©1999 IEEE, final version published in IEEE Software March 1999 issue

Harissa: A Hybrid Approach to Java Execution

Gilles Muller and Ulrik Pagh Schultz
{muller,ups}@irisa.fr
Compose Group, IRISA/INRIA*

January 1999

Abstract

Java provides portability and safety but falls short on efficiency. To resolve this problem, the authors developed Harissa, an execution environment that offers efficiency without sacrificing portability or dynamic class loading.

1 Introduction

The Java language is increasingly accepted as the solution to designing safe programs, with an application spectrum ranging from Web services to operating system components. Java's success is partly due to its basic execution model, which relies on the interpretation of a highly portable, object-based virtual machine. However, the well-known tradeoff of Java's portability is its inefficiency in interpreting code.

Several solutions to this tradeoff have been proposed, including just-in-time (JIT) and offline byte code compilers. JIT systems compile byte code at run time on demand. This approach avoids the overhead of compiling unused code, but inhibits aggressive optimizations because high overhead for compilation is unacceptable. Typical offline compilers compile code before the program is run, and thus do not impose bounds on compilation time; optimizing analyses can be run as needed. However, many Java applications dynamically load classes (byte code) at run time. This limits the usability of traditional offline compilers.

We have developed a hybrid approach, embodied in Harissa, an offline compilation system that fully supports dynamic byte code loading. Harissa provides an optimizing byte code-to-C compiler and an interpreter integrated into the run-time library. A compiled program can thus dynamically load and interpret classes. Our work has several other important aspects. Harissa's compiler

- replaces stack management with variables and virtual method calls with simple procedure calls;
- generates faster, optimized code for single-threaded programs; and
- produces C code that executes more efficiently than any alternative implementation strategy.

To evaluate Harissa, we ran an extensive study comparing existing alternatives for executing Java programs. Among other things, our study shows that for frequently used programs, offline compilation is always better than JIT compilation.

We offer an overview of our earlier work [9] and describe Harissa version 3.1, which implements additional global program optimizations, threads, and graphics using the Biss AWT library.[8] The source code of Harissa (distributed under the GNU Public License) and pre-compiled binaries are available from the Harissa homepage at `http://www.irisa.fr/compose/harissa`.

*Contact author: Gilles Muller. Address: Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France. Phone: (+33)299847410, fax: (+33)299847171.

2 Improving Java Execution

Several strategies have been proposed to optimize Java program execution, several of which are relevant to Harissa's design.

2.1 Execution strategy

Strategies for optimizing Java execution range from aggressive compilation schemes to specific hardware processors. Each scheme has advantages and drawbacks.

- Source Java compilers. Compiling source code into native code is the most common way of compiling a language. However, Java programs are distributed in byte code, so the source code is usually unavailable. Thus, the end-user cannot use a source compiler to optimize such programs for a specific platform.

- Byte code compilers. Byte code compilers take byte code, rather than source code, as input. Java byte code contains nearly the same amount of information as source code, to allow classes to be dynamically loaded at run time. As a result, byte code compilers can produce code that is as efficient as that generated by source code compilers. The drawback is that compilation must be completed offline, before program execution.

- Just-in-time compilers. A JIT compiler compiles code at execution time, and only when needed. The compiler compiles a method the first time it's called, pausing the execution. The drawback here is that, unlike the offline approach, optimization becomes an overhead during execution. JIT compilers are highly platform-dependent; they generate native machine code for a specific processor at run time.

Other techniques have also been proposed, including hardware support and dynamic compilation. A Java processor is a dedicated processor that implements the Java Virtual Machine and directly executes the Java byte code. However, because such processors are not widely available, we only consider approaches that do not require specific hardware. As we discuss below, dynamic compilation is a promising approach that has yet to be demonstrated in the context of Java.

2.2 Applications and environment

Although Java was originally designed for programming embedded applications, it has spread to many domains. To choose the appropriate execution scheme, we must consider many factors, such as the frequency of code reuse and target machines' heterogeneity.

At least three common situations must be considered:

- Applets. These components can undergo frequent changes from one load to another on a single client. Here, a JIT compiler is most appropriate.

- Large, platform-independent software. Large programs use Java technology because it's machine independent. Java tools, such as compilers and disassemblers, are examples of such programs, as are Java servlets—programs that provide server functionality. Such programs change infrequently and are often heavily used. It's therefore advisable to keep a local, optimized version of the offline-compiled code. However, this requires version management.

- Platform-dependent software. For platform-dependent software, such as operating system components and embedded applications, Java provides safety via static typing and a guaranteed absence of stray pointers. Because platform-dependent applications change infrequently, it's best to optimize the final code for the target system, unless size constraints make an interpreter necessary.

Some statically configured tools, such the standard `javadoc` tool, require dynamic class loading. For such applications, the best bet is to combine the binary code with an interpreter or a JIT compiler, which allows dynamic loading of new or revised features.

2.3 Target output

There are two types of offline byte code compilers: native and non-native. Native compilers produce directly executable code, while non-native compilers produce code in an intermediate language.

A native compiler has two advantages: compilation speed and a code-generationn back end dedicated to the source language. However, a native compiler is not portable, and implementing it requires extensive knowledge of the target processor.

Non-native compilers are more flexible and offer competitive performance. In particular, using C as an intermediate language permits reuse of existing compiler technology: There are very good C compilers available for back-end use, and C compilers are available for all machines. Thus, you don't have to address subtle differences between a processor and its successors. Finally, C makes compiler development safer, quicker, and in some ways simpler since optimizations can be done on the C code.

For these reasons, we opted for a non-native offline byte code compiler for Java that generates C programs. Section 5 explores other compiler options that are either on the market or in development.

3 Implementation

We designed Harissa with statically configured applications, such as the `javac` compiler, in mind. We based Harissa's garbage collector on the Boehm-Demers-Weiser conservative garbage collector [1], and its thread model on the Posix thread library. Harissa implements JDK 1.0.2. Because Harissa's compiler reads Java class files and produces ANSI C programs, it is easily portable. Current ports include SunOS, Solaris, and Linux.

3.1 Global analysis and optimizations

A Java program consists of a set of classes that contain static references to each other. Most often, these classes interoperate using virtual method calls; dispatching is based on the program's run-time configuration. With Harissa, we load the entire set of classes at compile time, letting it examine each class to detect code for starting threads or dynamic class loading. If none exist, further optimizations are possible.

For input, Harissa's compiler takes a class containing a main method and generates a makefile, a main C file, and a C source file for each program class. To determine the set of classes that depend on the initial class, Harissa recursively analyzes the byte code to find all classes referenced by the main class. This analysis makes the compilation task trivial for the user, since Harissa automatically compiles all program classes into the binary program. Harissa also supports separate compilation, as we describe below.

Object-oriented programming encourages both code factoring and differential programming, resulting in smaller procedures and more method invocations. To optimize method invocation, Harissa integrates class-hierarchy analysis and an intraprocedural static class analysis [6]. These analyses can be used to optimize virtual method invocations by statically computing the set of possible callees at each call site. Thus, virtual calls that always invoke the same method can be replaced with a direct procedure call.

In Harissa-generated C code, synchronized blocks and methods are protected by monitors and exception handlers. However, programs that do not use multithreading do not need synchronization. We assume that, in general, applications are either single-threaded (command-line programs such as `javac`) or multithreaded (graphical applications or server applications); the latter rarely return to a single thread of control. Given a completely loaded program, Harissa statically detects whether threads can be created. (We assume, conservatively, that dynamically loading programs are multithreaded.) For single-threaded program code, Harissa does not need to generate monitor instructions or the associated exception regions.

3.2 Class representation

Harissa treats each class as a separate unit. A set of compiled classes can either be statically linked with a main C file to form an application or be placed in a separate shared library. When a class is first refer-

enced at run time, the Harissa run-time system first searches for the class in the statically linked application, then in any dynamically linked compiled-class libraries, and, finally, in the file system for uncompiled classes.

When using separate compilation, Harissa checks to see if a target class is in the library before generating it and linking it into the application. This improves compilation time and reduces the size of the generated code.

A class implementation includes a virtual table of function pointers that stores the addresses of method-implementing procedures. All virtual calls are made through this virtual table. Each pointer in the table can point to either the compiled class's C procedure (method), a super class's C procedure, the run-time library's C native function, or a stub procedure. Stub procedures are associated with specific Java methods; they let compiled code call interpreted code, and vice versa. A stub for calling from compiled code to the interpreter pushes arguments onto the interpreter stack, calls the interpreter, and returns the element to the top of the stack. Similarly, a stub for calling from the interpreter to compiled code pops arguments from the interpreter stack, calls the compiled method, and pushes the compiled method's return value onto the stack.

As with calls through the virtual table, virtual calls through an interface are implemented using a two-dimensional vector of function pointers for each class.

3.3 Method compilation

To generate a method's C code, the compiler transforms byte code instructions into C statements. Harissa implements exceptions using the C `longjmp` library function and method calls using ordinary (possibly indirect) C function calls. All other control structures are expressed by `goto` statements.

In Java byte code, each exception has an associated region; such regions are either disjoint or nested, but cannot overlap. In the generated C code, entering an exception region pushes the corresponding exception handler onto the exception stack, and exiting the region pops the exception handler off the exception stack. When an exception is thrown, a loop traverses the exception stack, using `longjmp` to transfer control to each exception handler until an appropriate handler is found.

To translate byte code instructions into C, the stack analysis statically evaluates the stack operations. It does this by calculating the stack pointer's value and all byte code instruction types. Most Java byte code instructions have an explicitly associated type, except those that control the stack. But because of the constraints enforced by the Java byte code verifier [7], a stack instruction can only have one type signature at each program point. Thus, we easily infer the stack-operation types. This lets us replace the stack with explicitly typed local variables.

Figure 1 shows the compilation steps using Java code for a method computing the power function. First, the Java source code is compiled into byte code—in this case, we used the `javac` compiler—then Harissa translates the byte code into C code. Variables with an `s` prefix are generated from stack operations; variables with a `v` prefix are user-defined. Assigning an `s`-variable corresponds to pushing a value on the stack; using it corresponds to popping a value off the stack. Once in C code, Harissa statically evaluates the stack and performs constant propagation and copy propagation on the resulting variables. In this example, si0 was the only variable Harissa's copy propagation did not eliminate. Such remaining variables can usually be eliminated by an optimizing C compiler.

3.4 Design perspectives

In Harissa, programs created by multiple components, such as Java Beans, can be either statically configured or dynamically linked to dynamically selected features. With static configuration, the entire program can be globally optimized for a more efficient end result. With dynamic linking, each component can be compiled separately and dynamically linked into the program as needed at runtime.

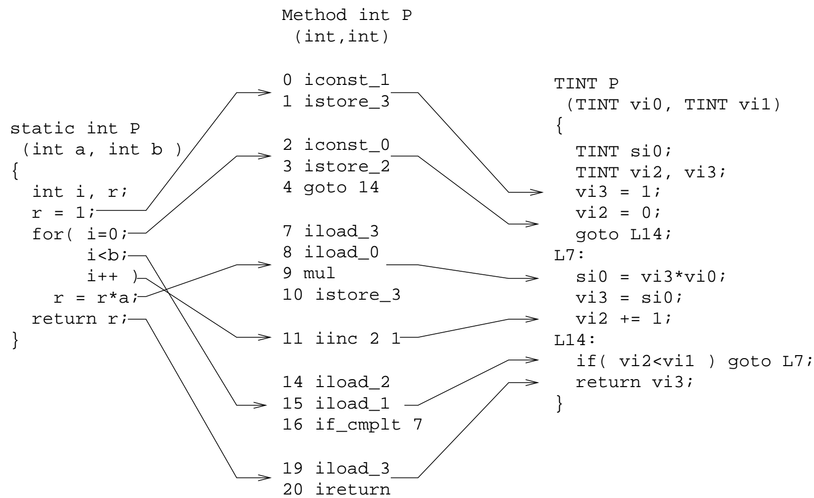

Figure 1: Translation of Java to C via byte code. The code here is for a method computing the power function; we used the `javac` compiler to translate the source code into byte code. Harissa then translates the byte code into C.

4 Benchmarks

In 1998, we evaluated the impact of Harissa's aggressive optimizations and its performance relative to JIT compilers. JIT compiler performance is naturally sensitive to the target architecture since it compiles into native code. To get more representative results, we ran all benchmarks on two different platforms: a Dell 200 MHz Pentium Pro PC and a Sun 300 MHz UltraSparc. The Pentium machine runs Linux 2.0 and Windows NT 4.0; the Sparc machine runs Solaris 2.6. Harissa has not been ported to Windows, so comparisons are made across operating systems, slightly affecting the results.

We compare Harissa with the Toba 1.0.6b byte code compiler Proebsting-al:coots97 Both compilers perform stack elimination, but only Harissa performs virtual call elimination and other aggressive optimizations. We also compared Harissa with the JIT compilers of Microsoft SDK 3.0, Sun JDK 1.2b4 (reference implementation), and Kaffe 0.8.4. The Microsoft SDK JIT compiler is only available for Windows on Pentium. The Sun JDK 1.2b4 JIT compiler for Windows on Pentium is licensed from Symantec. Information about these compiler systems is available from the Harissa homepage.

4.1 Methodology

We ran two different types of benchmarks. First, we ran large benchmarks to compare the expected performance of JIT and offline compilers for real applications that may include I/O. Second, we ran microbenchmarks to evaluate the efficiency of basic operations when using JITs and offline compilers for pure computations (without I/O).

On the Sparc, Harissa and Toba used Sun's commercial C compiler to compile the C code. On the Pentium, Harissa used the Pentium Gnu C Compiler (pgcc) version 2.90.29.

Harissa was invoked with the `-EO3` flag, selecting maximal optimization. This included the inlining op-

timization, which produced a code growth of 50 percent on the Sparc and Pentium. The longest running compilation, `javac` (with full optimization), took 10 minutes on the Pentium, including code generation and compilation.

All comparisons are based on real execution time, which includes waiting for the end of I/O, as this corresponds to what the user observes. The programs we used for the study are available—along with detailed benchmark results—from the Harissa homepage.

4.2 Realistic benchmarks

We used sizeable benchmarks to estimate Harissa's efficiency in a realistic environment. To do this, we presented the execution time of a set of programs doing pure computations, substantial I/O, and a mixture of both.

For pure computation, we used an Othello game, measuring the time spent to solve up to depth 7 on the first move. For the I/O study, we used JHLZip, a file handling application that inserts files into an archive without compressing them. For input, we used the JDK 1.0.2 `classes.zip` file.

We represented large, complex applications using two of Sun's JDK 1.0.2 tools: the `javac` compiler, which performs significant data processing; and the `javadoc` documentation generator, which relies on Java's dynamic capabilities to load byte code during execution. (For this reason, it can't be executed with a pure byte code-to-C compiler, such as Toba.) The performance of these tools depends significantly on their input. To get representative results, we ran them on a set of large Java programs. (The source of these programs is available from the Harissa homepage.) Running `javac` and `javadoc` using Kaffe failed on certain source programs, for reasons we could not determine. However, our limited experiments with Kaffe and the JDK tools indicated that it was the slowest system on this benchmark.

On the real application benchmarks, results were tied to how much work the program did. On the JHLZip application, which executes in a couple of seconds and is dominated by I/O, the offline compilers slightly outperform the JIT systems. This is because offline systems don't have to compile byte code, and any JIT-system optimizations are unlikely to have a major effect on code efficiency.

On the computationally intensive Othello game, compared with JDK 1.2b4, Harissa was 1.15 times faster on the Pentium and 3.9 times faster on the Sparc; compared with MS SDK 3.0, Harissa was 1.2 times faster on the Pentium. On the same benchmark, compared with Toba, Harissa was 2.0 times faster on the Pentium; compared with Kaffe, Harissa was 4.0 times faster on the Pentium and 7.0 times faster on the Sparc. On the complex `javac` and `javadoc` applications, compared with JDK 1.2b4, Harissa was 2.6 times faster on the Pentium and 3.9 times faster on the Sparc, and 1.6 times faster than MS SDK 3.0 on the Pentium. Compared with Toba, Harissa was 1.4 times faster on the Pentium and 2.4 times faster on the Sparc. Figures 2 and 3 summarize our results for the Intel and the Sparc architectures, respectively.

4.3 Micro-benchmarks

We ran micro-benchmark tests using Caffeine 3.0 [12]. Each Caffeine micro-benchmark tested one feature of the Java machine and produced numbers—in CaffeineMarks, wherein higher is faster—that let us directly compare heterogeneous architectures and Java implementations. Among them, we consider those that are included in the embedded version.

We used micro-benchmarks to compare compilation-scheme efficiency in Harissa and JIT compilers. Because the tests looped on the same code, JIT compilers did not lose execution time waiting for a method's compilation. The tests were very limited and were ideal for a JIT compiler. This let us precisely evaluate the relative quality of the basic code produced by Harissa and the JIT compilers.

Our measurements show that Harissa-generated code was superior to JIT code on Sparc and inferior to JIT code on Pentium. The overall score of Harissa's code on Sparc was 5.2 times higher than that of Kaffe and 2.0 times higher than that of JDK 1.2b4. On the Pentium, the overall score of Harissa's code was 3.2 times higher than that of Kaffe, 1.2 times lower than that of MS SDK 3.0, and 1.6 times lower than that

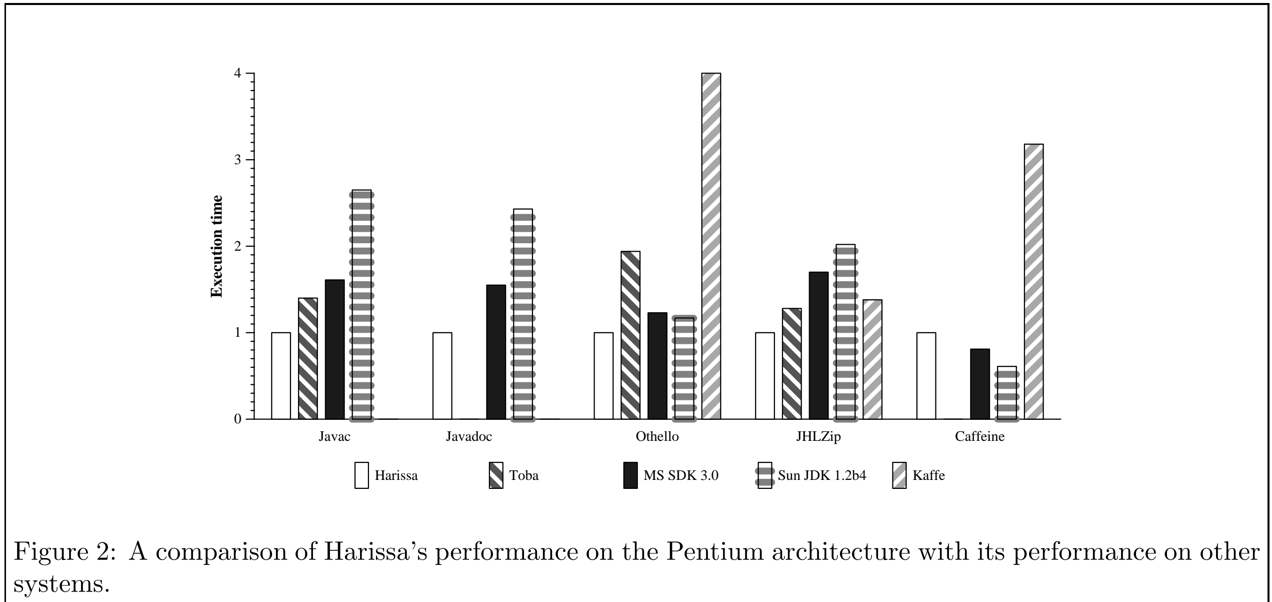

Figure 2: A comparison of Harissa's performance on the Pentium architecture with its performance on other systems.

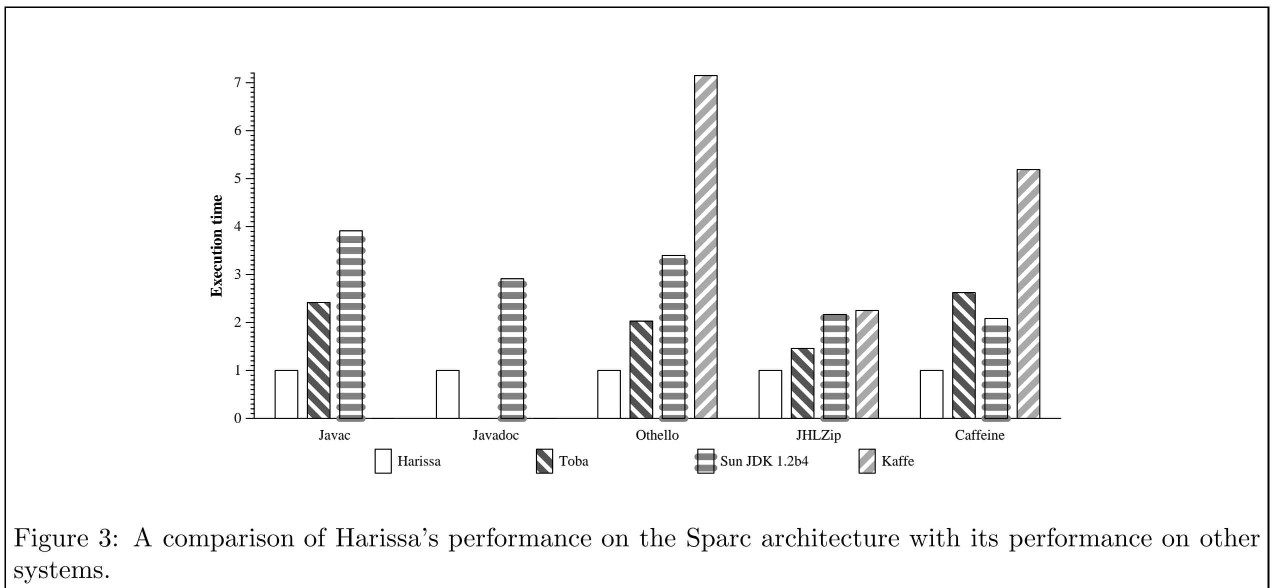

Figure 3: A comparison of Harissa's performance on the Sparc architecture with its performance on other systems.

of JDK 1.2b4.

4.4 Optimizations

Unlike some other Java-to-C compilers, such as Toba, Harissa performs class-hierarchy analysis and single-thread optimization. The impact of the class-hierarchy analysis has been studied for many object-oriented languages, including Java. Among the findings are that such analysis can improve program performance between 23 and 89 percent [6].

On the benchmarked programs, our class-hierarchy analysis implementation let between 14 and 40 percent of the virtual call points be transformed into procedure calls. Replacing virtual calls by direct procedure invocations had little effect, but subsequent inlining increased speed by at least 1.5 times in most cases.

Comparing `javac` compiled with Harissa as a single-threaded program against `javac` compiled as a multithreaded program let us estimate the cost of synchronization primitives. The single-threaded version of `javac` ran from 1.2 to 1.7 times faster than the multithreaded version. The increased speed is primarily due to the generation of simpler, unsynchronized code for the many synchronized methods in the JDK library.

5 Existing Solutions

The performance of Java environments is constantly improving, making it difficult to present a clear picture of this rapidly evolving domain. We present a short overview of the features of the contemporary Java compiler systems that we are aware of, including the emerging strategy of dynamic compilation.

Offline compilers

Several byte-code-to-C compilers have been implemented, including Toba, TowerJ, and TurboJ. (References to these systems are available from the Harissa homepage.)

Toba [11] does a stack-elimination analysis similar to Harissa's, but does not optimize method calls. Unlike Harissa, Toba switches between lightweight and standard monitors dynamically, as the application switches between single-threaded and multithreaded mode. Lightweight monitors still require exception handler protection, but do not perform synchronization.

TowerJ is a commercial system; it implements analysis and optimizations similar to Harissa's and produces standard C code. It is available for a wide range of platforms, taking advantage of the widely available Gnu C Compiler.

TurboJ dynamically links with an existing implementation of the Java run-time library, trusting the existing implementation to handle garbage collection, multi-threading, I/O, and graphics. It also relies on the Java run-time library to automatically interface compiled code with dynamically loaded code.

Vortex [2] also generates C code, but at a very low level. Vortex has front-ends for Java, Cecil, C++, and Modula-3. It implements a wide range of advanced optimizations and has served as a test-bed for examining the effects of combining these optimizations. Unfortunately, the Java front-end is still very primitive, making its use impractical.

Many commercial, native off-line compiler systems have been implemented for Windows. Very little information is available on the compilation strategies these systems offer.

Dynamic compilation

Essentially, a dynamic compiler works like a JIT, except that it only compiles and optimizes those parts of the code that execute often. Performing optimizations at run time is an advantage in that more information is available when the program is running, including computed values and dynamically gathered profile information. This strategy is the basis of Sun's HotSpot compiler.

Many of the optimizations performed by compilers such as HotSpot can be performed at compile time. The compiler can perform costly analysis and optimizations at compile time, and then augment these dynamically as more information becomes available at run time. Although this increases time spent compiling the program, it requires less work at run time, resulting in a faster program.

Because HotSpot has yet to be released, we have not yet evaluated its performance relative to JITs and offline compilers.

6 Conclusion

In our benchmarks, code produced by Harissa's optimized offline compiler ran real programs faster than a JIT system, particularly on the Sparc. This is primarily because binary code for modern RISC processors is difficult to optimize, requiring analyses that are hard to run on the fly.

However, the JIT and offline strategies can be complementary. Michael Plezbert and Ron Cytron [10] showed that it is possible to compile code by running the unoptimized code while another process aggressively compiles in the background using offline techniques. Once the optimized code is available, unoptimized code is replaced with optimized code. Because Harissa already mixes byte code interpretation and binary execution, this continuous compilation scheme can be easily incorporated.

Although the code Harissa generates is already fast, the micro-benchmarks show opportunities for improvement. We are now investigating how to improve code quality for Pentium processors using different C compilers. We also plan to eliminate some type and bound checks, as our close examination of the C code showed that most such checks could be evaluated statically through a simple intraprocedural analysis. To improve dynamically loaded code execution, we are experimenting with an automatically generated JIT for Pentium and Sparc architectures. This JIT is generated from the existing Harissa interpreter using the Tempo partial evaluator's run-time specialization system [4, 3], and will be included in a future version of Harissa.

Finally, we are investigating the use of Harissa to partially evaluate Java pprograms in conjunction with the specialization class framework [13] and the Tempo partial evaluator. Partial evaluation of Harissa-generated code will let us further increase compiled program efficiency.

Acknowledgments: We thank the Harissa users for their interest and helpful comments. We are also grateful to Lars Raeder Clausen, Olivier Danvy, Julia Lawall, Renaud Marlet, and the anonymous referees for their comments. This work is supported in part by the Brittany Council. The first prototype of Harissa was designed in part by Fabrice Bellard.

Gilles Muller is a research associate scientist at IRISA/INRIA-Rennes. His work focuses on designing adaptive programs and operating systems, including adaptable system components for drivers and active networks. He is developing an approach to improving efficiency in generic programs by adapting them to particular contexts, and is also working on the design of Tempo, a specialized tool for C programs. Gilles Muller received his PhD from the University of Rennes in 1988.

Ulrik Pagh Schultz is a PhD student in the Compose group of IRISA, University of Rennes. His research interests include partial evaluation, object-oriented programming, and block-structural transformations. He is investigating the use of partial evaluation (automatic program specialization) to perform highly aggressive optimizations on Java programs, and is currently using the Harissa compiler in conjunction with Tempo, a partial evaluator for C, to perform transformations on Harissa's intermediate C code. Ulrik Schultz received his MS at the Department of Computer Science (DAIMI) of the University of Aarhus in Denmark.

References

[1] H.J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

[2] C. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report 96-06-02, Department of Computer Science, University of Washington, June 1996.

[3] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[4] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[5] *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, Portland (Oregon), USA, June 1997. Usenix.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[8] P.C. Mehlitz. The BISS java framework. URL: http://www.biss-net.com/biss-awt.html, 1997.

[9] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In COOTS97 [5], pages 1–20.

[10] M.P. Plezbert and R.K. Cytron. Does "just in time" = "better late than never"? In *Proceedings of POPL'97*, pages 120–131, Paris (France), January 1997.

[11] 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. In COOTS97 [5], pages 41–53.

[12] Pendragon Software. Caffeinemark 3.0. URL: http://www.webfayre.com/pendragon/cm3-/index.html, 1997.

[13] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.