Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems

Ankush Varma and Shuvra S. Bhattacharyya
University of Maryland, CollegePark
{ankush,ssb}@eng.umd.edu

Abstract

The Java programming language is acheiving greater acceptance in high-end embedded systems such as cellphones and PDAs. However, current embedded implementations of Java impose tight constraints on functionality, while requiring significant storage space. In addition, they require that a JVM be ported to each such platform.

We demonstrate the first Java-to-C compilation strategy that is suitable for a wide range of embedded systems, thereby enabling broad use of Java on embedded platforms. This strategy removes many of the constraints on functionality and reduces code size without sacrificing performance. The compilation framework described is easily retargetable, and is also applicable to bare-bones embedded systems with no operating system or JVM.

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

1. Introduction

The Java [1] programming language is a strongly typed, general-purpose, concurrent, class-based, object-oriented language. While it has a number of features that set it apart from other languages such as C, C++ or Pascal, one of its key characteristics is its portability.

The "write once, run anywhere" portability of Java is achieved by compiling the source code into a *class file* format. A class file is a byte stream that contains the definition of a single class or interface. This includes class information and *bytecode*. Bytecode is object code that is processed by a program, referred to as the *Java Virtual Machine* (JVM) [2], rather than by the hardware processor directly.

This portability is based on the implicit assumption that a full-featured JVM exists for all target platforms. In reality, many embedded implementations of Java do not support all Java features, and use an extremely restricted class library instead of the Java standard library. This requires applications to be written with a particular implementation in mind, and leads to significant loss of portability.

We propose a strategy that is suitable for a wide range of embedded systems. We compile Java code into platform-specific C code, which is then converted to an executable by a C compiler. The JVM overhead of dynamic bytecode-to-machine code conversion is thus completely eliminated. Static compilation also allows more aggressive optimization techniques, which lead to lower code size and higher speed. However, much of the functionality is preserved, and the source code of most Java applications need not be modified to run them on embedded platforms. Standard Java classes and utilities can be supported to a high degree.

2. Related Work

Various alternative Java implementations have been explored recently. `TurboJ` [4] speeds up execution by compiling bytecode to native code ahead of time, but using a JVM for some functions. `Harissa` [5] generates C code from Java, but uses a JVM for some functionality. The JVM allows Java to be fully supported, at the cost of increased code size.

`Jove` [22] is a native compiler targeted at large server/workstation programs. It creates executables that are aggressively optimized for speed, not for code size, and thus generates relatively large executables [23].

`gcj` [8] provides a 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. Earlier native-code compilers for Java include `Vortex` [6] and `Caffeine` [7]. Commercial Compilers such as Excelsior's JET [24] are also available, with generated code sizes similar to those for gcj [25].

Sun's KVM (K Virtual Machine) [3] is typical of embedded implementations of Java. It provides reasonable performance and small code size by tightly restricting functionality. It does not support floating-point data types, reflection or object finalization methods, and it places some limitations on threads. It supports a minimal class library and has a code size (the combined size of the Virtual Machine and class library) of ~100KB. The small code size constraint does not allow JIT-compilation. WabaSoft's Waba Virtual Machine [20] 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 existing programs need to be rewritten for Waba. The code size of a Waba implementation is ~100KB. Both of these have versions that provide additional functionality by increasing code size to 300-500KB.

`Toba` [9] is an elegantly-structured Java-to-C compiler. Unfortunately, it is built around a large runtime library and does not attempt to meet the code size constraints of small embedded systems.

We build on some of the concepts introduced in [9] further, introducing features and optimizations to create a viable embedded implementation. We show that a C-based optimized compilation strategy can meet the size constraints inherent in embedded systems and provide performance comparable to the best of other

1530-1591/04 $20.00 (c) 2004 IEEE

implementations without sacrificing Java functionality. The richly-featured Java libraries can be used instead of stripped-down versions.

3. Limitations

A C-based static compilation policy removes many of the restrictions imposed by stripped-down embedded JVMs. In particular, it allows developers to use any data types or standard library classes. The restrictions imposed by such a strategy are:

- Dynamic Loading and Reflection are not supported, simply because the underlying embedded systems do not support these.
- The generated executable runs as a user process, so applications that rely on a JVM as a buffer between them and the platform for security cannot be guaranteed to run correctly.
- Our current implementation does not support threads. Thread libraries are platform dependant and we favored portability while making design decisions. There are no theoretical reasons why Java threads cannot be implemented in a C-based compilation framework, and compilers such as Toba [9] have successfully implemented Java threads in C on a Solaris platform.

4. Runtime Data Structures

The Java object model provides a rich set of features to describe object types and operations. We provide data structures that provide 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 [9].

4.1 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, the name of a Java entity 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. 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 and all classes to share a global namespace.

4.2 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`. These 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 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 1.

The method table contains 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.

TABLE 1. 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 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.

4.3 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 2.

```
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 2. C equivalents of Java references

Reference	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.

4.4 Arrays

The Java programming language allows dynamically-created arrays. 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. Multi-dimensional arrays are treated as arrays of arrays. C functions and macros were written to implement functionality for array initialization and array access

5. 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 [10][11][12], a sophisticated Java bytecode analysis and optimization framework. Soot allows bytecode to be transformed into *Jimple* [13], 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 and a header file containing various declarations and type definitions. A makefile tailored to the target system is also generated for allowing the platform-specific C compiler to create an executable.

5.1 Interfaces

Type polymorphism arising from class-subclass relationships is easily resolved via the structure of the method table 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 [1]. Calls to methods defined in interfaces may also be polymorphic, i.e. there is a set of possible targets of the method call, and a single statically known target may not exist. 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 runtime table-lookup.

5.2 Exception Handling

Exceptions in Java involve 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. If an exception cannot be handled within a procedure, the call stack is unwound until a method is reached which can handle the exception.

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*). This is incremented 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. The appropriate handler for each exception is resolved via a table-lookup.

5.3 Native Methods and User-Defined Code

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.

In addition, we also allow *user-defined code* for non-native methods. This allows hand-optimization of critical code, or easy use of specialized I/O and 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.

5.4 Garbage Collection

Java implements *automatic garbage collection*. Objects no longer in use are destroyed and memory freed up without any explicit programming directive. The choice of garbage collection strategy used is left to the implementation.

Our implementation uses the Boehm-Demers-Weiser conservative garbage collector [15][16]. A conservative collector checks all allocated memory for potential pointers, and traces the transitive closure of all memory reachable from these pointers. It does not require type information, and thus memory management is transparent to the programmer.

6. 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 execution[1]. All 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

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

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 unneccessarily.

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 [9] and `gcj` [8]. This simplifies compilation and linking, but 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 large reductions in code size by analyzing all relevant files, and discarding not only unnecessary classes, but also unneccessary methods and fields. This leads to generation of highly optimized C code, which compiles into an executable with a small footprint.

6.1 Analysis

We use the Soot framework to create a *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 closure of the methods in the main class 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* [1][2] is also invoked, and this can reference other methods or fields in turn.

The method call graph also contains an edge from a 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) [17][18][19] to perform this call graph trimming. This analysis computes the possible runtime types of each variable using a reaching type analysis, and uses this information to remove spurious edges.

6.2 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 statically computed. 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, all methods and fields of the `java.lang.Object` class (since it is the global superclass) and the main method of the main class to be compiled.
2. 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*.
3. If a field *f* is required, the following also become required: the class declaring *f*, the class corresponding to the type of *f* (if any) and the field corresponding to *f* in all required subclasses of the class declaring it.
4. 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 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 2, 3 and 4 encapsulate all possible dependencies between entities. This makes the set of required entities self-contained.

6.3 Pruning and Code Generation

The algorithm described above performs a form of inter-procedural 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.

7. Performance Studies

7.1 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 CaffeineMark**[1] 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.
- 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 evaluate method invocation efficiency.
- String: Performs basic string manipulations.

1. 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.

- 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.).

7.2 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. This platform allows 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.

Our 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 TMSC320C6711 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.

7.3 Results

7.3.1 Performance

The performance of various benchmarks across a number of possible execution strategies is shown in Figure 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 the first time a method is invoked. *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 best 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.

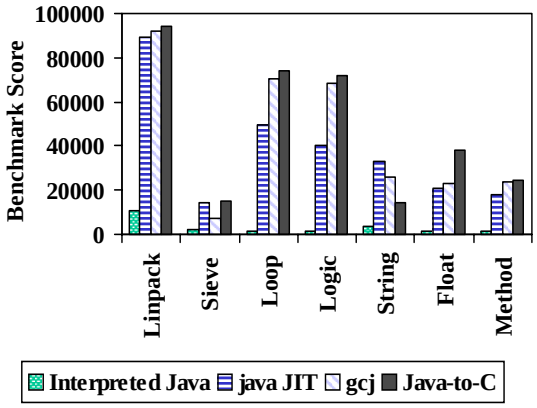

Figure 1. Performance for various benchmarks. A higher score indicates better performance.

TABLE 3. 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)

7.3.2 Code Size

Typical Java 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 Wabasoft'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. Note that comparison of our Java-to-C approach with a stripped-down JVM is not an apples-to-apples comparison because our approach achieves much greater functionality. We allow direct use of standard Java library classes and data types throughout these benchmarks, and in most applications. This is not uniformly permitted by lightweight Java implementations.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 in Java to be implemented. 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 linked[1]. On the other hand, our 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 the order of 10-100 kilobytes, which is small enough for low-end embedded systems.

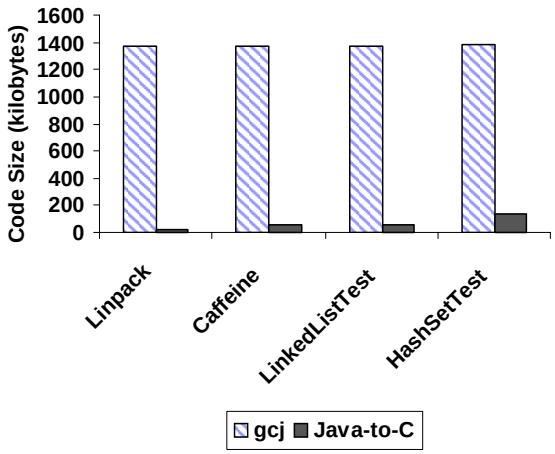

Figure 2. Sizes of generated executables for various applications.

TABLE 4. 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

8. Conclusion and Future Work

We have demonstrated that a C-based static compilation strategy is viable as an embedded implementation for Java.We have described the first Java implementation that is suitable across a wide range of embedded systems.

The performance achieved was comparable to a JIT-enabled JVM while using advanced code-pruning techniques to obtain code size over 25 times smaller than that with a best-of-class Java-to-native-code compiler.

This approach is most promising because, as compared to typical embedded virtual machines, we see that a C-based compilation strategy can remove many of the restrictions on functionality without an accompanying increase in code size.

Preliminary studies comparing the generated C code with hand-coded C code have been encouraging. We intend to measure the performance/code size of generated code, as opposed to hand-coded C. It would also be interesting to perform performance/size characterizations of this framework using a class library specifically targeted for embedded systems.

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

9. Acknowledgements

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

10. References

[1] James Gosling, Bill Joy, and Guy Steele, "The Java Language Specification", *Addison-Wesley*, 1996.
[2] Tim Lindholm, and Frank Yellin, "The Java Virtual Machine Specification", *Addison-Wesley*, 1996.
[3] Sun Microsystems, "Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices", *Sun Microsystems white paper*, 2000.
[4] M. Weiss et. al, "TurboJ V1.1.2, Ahead-of-time Java compiler", *The Open Group Research Institute*, Grenoble, France, 1997-1999.
[5] G. Muller and U. Schultz, "Harissa: A hybrid approach to Java execution", *IEEE Software, pages 44-- 51*, March 1999.
[6] J. Dean et. al, "Vortex: An optimizing compiler for object-oriented languages" *ACM SIGPLAN Notices, 31(10):83-- 100*, Oct. 1996.
[7] Hsieh et. al., "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results", *MICRO*,1996.
[8] "Guide to GNU `gcj`", http://gcc.gnu.org/java/index.html.
[9] T. A. Proebsting et. al., "Toba: Java For Applications -- A Way Ahead of Time (WAT) Compiler", *COOTS*, 1997.
[10] Raja Vallee-Rai, "Soot: A Java Bytecode Optimization Framework", *Master's Thesis*, McGill University, July 2000.
[11] Raja Vallee-Rai et. al, "Soot - a Java bytecode optimization framework", *CASCON*, 1999.
[12] R. Vallee-Rai et. al., "Optimizing Java bytecode using the Soot framework: Is it feasible?", *International Conference on Compiler Construction*, LNCS 1781, 2000.
[13] Raja Vallee-Rai and Laurie J. Hendren, "Jimple: Simplifying Java Bytecode for Analyses and Transformations", *Technical Report, Sable Group, McGill University*, July 1998.
[14] 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.
[15] Boehm, H., and M. Weiser, "Garbage Collection in an Uncooperative Environment", *Software Practice & Experience*, September 1988, pp. 807-820.
[16] Boehm, H., "Space Efficient Conservative Garbage Collection", *PLDI*, 1993.
[17] Vijay Sundaresan et. al., "Practical Virtual Method Call Resolution for Java", *OOPSLA*, 2000.
[18] Vijay Sundaresan et. al.,"Practical Virtual Method Call Resolution for Java". *Sable Technical Report 1999-2*, McGill University, April 1999.
[19] Vijay Sundaresan. "Practical Techniques For Virtual Call Resolution In Java", *Master's thesis, McGill University*, Montreal, Canada, Sep. 1999.
[20] *What is Waba?*, http://www.wabasoft.com/products.shtml
[21] *SuperWaba, THE Java VM for PDAs* http://www.superwaba.com.br
[22] *Jove:Optimizing Native Compiler for Java technology*, http://www.instantiations.com/jove/product/literature.htm.
[23] *Jove Technical FAQ*, http://www.instantiations.com/jove/product/Docs/FAQ.html.
[24] *Excelsior JET - Developer's Perspective*, http://www.excelsior-usa.com/jetdev.html
[25] *Excelsior JET Installation Package Size*, http://www.excelsior-usa.com/jetbenchsize14.html