of Java-to-C strategy over any other approach. The table shows the amount by which the performance improvement of a Java-to-C implementation over the highest-performing of the other strategies.

	% improvement in performance over next-best strategy
Linpack	1.73% (over gcj)
Sieve	5.85% (over Java JIT)
Loop	4.35% (over gcj)
Logic	5.94% (over gcj)
String	-57.1 (JIT scores higher than Java-to-C)
Float	64.84% (over gcj)
Method	2.2% (over gcj)

6.3.2 Code Size

Java class files are remarkably compact compared to executables. This is because a bytecode instruction is only 1 byte long, whereas a machine instruction can be 2-8 bytes long, depending upon the word size. Typical class files are smaller than a few kilobytes in size. However, a class file requires a JVM and a

class library in order to run. The virtual machine and the core Java classes alone total around 20 megabytes on both Windows and Solaris.

Extensively pared-down Virtual Machines and class libraries for embedded systems can be much smaller. Sun Microsystems' K Virtual Machine and Waba-soft's Waba both have Virtual machine and class library sizes that are around 100 kilobytes. However, these have very basic functionality, such as minimal library classes and limited arithmetic support. Java is no longer as portable on these, and applications need to be written with the target VM in mind.

It is interesting to compare these with the sizes of stand-alone executables generated with *gcj* and the Java-to-C compiler. Both of these enable much richer functionality Java to be implemented.

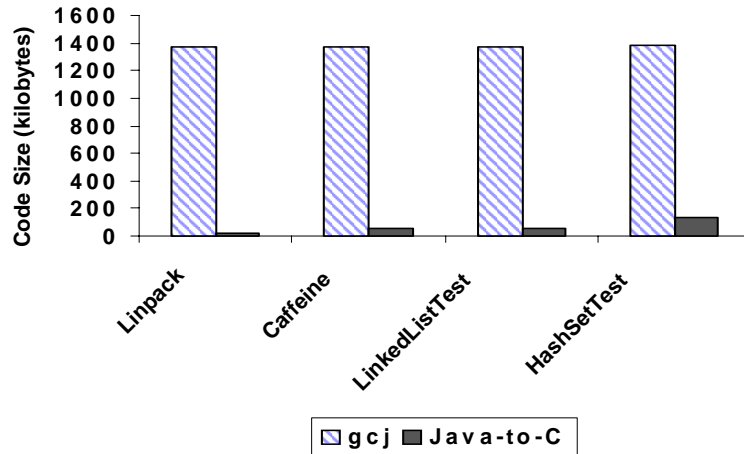


Figure 2. Sizes of generated executables for various applications (Kilobytes).

TABLE 5. Sizes of generated executables for various applications. (Kilobytes)

	gcj	Java-to-C	Ratio of code sizes
Linpack	1371	23	59.6
CaffeineMark	1378	60	23.0
LinkedListTest	1378	52	26.5
HashSetTest	1379	135	10.2
Average			27.3

We see that gcj-generated executables are all around 1.3MB, regardless of the application. This is because the lack of a pruning algorithm causes a very large part of the library to be linked⁸. On the other hand, the Java-to-C approach generates optimized and pruned C code for *all* required classes, effectively building a custom library for each application. This accounts for the significantly lower code size of

8. The size of libgcj.a is 10MB.

the order of 10-100 kilobytes, which is small enough for low-end embedded systems.

7. Conclusion and Future Work

7.1 Contributions

We have created a retargetable Java-to-C compilation framework that provides performance comparable to a JIT-enabled JVM or a direct Java-to-native code compiler. However, it uses an advanced code-pruning algorithm to generate executables over 25 times smaller, on average, than those generated with a best-of-class Java-to-native code compiler. This allows use of Java on resource-constrained systems. We also observed that such a C-based compilation strategy imposes minimal restrictions in terms of Java functionality.

7.2 Future Work

There are a large number of optimizations that we have not yet applied to the generated code. We intend to explore the impact of further optimizations, such as static resolution of provably monomorphic references, and usage of data-flow analysis in the pruning algorithm.

Preliminary studies comparing the performance of the generated C code with hand-coded C code have been encouraging. We intend to measure the performance/code size cost involved in writing programs in Java and generating C code, as opposed to directly coding in C. It would also be interesting to perform perfor-

mance/size characterizations of this framework using a class library specifically targeted for embedded systems.

8. Appendices

8.1 Example of Generated Code

Java code for a simple test class:

```
public class PrintStreamTest{

    public static void main(String[] args) {
        System.out.println(1); // print integer
        System.out.println(2.012); // print float

        String string = new String("Fear the Turtle!!!");
        System.out.println(string); // print string
        Object object = string;
        System.out.println(object);
        // print string cast as object.

        System.out.println(true); // print boolean
        System.out.println(false);
    }
}
```

Corresponding C code:

```
/* Function that implements Method <PrintStreamTest: void
main(java.lang.String[])> */
void f01840038560_main(iA1_i1195259493_String Lr0)
{
    /* Declarations for local variables. */
    i806420721_PrintStream L_r1;
    i1195259493_String Lr2;
    i1195259493_String Lr3;
    i806420721_PrintStream L_r4;
    i1195259493_String L_r5;
    i806420721_PrintStream L_r6;
    i806420721_PrintStream L_r7;
    i806420721_PrintStream L_r8;
    i806420721_PrintStream L_r9;

    /* Initializations for local variables. */
    L_r1 = NULL;
    Lr2 = NULL;
}
```

```

Lr3 = NULL;
L_r4 = NULL;
L_r5 = NULL;
L_r6 = NULL;
L_r7 = NULL;
L_r8 = NULL;
L_r9 = NULL;

L_r1 = (i806420721_PrintStream)Vil199917187_System.class-
vars.f0857384033_out;/* $r1 = java.lang.System.out */
L_r1->class->meth-
ods.m01763166785_println((i806420721_PrintStream/* actual cast
*/)L_r1, (long) 1);/* $r1.println(1) */
// Print integer.

L_r4 = (i806420721_PrintStream)Vil199917187_System.class-
vars.f0857384033_out;/* $r4 = java.lang.System.out */
L_r4->class->meth-
ods.m837144755_println((i806420721_PrintStream/* actual cast
*/)L_r4, (double) ((double)2.012));/* $r4.println(2.012) */
// rint float

L_r5 = (i1195259493_String)( malloc(sizeof(struct
i1195259493_String)));
L_r5->class = &Vil195259493_String;/* $r5 = new
java.lang.String */
L_r5->class->methods.m0914853318__init_((i1195259493_String/*
actual cast */)L_r5, (i1195259493_String) charArrayToString("Fear
the Turtle!!!"));/* specialinvoke $r5.<init>("Fear the Turtle!!!")
*/
Lr2 = (i1195259493_String)L_r5;/* r2 = $r5 */
L_r6 = (i806420721_PrintStream)Vil199917187_System.class-
vars.f0857384033_out;/* $r6 = java.lang.System.out */
L_r6->class->meth-
ods.m193796831_println((i806420721_PrintStream/* actual cast
*/)L_r6, (i1195259493_String) Lr2);/* $r6.println(r2) */
// Print String.

Lr3 = (i1195259493_String)Lr2;/* r3 = r2 */
L_r7 = (i806420721_PrintStream)Vil199917187_System.class-
vars.f0857384033_out;/* $r7 = java.lang.System.out */
L_r7->class->meth-
ods.m415907185_println((i806420721_PrintStream/* actual cast
*/)L_r7, (i1063877011_Object) Lr3);/* $r7.println(r3) */
// Print String cast as Object.

```

```

    L_r8 = (i806420721_PrintStream)Vil199917187_System.class-
vars.f0857384033_out;/* $r8 = java.lang.System.out */
    L_r8->class->meth-
ods.m828280358_println((i806420721_PrintStream/* actual cast
*/)L_r8, (short) 1);/* $r8.println(1) */
    L_r9 = (i806420721_PrintStream)Vil199917187_System.class-
vars.f0857384033_out;/* $r9 = java.lang.System.out */
    L_r9->class->meth-
ods.m828280358_println((i806420721_PrintStream/* actual cast
*/)L_r9, (short) 0);/* $r9.println(0) */
// Print boolean.

    return ;/* return */
} /* Function that implements Method <PrintStreamTest: void
main(java.lang.String[])> */

```


References

- [1] Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, *Prentice Hall, Inc.*, 1988.
- [2] GNU's Not Unix!, <http://www.gnu.org/>
- [3] James Gosling, Bill Joy, and Guy Steele, “The Java Language Specification”, *Addison-Wesley*, 1996.
- [4] Tim Lindholm, and Frank Yellin, “The Java Virtual Machine Specification”, *Addison-Wesley*, 1996.
- [5] Sun Microsystems, “Java 2 Platform MicroEdition (J2ME) Technology for Creating Mobile Devices”, *Sun Microsystems white paper*, 2000.
- [6] Sun Microsystems, “The CLDC HotSpot™ Implementation Virtual Machine”, *Sun Microsystems white paper*, 2002.
- [7] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler, “TurboJ V1.1.2, Ahead-of-time Java compiler”, *The Open Group Research Institute*, Grenoble, France, 1997-1999.
- [8] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert and Xavier Spengler, “TurboJ, a Java Bytecode-to-Native Compiler”, cite-seer.nj.nec.com/269359.html.
- [9] G. Muller and U. Schultz, “Harissa: A hybrid approach to Java execution”, *IEEE Software*, pages 44-- 51, March 1999.
- [10] Gilles Muller, Bárbara Moura, Fabrice Bellard and Charles Consel, “Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code”, *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, 1997.
- [11] J. Dean, G. DeFouw, D. Grove, V. Livinov, and C. Chambers. “Vortex: An optimizing compiler for object-oriented languages” *ACM SIGPLAN Notices*, 31(10):83-- 100, Oct. 1996.
- [12] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu, “Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results”, *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996

- [13] “Guide to GNU `gcj`”, <http://gcc.gnu.org/java/index.html>.
- [14] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, “Toba: Java For Applications -- A Way Ahead of Time (WAT) Compiler”, *Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS)*, USENIX Association Press, 4153; 1997.
- [15] Raja Vallee-Rai, “Soot: A Java Bytecode Optimization Framework”, *Master’s Thesis*, McGill University, July 2000.
- [16] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. “Soot - a Java bytecode optimization framework”. In *Proceedings of CASCON*, 1999.
- [17] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. “Optimizing Java bytecode using the Soot framework: Is it feasible?”, *International Conference on Compiler Construction*, LNCS 1781, 2000.
- [18] Raja Vallee-Rai and Laurie J. Hendren, “Jimple: Simplifying Java Bytecode for Analyses and Transformations”, *Technical Report, Sable Group, McGill University*, July 1998.
- [19] Paul R. Wilson. “Uniprocessor garbage collection techniques”. In *Proc of International Workshop on Memory Management in the Springer-Verlag Lecture Notes in Computer Science series.*, St. Malo, France, September 1992.
- [20] Boehm, H., and M. Weiser, “Garbage Collection in an Uncooperative Environment”, *Software Practice & Experience*, September 1988, pp. 807-820.
- [21] Boehm, H., “Space Efficient Conservative Garbage Collection”, *Proceedings of the ACM SIGPLAN ’93 Conference on Programming Language Design and Implementation*, *SIGPLAN Notices* 28, 6, (June 1993), pp. 197-206.
- [22] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin, “Practical Virtual Method Call Resolution for Java”, *OOPSLA*, 2000.
- [23] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallee-Rai, Patrick Lam, and Etienne Gagnon. “Practical Virtual Method Call Resolution for Java”. *Sable Technical Report 1999-2*, Sable Research Group, McGill University, April 1999.
- [24] Vijay Sundaresan. “Practical Techniques For Virtual Call Resolution In Java”, *Master’s thesis, McGill University*, Montreal, Canada, Sep. 1999.
- [25] WabaSoft, “What is Waba?”, <http://www.wabasoft.com/products.shtml>, 2003.

- [26] “SuperWaba, THE Java VM for PDAs”, <http://www.superwaba.com.br>
- [27] Sun Microsystems, “The Java Language, An Overview”, <http://java.sun.com/docs/overviews/java/java-overview-1.html>
- [28] Sun Microsystems, “The Source for Java Technology”.
- [29] David Bacon, Stephen Fink, and David Grove, “Space and Time-Efficient Implementation of the Java Object Model”, *European Conference on Object-Oriented Programming (ECOOP 2002)*, Malaga Spain, June 10-14, 2002.
- [30] B. Alpern et. al., “The Jalapeño Virtual Machine”, *IBM System Journal*, Vol 39, No 1, February 2000.
- [31] Sadaf Mumtaz and Naveed Ahmad, “Architecture of Kaffe”, <http://wiki.cs.uiuc.edu/cs427/Kaffe+Architecture+Project+Site>, 2002.
- [32] Andreas Krall, “Efficient {JavaVM} Just-in-Time Compilation”, *International Conference on Parallel Architectures and Compilation Techniques*, Paris, 1998.
- [33] A.-R. Adl-Tabatabai, M. Ciernak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. “Fast, effective code generation in a just-in-time Java compiler”, *Conference on Programming Language Design and Implementation*, volume 33(6) of SIGPLAN, Montreal, 1998. ACM.
- [34] Bjarne Stroustrup, “The C++ Programming Language”, Third Edition, *Addison-Wesley*, 1997.
- [35] Bjarne Stroustrup, “The Design and Evolution of C++”, *Addison-Wesley*, 1996.
- [36] Niclaus Wirth and Kathy Jensen, “PASCAL - User Manual and Report. ISO Pascal Standard”, *Springer-Verlag*, 1974.
- [37] Roedy Green, “Java & Internet Glossary”, <http://mindprod.com/jgloss/jgloss.html>, 2003.

