The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code

Gilles Muller, Barbara Moura, Fabrice Bellard, Charles Consel
IRISA / INRIA-University of Rennes
Campus de Beaulieu
F-35042 Rennes Cedex - France

Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code

Gilles Muller, Bárbara Moura, Fabrice Bellard, Charles Consel
IRISA / INRIA-University of Rennes
Campus de Beaulieu
F-35042 Rennes Cedex - France
Fax: +33 2 99 84 71 71
Harissa@irisa.fr
http://www.irisa.fr/compose/harissa/harissa.html

Abstract

The Java language provides a promising solution to the design of safe programs, with an application spectrum ranging from Web services to operating system components. The well-known tradeoff of Java's portability is the inefficiency of its basic execution model, which relies on the interpretation of an object-based virtual machine. Many solutions have been proposed to overcome this problem, such as just-in-time (JIT) and off-line bytecode compilers. However, most compilers trade efficiency for either portability or the ability to dynamically load bytecode.

In this paper, we present an approach which reconciles portability and efficiency, and preserves the ability to dynamically load bytecode. We have designed and implemented an efficient environment for the execution of Java programs, named Harissa[1]. Harissa permits the mixing of compiled and interpreted methods. Harissa's compiler translates Java bytecode to C, incorporating aggressive optimizations such as virtual-method call optimization based on the Class Hierarchy Analysis. To evaluate the performance of Harissa, we have conducted an extensive experimental study aimed at comparing the various existing alternatives to execute Java programs. The C code produced by Harissa's compiler is more efficient than all other alternative ways of executing Java programs (that were available to us): it is up to 140 times faster than the JDK interpreter, up to 13 times faster than the Softway Guava JIT, and 30% faster than the Toba bytecode to C compiler.

Keywords: Java, C, Bytecode, Off-line compilers, JIT compilers

1 Introduction

The Java language [1, 2] provides a promising solution to the design of safe programs, with an application spectrum ranging from Web services to operating system components [3]. The success of Java is partly due to the fact that its basic execution model relies on the interpretation of an object-based virtual machine which is highly portable. However, the well-known tradeoff of Java's portability is the inefficiency of interpretation. Several solutions have been proposed to overcome this

[1]This research was supported in part by the Britany Council.

problem, such as just-in-time (JIT) [4, 5, 6, 7] and off-line [8, 9] bytecode compilers.

Just-in-time systems compile code to native form at runtime on demand. This approach avoids the overhead of compiling unused code, and eliminates the gap between compile time and execution time. Compiling during program execution, however, inhibits aggressive optimizations because compilation must only incur a small overhead. This is particularly important in the case of modern RISC processors for which complex analyses are required to achieve the best result. Moreover, the quality of the generated code critically relies on knowledge about the specific features of the target processor. Therefore, such compilers are not platform independent and requires a large amount of work to be ported.

Off-line compilers does not impose critical bounds on compilation time; optimizing analyses can be run as needed. They can also be platform independent, if they generate as output an intermediate language. However, in the context of Java, many applications dynamically load classes (i.e., bytecode) at runtime that limits applicability of pure off-line compilers.

In this paper, we present an approach that reconciles portability and efficiency, and preserves the ability to dynamically load bytecode. We have designed and implemented an efficient environment for the execution of Java programs, named Harissa[2]. Harissa provides a bytecode compiler and an interpreter integrated in the runtime library. Thus, a compiled program is still able to dynamically load classes and to interpret them. Harissa's compiler translates Java bytecode to C and furthermore incorporates aggressive optimizations.

To evaluate Harissa, we have conducted an extensive experimental study aimed at comparing the various existing alternatives to execute Java programs. The contributions of our work are as follows.

- The C code produced by Harissa's compiler is more efficient than all other alternative ways of executing Java programs (that were available to us): on the Caffeine Micro-benchmarks [10], it is 5 to 140 times faster than JDK 1.0.2 interpreter, 2 to 13 times faster than the Softway Guava JIT [6] and on average 20% faster than the Microsoft JIT compiler. On real application benchmarks, such as the *Javac* compiler, it is 5 times faster than the JDK interpreter and 30% faster than the Toba [9] bytecode to C compiler.

- The compiler statically evaluates the stack by abstractly interpreting the bytecode and replaces stack management with variables. This optimization suppresses one of the main sources of inefficiency in Java.

- The compilation process does virtual-method call optimization based on the class hierarchy analysis (CHA) [11, 12]. On the set of programs used in our benchmarks, this analysis permit the replacement of up to 40% of virtual methods calls by simple procedure calls.

- In contrast to existing off-line compilers, the runtime system of Harissa includes an interpreter that preserves the ability of an application to dynamically load bytecode.

- Finally, we discuss the benefits and limitations of off-line compilation *vs* JIT compilation. Based on our experimental study, we show that, for frequently used programs, it is always more advantageous to use off-line compilation rather than JIT compilation.

[2] Harissa was previously named Salsa.

The paper is organized as follows: Section 2 describes existing approaches for optimizing the execution of Java programs. Section 3 presents Harissa. Section 4 presents related work in class hierarchy analysis and existing bytecode compilers. Section 5 analyzes the performance of the code generated by Harissa's compiler on micro-benchmarks and real benchmarks, such as the *Javac* compiler and the *Javadoc* documentation generator. Section 6 concludes by describing future work and comparing JIT and off-line compilers.

2 How to Improve Java Execution

Several strategies have been presented to optimize execution of Java programs. They range from aggressive compilation schemes to specific hardware processors. Advantages and drawbacks of these schemes are the following:

- **Native Java compilers** - Compiling source code into native code is the most common way of compiling a language. But this approach is contrary to the Java philosophy since all the advantages of having an platform independent language disappear. For instance, the source code of Java programs is often not available. However, this strategy may be useful to obtain very efficient target binaries for very specific environments. This approach is implemented in the Vortex project [12].

- **Bytecode compilers** - Unlike native compiler, bytecode compilers take bytecode as input. One of the interesting characteristics of Java is that the bytecode contains nearly the same amount of information as the source itself. It has even been shown by Ford [13] and by Vliet [14] that it is possible to decompile the bytecode of a program and produce a Java source program similar to the original one. This is mostly due to the fact that the signature of the classes in the program must be kept in the bytecode to allow classes to be dynamically loaded at runtime. The only significant loss of information in the bytecode concerns structured loops, which are transformed into goto statements. Hence, a bytecode compiler can easily be as efficient as a native compiler. There are two types of bytecode compilers: those that generate native code and those that generate an intermediate language, such as C. The advantages of these two approaches are discussed below.

- **Just In Time compilers** - A just-in-time compiler differs from the a "classical" off-line compiler, in that the code is compiled only when needed at execution time. The difference in performance between those approaches is the time that can be spent during execution to perform optimizations. Vendors such as Borland [4], Symantec [5], Softway [6], and others have already released JIT compilers. The basic scheme is to compile a method when it is called for the first time, pausing execution while doing so. Refinement to this approach has been recently described by Plezbert and Cytron [7]. They mix interpretation and JIT compilation by taking advantage of multi-threading (on a multiprocessor)

- **Java Processors** - A Java processor is a dedicated processor that implements the Java Virtual machine and directly executes the Java bytecode. Such processor can be used as the main processor in a dedicated Java machine (workstations, embedded systems) or as a co-processor in a workstation. Sun and other manufacturers are already designing such chips. However, their competitiveness has not yet been proved [15]. Since such processors are not currently available, in this paper we only consider approaches that do not require specific hardware.

When is an Off-line Bytecode Compiler the Right Choice?

Although Java was originally designed for programming embedded applications, it has recently spread to many domains. Therefore, to choose the appropriate execution scheme many factors, such as the frequency of reuse of the same code or the heterogeneity level of the set of target machines, have to be considered. The most frequent situations are the following:

- **Small software components integrated in Web services** - These components can undergo frequent changes from one load to another by the same client. As a result, in this context, a JIT compiler is the most appropriate solution.

- **Platform-independent large software** - Such programs may or may not be related to Web services. Java technology is used because of its machine independence. The Java tools themselves are examples of such programs (e.g., compiler, disassembler, ...). These programs change infrequently and are often used by many users. Therefore, keeping a local, optimized version of the compiled code is advantageous. By comparison to a JIT, that always get the latest version of the software, this approaches requires the management of local optimized versions. This can be implemented by a revision control system that compiles and installs new software versions as they are released, in a automatic and transparent way.

- **Platform-dedicated software** - Examples are operating system components [3] and embedded applications. For these applications, the Java technology provides safety. These applications are characterized by very infrequent changes. Hence, it is advantageous to optimize the final code for the target system.

Finally, it should be noticed that even some statically configured tools, such as *Javadoc*, dynamically choose and load classes at execution time. For these applications, it is thus worthwhile to combine the binary code with an interpreter or a JIT compiler to allow dynamic (over)loading of new features.

Choosing C as a Target Output

As was already stated, there are two types of off-line bytecode compilers: native and non-native. Native compilers produce code that is directly executable, while non-native compilers produce code in an intermediate language.

Designing a native compiler has two advantages: (i) the generated binary code may be more efficient than that resulting from code written in an intermediate language and (ii) compilation is fast since it does not require successive tools. However, this choice has drawbacks: (i) it is not portable and (ii) generation of efficient code requires extensive knowledge of the features of the target processor.

Non-native compilers are more flexible and also achieve competitive performance. In particular, choosing C as an intermediate language permits the reuse of extensive compiler technology that has already been developed. In fact,

- There are very good C compilers.

- C compilers are available for all machines. The developer does not have to address subtle differences that exist between a processor and its successors.

- The development process is safer, quicker, and in some ways simpler since optimizations can be done on the generated C code.

- It is possible to reuse existing, aggressive optimizers such as Suif [16] or partial evaluators for C such as C-mix [17] or Tempo [18, 19].

These reasons led us to develop a non-native off-line compiler for Java bytecode that generates C programs.

3 Overview of Harissa

Harissa is a Java environment that includes a compiler from Java bytecode to C and a Java Virtual Machine integrated in a runtime library. While Harissa is aimed at applications that are statically configured, such as the *Javac* compiler, it is also designed to allow code to be dynamically loaded in an already compiled application. This novel feature is introduced by integrating a bytecode interpreter into the runtime library. Data structures between the Java compiled code and the interpreter are compatible and data allocated by the interpreter do not conflict with data allocated by the compiled code. Harissa is written in C and is designed with the primary goal of providing efficient and flexible execution of Java applications.

Because Harissa is written in C and its compiler generates C code, it is easily portable. In fact, current ports include SunOS, Solaris, Linux, and Dec Alpha. This allows us to compare the effects of optimizations on different architectures.

Because Harissa's compiler produces C programs, various compilers and optimizers can be used. As a result, contrary to JIT compilers, the generated C code does not have to be heavily optimized, since final optimizations are made by the C compiler. Harissa only concentrates on inefficiencies due to the architecture of the Java Virtual Machine: stack and method calls. To do so, several transformations are introduced. First, the stack is statically evaluated away. This analysis is described in section 3.3. Second, virtual method calls are transformed, when possible, into static (i.e., procedure) calls. For these virtual calls, type checks are also eliminated. This is described in section 3.4. Finally, Harissa implements several other optimizations for object-oriented languages such as method inlining, which are not presented.

The following sections describe the system in more detail.

3.1 Compiling a Java Program

Harissa's compiler takes as input a class $\mathcal{C}$ containing a `main` method and generates as output a `makefile`, a `_main.c` file, and a C source file for each class used in the program[3](see Figure 1). To determine the set of classes that depend on the initial class, an analysis is recursively performed on the bytecode to search for all the classes referenced by the main class. Because of the simplicity of this phase, it is omitted in the paper.

Compilation of a method's bytecode into C is organized as follows:

- Step 1 - The bytecode of the method is transformed into an intermediate bytecode representation (IBR). The purpose of this phase is to obtain a simpler and more regular representation. The IBR

[3]The system also supports separate compilation to reduce compilation time and the size of generated code. To do so, the compiler checks if a target class exits in the library before translating it. Since separate compilation conflicts with class hierarchy analysis and method call optimization, it has not been used in our benchmarks.

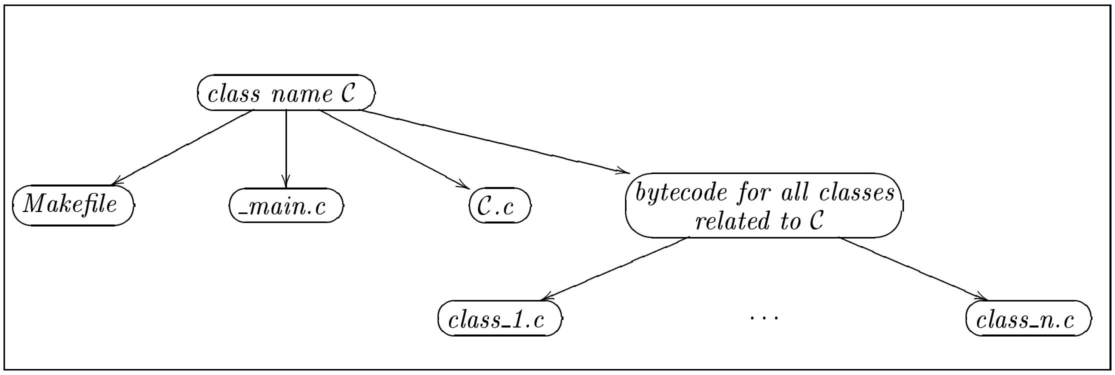

Figure 1: Set of generated C files given an initial class name $\mathcal{C}$

simplifies the implementation of the subsequent passes by making explicit more detailed information than in the original Java bytecode.

- Step 2 - An analysis determines the value of the stack pointer before each instruction and the signature type for instructions that handle the stack. The result of this analysis allow the stack to be statically evaluated.

- Step 3 - A class hierarchy analysis is performed as described in [11]. This analysis permits the implementation of further optimizations on the intermediate representation. These optimization include: method inlining, transformation of virtual method calls into static (non-virtual) ones, and elimination of type checking. Method inlining and conversion of virtual method calls into non-virtual ones are iterated until no opportunities for further optimizations remain.

- Step 4 - This phase aims at eliminating bound checking. Checks are eliminated when it is possible to merge references to the same index as well as when the array bound and the index can be statically determined.[4]

- Step 5 - The final step generates the C code from the IBR. This phase is divided into three phases: (i) generation of goto labels and exception handling, (ii) declaration of local variables for the method, and (iii) translation of each intermediate bytecode instruction into C.

The following sections present our intermediate bytecode representation and the main algorithms that have an impact on performance. That is, the calculation of the stack pointer and the types of the stack instructions, and the transformation of virtual method calls into static procedure calls.

3.2 Intermediate Bytecode Representation

Our intermediate bytecode representation has a simpler and more regular syntax, and contains more detailed information than the original Java bytecode. The main difference is that, in the IBR, the types of the arguments of the instructions that handle the stack are

[4] This phase is not yet implemented in the current version of Harissa, although the code is conceived to include this optimization.

made explicit. This information simplifies subsequent passes.

The data structures defining the IBR are shown in Figure 2. A method contains information about its body and the exceptions that it can raise. Associated with each exception is the program counter of its handler. The `CodeInfo` structure has all the information about each instruction. Fields `in_sig` and `out_sig` represent the instruction's input and output signature, respectively. This explicit representation of signature types eases the subsequent analyses. The analysis described in the next section infers the type of instructions whose type is not explicit in the Java bytecode.

3.3 Calculation of the Stack Pointer and Instruction Signature Types

This analysis statically evaluates the stack by calculating the value of the stack pointer and the types of all the bytecode instructions. Most Java bytecode instructions have their type already associated with them, except those that control the stack. Because of the constraints enforced by the Java bytecode verifier [20], at each program point a stack instruction can have only one type signature. For example, when the instruction DUP is used to duplicate an integer, it can not be used at the same program point to duplicate a double. Thus, we can straightforwardly infer the types of the stack operations.

The analysis of a method via `CalculateSPandTypes` abstractly interprets each instruction with respect to a `Stack` structure (see Figure 2), which contains the current stack pointer value and the type of its items. `AnalyseCode` and `AnalyseExc` interpret the method's body and the code fragments corresponding to the method's exception handlers, respectively. The stack is initially empty. Abstract interpretation of an instruction can modify the contents of the stack. If an instruction I branches to more than one program point, then each branch is interpreted with respect to the stack resulting from abstractly interpreting I. Note that for the specific case of the jump to subroutine instruction (JSR), used to implement exceptions, the stack is assumed to be empty before and after the execution of the instruction. The JSR and RET instructions are considered to have the same control flow as a test instruction and the RETURN instruction, respectively. This approximation is not correct in terms of control flow information but gives correct results for stack type information.

Interpretation of an instruction is as follows: if the type of the instruction is not explicit in the Java bytecode, then the analysis has to infer it. The input signature is inferred from the types on the stack (function `infer_in_sig`). The output signature is inferred by abstractly interpreting the instruction with respect to this input signature (function `infer_out_sig`). Once the signature is known, then the instruction is abstractly interpreted with respect to the stack and its signature, with functions `pop_sig` and `push_sig`. The former checks for type consistency between the input signature and the type of the items it pops off the stack and the latter pushes the instruction's output signature onto the stack.

3.4 Transforming Virtual Calls into Non-Virtual Calls

Object-oriented programming encourages both code factoring and differential programming. This results in smaller procedures and more procedure calls. Procedure calls in an object-oriented language are dynamically dispatched. There are many analyses targeted at optimizing dynamically dispatched message sends. The most common are: intra-procedural static class analysis [11],

```
struct MethodInfo {
 CodeInfo *code;
 ExceptionInfo *einfo;
}

struct CodeInfo {
  char *in_sig, *out_sig;
  char opcode;
  list *instr_branch;
}

structure Stack =
  int sp;
  char *stack_type;
}

struct ExceptionInfo {
   ExceptionInfo *next;
   int handler_pc;
}

AnalyseCode (CodeInfo *code, int pc,
             Stack stk)
{
  CodeInfo instr;

  instr = get_instr (code, pc);
  if (visited? instr)
     return;
  else
     stk = AnalyseInstr (instr, stk);
  for each instr_branch do
  {
     stk' = stk;
     if (instr->opcode == JSR)
       stk' = empty_stk;
     AnalyseCode (code, branch, stk');
  }
}
```

```
CalculateSPandTypes (MethodInfo *minfo) {

  AnalyseCode (minfo->code, 0, empty_stk);
  AnalyseExc (minfo, minfo->code);
}

AnalyseInstr (CodeInfo instr, Stack stk) {
  char  *in_sig, *out_sig;

  in_sig = instr->in_sig;
  out_sig = instr->out_sig;
  if (unknown_sig? instr)
   {
     in_sig = infer_in_sig (stk);
     out_sig = infer_out_sig (in_sig);
   }
  stk = pop_sig (in_sig, stk);
  stk = push_sig (out_sig, stk);
  return stk;
}

AnalyseExc (MethodInfo *minfo, CodeInfo *code) {
  ExceptionInfo *einfo;
  Stack stk;

  einfo = minfo->einfo;
  while (einfo != NULL)
    {
      stk = push_item (REF, empty_stk);
      AnalyseInstr (code, einfo->handler_pc, stk);
      einfo = einfo->next;
    }
  }
```

Figure 2: Inferring instruction's type

class hierarchy analysis (CHA) [11], and profile-guided class receiver prediction [21]. In Harissa, we have opted to integrate a class hierarchy analysis to address this problem.

A class hierarchy analysis is a static analysis that determines a program's complete class inheritance graph (CIG) and the set of methods defined in each class. With the CIG, a specific set of possible classes, given that the receiver is a subclass of the class $\mathcal{C}$, can be statically inferred and messages sent to the method's receiver can be optimized. Further, if there are no overriding methods in subclasses, a message sent to the method's receiver can be replaced with a direct procedure call and possibly inlined. Inlining of a method can trigger other opportunities for converting dynamic method calls into static ones. Hence, these two transformations are iterated.

3.5 Generation of C Code

The generation of the C code for a method is done in three phases. First, the goto labels and exception handlers are generated. Then, the local variables of a method are declared and, finally, each IBR instruction is translated to C.

Generation of goto labels and declaration of local variables are simple and are not discussed here. The treatment of exceptions needs some explanation. To ensure portability, Harissa handles exceptions in a stack-based manner. In the Java bytecode, each exception has a region associated to it. As described in the bytecode verifier documentation [20], different exception regions are either disjoint or nested, but cannot overlap. When translating the intermediate bytecode to C, entering of an exception region pushes the corresponding exception handler onto the stack, and exit of an exception region pops the exception handler off the stack. If a jump or goto instruction leaves an exception region or a set of nested exception regions, the corresponding exception handlers are popped off the stack prior to the jump or goto instruction.

The actual generation of the C code from the intermediate bytecode representation is straightforward. Figure 3-a shows some Java source code for a method computing a power function, Figure 3-b shows the corresponding Java bytecode. Figure 3-c shows the translated C code. In the C code, the stack has been statically evaluated: variable names prefixed with "s" are variables that handle the stack, while variable names prefixed with "v" are user-defined variables. An assignment to an `s`-variable corresponds to pushing a value on the stack. A use of an `s`-variable corresponds to popping a value off the stack. The `s`-variables can be eliminated either by a C compiler or by a C optimizer such as Suif [16]. Figure 3-d shows the optimized code generated by Suif.

3.6 Method Call Implementations

The implementation of a class includes a vector of function pointers that store the addresses of procedure implementing methods. Initialisation of this vector is performed when intantiating the class either at compile-time, by the compiler, or at run-time when dynamically loading byte-code. After initialisation, a pointer may refer either to a C procedure (i.e., method) of the compiled class, to a C procedure of an inherited compiled class, to a C native function of the run-time library, or to a `stubc` procedure. A `stubc` procedure interfaces compiled code with the interpreter: it allocates a stack for the interpreter, pushes arguments, calls the intrepreter's entry-point, and pops the result. `Stubc` procedures are generated by the compiler for each method that might be dynamically overloaded.

Interface calls are implemented using of a two dimensional sparse vector of function pointers for each class. The first dimension

```
static int P(int a,int b)
{
  int i,r;
  r=1;
  for(i=0;i<b;i++) r=r*a;
  return r;
}
```

a: Java source code

```
Method int P(int,int)
   0 iconst_1
   1 istore_3
   2 iconst_0
   3 istore_2
   4 goto 14
   7 iload_3
   8 iload_0
   9 imul
  10 istore_3
  11 iinc 2 1
  14 iload_2
  15 iload_1
  16 if_icmplt 7
  19 iload_3
  20 ireturn
```

b: Java ByteCode

```
TINT P(TINT vi0,TINT vi1)
{
  TINT si1,si0;
  TINT vi2,vi3;
  si0=1;
  vi3=si0;
  si0=0;
  vi2=si0;
  goto L14;
  L7:
  si0=vi3;
  si1=vi0;
  si0*=si1;
  vi3=si0;
  vi2+=1;
  L14:
  si0=vi2;
  si1=vi1;
  if (si0<si1) goto L7;
  si0=vi3;
  return si0;
}
```

c: C generated code

```
extern int P(int vi0, int vi1)
  {
    int vi2;
    int vi3;

    vi3 = 1;
    vi2 = 0;
    goto L14;
  L7:
    vi3 = vi3 * vi0;
    vi2 = vi2 + 1;
  L14:
    if (vi2 < vi1)
        goto L7;
    return vi3;
  }
```

d: Suif optimized code

Figure 3: Compilation of the power method

equals to the total number of interfaces referenced by the program, each interface being assigned an index at compile time. When a class is instantiated, if the class implements a given interface, the corresponding second dimension of the vector is allocated and is initialized with C procedures.

3.7 Current Status and Limitations of Harissa

Harissa is provided in two versions, with and without garbage collection (GC). This allows us to estimate the influence of GC on its performance. The GC version is based on the Boehm-Demers-Weiser conservative garbage collector [22]. The non-GC version relies on malloc, which leads to an increase in swapping and I/O since objects are never deallocated.

At the current time, threads are not implemented. Nevertheless, the system is already conceived to include them and the generated C code contains the necessary calls to synchronization functions. Implementation of synchronization optimizes the single thread case. As long as no additional threads are created, synchronization calls point to a null procedure. Additional threads creation is detected by guards [23] that then plug-in the multi-thread synchronization function.

For efficiency, Harissa produces a target C that relies on some gcc extensions. This is not a major limitation since gcc is available on many platforms. We plan to eliminate this dependency, in order to be able to test vendor C compilers. Finally, there are some native libraries, such as the graphic library, that are not yet supported.

4 Related Work

Other Off-line Compilers

To our knowledge, there are two other compilers from bytecode to C: J2C and Toba[5]. Harissa is the only environment that integrates an interpreter. J2C performs no optimizations when generating C code (i.e., stack evaluation or method call optimization). It is still immature and fails for many applications. Toba does a stack analysis similar to the one included in Harissa and generates C code from which transient variables have been eliminated. However, Toba does not do any method call optimizations. Currently, Toba is slightly more mature than Harissa since it supports threads.

Previous Work in CHA

Compilers for other object-oriented languages have included a CHA to optimize dynamically dispatched calls. In [24], Vortex, an optimizing compiler for object-oriented languages is presented. Vortex differs from Harissa in the following ways. It is a language-independent compiler with front-ends for Java, Cecil, C++, and Modula-3. Vortex takes as input source code. This approach limits its domain of use in the case of Java since source code is often not available. The optimizations it performs range from standard ones, such as constant propagation, dead code elimination, and method inlining, to optimizations specific to object-oriented languages, such as intraprocedural static class analysis, class hierarchy analysis [11], and profile-guided class receiver prediction [21]. The Vortex compiler has been used to study the impact of each of these optimizations alone and in combination. In [11], it is shown that class hierarchy analysis and profile-guided class receiver prediction are complementary transformations:

[5] Harissa and Toba have been developed independently at the same time.

the combination of the two produces a compounding effect.

Fernandez presents an optimizing linker that does class hierarchy analysis of Modula-3 programs [25]. Optimizations and code generation are done at link-time. The problem with this approach is that further optimizations that can result from transforming virtual calls into static procedure calls cannot be done by the compiler. An optimizing source-to-source C++ compiler is presented in [26]. The number of virtual method calls are reduced by performing both type feedback [27] and class hierarchy analysis. Method inlining is done as well. The optimized program is compiled by a native host C++ compiler.

5 Benchmarks

This section analyzes the performance gain that can be expected from an aggressive bytecode compiler. We compare execution of Harissa compiled programs with several industrial JIT compilers, the J2C and Toba bytecode compilers, and the JDK 1.0.2 interpreter.

Performance of JIT compilers is by nature sensitive to the target architecture since they compile into native code. To get more representative results, we have run the benchmarks on two different platforms: a Dell 100Mhz Pentium PC and a Sun 85 Mhz Sparcstation 5 (SS5). On the Pentium, Harissa is compared with the JIT compilers embedded in Netscape 3.0 and Microsoft Internet Explorer 3.0. On the Sparc, Harissa is compared with the Guava JIT compiler from Softway [6].

Three different kinds of benchmarks are presented: micro-benchmarks, which are used to evaluate the efficiency of JIT and off-line compilers for pure computations (without I/O); large benchmarks, which are used to compare JIT and off-line compilers for real applications that include I/O; and finally, benchmarks to evaluate the effectiveness of the CHA for Java applications.

Summary of results

Figure 4 summarizes our results. The micro-benchmark tests are made using Caffeine 2.5 [10]. Each Caffeine micro-benchmark tests one feature of the Java machine. On these tests, Harissa generated code is on average 50 times faster than JDK, 5 times faster than Softway Guava JIT [6] and 50% faster than Microsoft JIT.

On real application benchmarks, results depend mainly on how much pure computation the program does. On applications dominated by I/O, such as JHLZip and JHLUnzip, there is not much difference between off-line and JIT compilers; JDK is only 1.5 slower that Harissa. On applications such as *Javac* and *Javadoc* which rely on a mixed set of computation and I/O, Harissa is 5 times faster than JDK, 3 times faster than Softway Guava JIT and 30% faster than the Toba [9] bytecode compiler. On pure computation programs, such as an Othello game [28], Harissa is 2.6 times faster than Guava, 1.7 faster than Toba and 44 times faster than JDK. Toba results are missing when it was not possible to run it successfully, for reasons described below.

Methodology

Harissa has been configured so that during compilation, only methods with a size smaller than 100 instructions are inlined. The C code generated by Harissa and J2C has been compiled using gcc with the "-O2" option. The gcc version used is 2.7.2 on the Sun and 2.7.0 on the PC/Linux. Toba-generated C code has been compiled using Sun's commercial C compiler with the "-xO4" option.

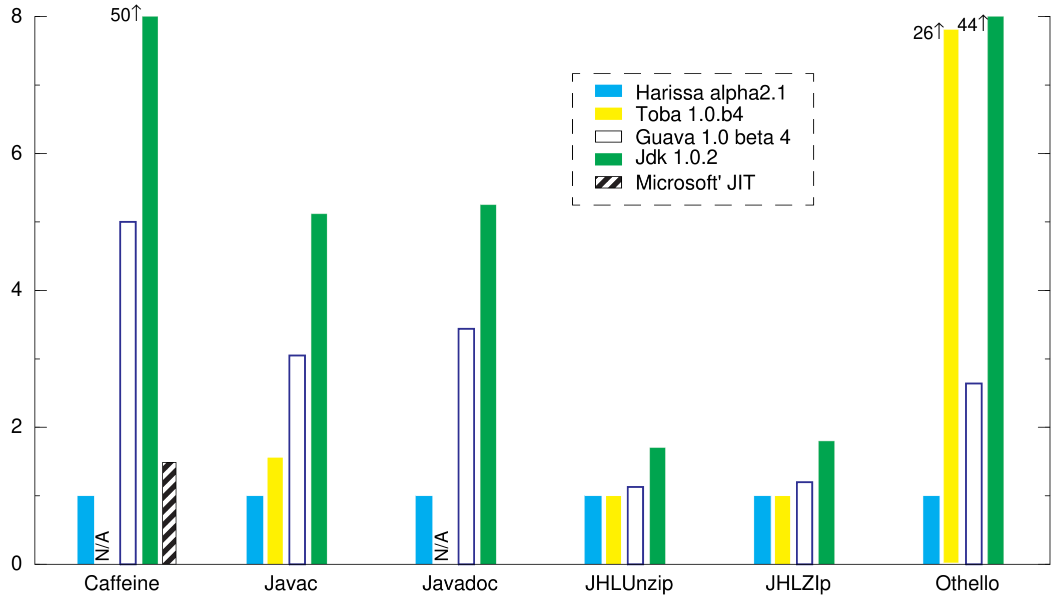

Figure 4: Execution time normalized to Harissa

5.1 Caffeine Micro-benchmarks

The Caffeine micro-benchmarks produce numbers, in CaffeineMarks (higher is faster), that allow one to compare heterogeneous architectures and Java implementations directly. Among them, we consider those that are related to the compilation scheme and that do not rely on graphic computations or the garbage collector:

- *Sieve* calculates prime numbers under 2048;
- *String2* tests string concatenation and search;
- *Logic* executes loops containing decision trees;
- *Loops* runs several types of integer loops;
- *Floating Point* (i.e., FP) simulates the calculations needed to rotate 50 three dimensional points by 90 degrees, 5 degrees at a time;
- *Method* tests how fast the VM performs method calls.

General comments about the results

The results of our evaluation are presented in Table 1 for the SS5 and in Table 2 for the PC. The two rightmost columns present Harissa's results with some further optimizations that are described below. In general, the PC is faster than the Sun. On the SS5, JDK and the interpreter embedded in Netscape achieve similar results, while the code generated by Harissa is 5 to 140 times faster than the JDK interpreter. On the PC, Microsoft's JIT compiler seems to be slightly faster than the Netscape's one, except for the tests *String2* and *FP*, which are twice as fast under Microsoft.

JIT compilers *vs* Harissa

The relevance of the micro-benchmarks when comparing JIT compilers and Harissa is to

	JDK	*Netscape*	*Guava*	*Harissa*		*Harissa+Suif*		*Harissa+Loop*	
	Cm *Interp.*	Cm *Interp.*	Cm *JIT*	Cm	ratio Guava	Cm	ratio Guava	Cm	ratio Guava
Loop	82	101	1657	11479	**6.92**	10170	**6.13**	-	-
Logic	95	116	968	9910	**10.23**	12935	**13.36**	-	-
String2	95	105	480	1390	**2.57**	1390	**2.57**	-	-
Method	82	75	102	460	**4.5**	460	**4.5**	-	-
Sieve	95	100	440	514	**1.16**	910	**2.06**	955	**2.17**
FP	83	92	544	970	**1.78**	1170	**2.15**	1295	**2.38**

Table 1: Comparison between JIT and Harissa on a 85Mhz SS5 (in *CaffeineMarks, Cm*)

measure the efficiency of the compilation scheme. Since the tests loop on the same code, JIT compilers do not loose time during execution waiting for the compilation of a method. Furthermore, with the exception of the *Method* test, no method calls are made. Harissa's inter-procedural optimizations such as CHA and method inlining thus have very little influence on the results. Therefore, these tests permit to evaluate precisely the quality of the code that is produced by JIT compilers.

Our measurements show that the code generated by Harissa'compiler is basically always faster than JIT compilers. Nevertheless, the results are architecture dependent. On the SS5, Harissa is 1.5 (for *Sieve*) to 13 (for *logic*) times faster than the JIT Guava. On the PC, results are more balanced and the difference in performance between Harissa and Microsoft is smaller than between Harissa and Guava, with a maximum of 2.5 times faster. For two tests, *Sieve* and *FP*, Harissa is actually twice as slow.

Improving the performance of the code generated by Harissa

To understand the reasons for the inefficiency of the code generated by Harissa for the tests *Sieve* and *FP*, we have analyzed the assembly code generated by gcc. For the *Sieve* test, it appears that the critical loop is about 20 instructions long. That does not leave much room for possible optimizations.

We have identified two reasons for inefficiency, which are in fact due to limitations of the gcc optimizer. As expected, transient stack variables are eliminated by gcc. But further optimizations resulting from variable and constant propagation are not triggered. For instance, in the *Sieve* test, stack variable elimination transforms a "`divide by i`" into a "`divide by 2`" that could then be efficiently transformed into a shift instruction. To evaluate the impact of this problem, we have used the Suif C optimizer [16] to systematically eliminate these variables using a combination of the "constant/variable propagation" and "dead code elimination" passes. The effect on the PC is dramatic for the *FP* and *Sieve* tests, nearly doubling the performance improvement. On the other tests there is little or no influence, which shows that this situation is not so frequent. On the Sparc, the influence of stack variable elimination is lower than on the PC. This is because the relative cost of processor instructions differs significantly between the Sparc and the Pentium.

A second source of inefficiency is the fact that loops are compiled into bytecode goto instructions. Therefore, gcc does not have all the necessary information to make the best choice regarding caching of temporary results in registers. To determine the consequences of this problem, we have reconstructed loops by hand for the *Sieve* and *FP* benchmarks.

	Microsoft Win 95	*Netscape Win 95*		*Harissa Linux*		*Harissa+Suif Linux*		*Harissa+loop Linux*	
	Cm	Cm	ratio Msoft	Cm	ratio Msoft	Cm	ratio Msoft	Cm	ratio Msoft
Loop	7087	7128	**1.005**	18000	**2.53**	18000	**2.53**	-	-
Logic	2032	1909	**0.93**	1930	**0.94**	2445	**1.20**	-	-
String2	1430	320	**0.22**	1833	**1.28**	1850	**1.29**	-	-
Method	2413	2028	**0.84**	4740	**1.96**	4790	**1.98**	-	-
Sieve	1370	1320	**0.96**	730	**0.53**	1420	**1.03**	1512	**1.1**
FP	2420	1306	**0.53**	1400	**0.57**	2350	**0.97**	2600	**1.07**

Table 2: Comparison between JIT and Harissa on a 100Mhz PC-Pentium (in *CaffeineMarks, Cm*)

On both the Sparc and the PC, there is a performance increase between 5% to 10%. Finally, it should be noted that after performing the optimizations, Harissa's compiled code is about 10% faster than Microsoft's JIT.

5.2 Real-Sized Benchmarks

These benchmarks are used to estimate the efficiency of Harissa in a real environment. To do so, we have evaluated the execution time of a set of programs that either do pure computations, substantial I/O, or a mixture of both. Pure computation programs are represented by an Othello game [28]. File handling applications (e.g., I/O) are represented by JHLZip and JHLUnzip, which insert and extract file from an archive without compression. Mixed computation-I/O programs are represented by two Sun's JDK tools, the *Javac* compiler and the *javadoc* documentation generator, and by *Kawa*, a scheme interpreter [29].

The benchmarks were made in a single-user environment to avoid external interferences. It was not possible to run benchmarks for JIT compilers embedded in the Web browsers for security protection reasons. Performance of tools such *Javac* and *javadoc* depends significantly on their input. To get representative results, we ran them on a set of large Java programs that are available on the net:

- Jas generates bytecode from a scheme based scripting language [30].
- Jax generates tokenizers from regular expressions [30].
- Jell generates a recursive descent parser from from a LL(1) grammar [30].
- Kawa is a scheme interpreter [29].

Comparisons are performed on real execution time, which includes waiting for the end of I/O, since this corresponds to what the user observes. For completeness, we have also detailed user and system CPU time spent during the execution to measure the efficiency of pure computations.

Detailed Javac results

Detailed timing of *Javac* execution are presented in Table 3. In comparison with JDK, Harissa achieves the highest speedup which is greater than 5. Toba is on average 3.3 times faster than JDK, J2C is about 2.5 times faster, and Guava is 1.5 times faster. These results clearly show the benefits of the various optimizations performed in Harissa.

We have also compared Harissa's GC version with the non-GC one. The GC version version is 20% faster than the non-GC one. This due to the fact that never reclaiming objects leads to an increase in swapping, I/O,

	JDK 1.0.2			*Guava 1.0 beta 4*				*J2C*			
	real mn	user cpu sec.	sys cpu sec.	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
Jax	0:44	37.2	2.5	0:29	16.5	4	**1.5**	0:16	7.7	2.6	**2.7**
Jell	0:47	41	2.5	0:30	18.4	4.4	**1.5**	0:17	8.9	3.5	**2.7**
Jas	1:45	69.5	9.0	0:59	37	9	**1.8**	0:41	15.6	7.6	**2.5**
Kawa	3:50	134	24	1:59	58	19	**1.9**	1:41	32	17	**2.3**

	Toba				*Harissa/no GC*				*Harissa/GC*			
	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
Jax	0:12	8	2	**3.6**	0:10	4.7	2.7	**4.4**	0:09	4	1.9	**4.9**
Jell	0:14	9.2	2.9	**3.3**	0:10	5.2	3	**4.7**	0:09	4.7	1.9	**5.2**
Jas	0:35	15	5	**3**	0:25	8.8	5.5	**4.2**	0:20	7.2	3.8	**5.2**
Kawa	1:10	28.7	10.5	**3.2**	0:58	17.6	11.5	**3.9**	0:44	14.5	8	**5.2**

Table 3: Compilation time of several Java programs

and in the amount of address space that has to be allocated by the system to the process.

Detailed Javadoc results

Javadoc is representative of tools that rely on the dynamic capabilities provided by Java to load bytecode during execution. Therefore, it is not possible to execute it with a pure bytecode to C compiler such as Toba or J2C. Although dynamically loaded classes are interpreted, most of the execution time is spent in the compiled code. Thus, Harissa's generated code is on average 5 times faster than JDK and 3 times faster than Guava.

Other Benchmarks

JHLZip and JHLUnzip tools [31] insert and extract files from an archive. The tested version of these tools does not include compression and therefore, execution is dominated by I/Os. Our tests have been done using the JDK 1.0.2 classes.zip file as input. As it could be expected, compilers (off-line and JIT) achieve the same level of performance. Finally, JDK is only 1.5 slower than the compilers.

The tested implementation of Othello game [28] allocates a finite time to the computer player to solve one move. The depth of the search depends on the speed of the generated code. We give the time spent to solve up to depth 5 on the first move.

5.3 CHA Evaluation

The impact of the class hierarchy analysis has been studied for many object-oriented languages, including Java. It has been shown that this analysis can improve program performance between 23% to 89% [11]. Table 6 presents the impact of CHA for the programs we have benchmarked. It shows that our CHA implementation allows between 14% to 40% of the virtual call points to be transformed into procedure calls.

6 Conclusion and Future Work

The contribution of this work is threefold: (i) we have designed a hybrid environment for Java, named Harissa, that permit mixing

	JDK 1.0.2			*Guava 1.0 beta 4*				*Harissa*			
	real mn	user cpu sec.	sys cpu sec.	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
Jax	0:30	24	2.3	0:27	19	3.2	**1.1**	0:05	2.6	0.9	**6**
Jell	0:37	30	3	0:28	21	3.3	**1.3**	0:08	3.7	1	**4.6**
Jas	0:53	28	3.5	0:26	15	3.8	**2**	0:11	3.4	1.6	**4.8**
Kawa (codegen)	0:34	27	2.7	0:20	14	3.5	**1.7**	0:06	3.5	0.9	**5.6**

Table 4: Javadoc execution time

	JDK 1.0.2			*Guava 1.0 beta 4*			
	real mn	user cpu sec.	sys cpu sec.	real mn	user cpu sec.	sys cpu sec.	ratio JDK
JHLUnzip	1:34	21	7.1	1:00	6	8.3	**1.5**
JHLZip	0:34	20	8.11	0:22	4.7	8.7	**1.5**
Othello	22			1.5			**14.6**

	Toba 1.0.b4				*Harissa*			
	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
JHLUnzip	0:56	4	6.6	**1.7**	0:56	3	6	**1.7**
JHLZip	0:20	5.4	7.2	**1.7**	0:18	2.3	7	**1.8**
Othello	0.85			**26**	0.50			**44**

Table 5: Other Benchmarks

	Javac	*Javadoc*	*JHLZip*	*JHLUnzip*	*Othello*	*Kawa*
Number of classes, interfaces, and arrays	280	281	99	100	103	213
Interfaces	10	10	6	6	4	12
Arrays	25	25	13	13	16	24
Methods containing bytecode	1867	1910	703	696	701	1310
Native methods	104	104	89	89	95	93
Average size of a method (in bytes)	63.8	64.8	48	47.9	41.2	45.6
Virtual call points	4734	4333	863	845	642	2334
Number of optimized calls due to CHA	**1828**	**1689**	**155**	**155**	**92**	**690**
percentage of virtual call points	**38%**	**40%**	**18%**	**18%**	**14%**	**25%**

Table 6: Detailed analysis of method and classes

of interpretation with compiled bytecode, (ii) we have designed an aggressive bytecode to C compiler whose generated code is more efficient than other compilers, and (iii) we have measured the relative efficiency of code produced by off-line and JIT compilers.

Tradeoffs Between JIT and Off-line Compilers?

The micro-benchmarks presented in section 5.1 clearly show that an optimized off-line compiler such as Harissa's is faster than a JIT compiler. The gap between JIT and off-line compilers is greater for the SPARC than for the Pentium. This is due to the fact that binary code for modern RISC processors is complex to optimize and requires analyses that are hard to run in the short time allocated to on the fly compilation.

However, the JIT and off-line strategies can be made complementary. As shown by Plezbert and Cytron [7], a compilation process can consist of running the unoptimized code while another process does aggressive compilation on the background. Once the optimized code is available, the unoptimized code is replaced with the optimized one. Since our system already mixes bytecode interpretation and binary execution, this *continuous compilation scheme* can be incorporated easily in Harissa.

Opportunities for Further Optimizations

While Harissa generated code is already fast, our micro-benchmarks show that there are still opportunities for improvement. In a near future, we plan to integrate an analysis for eliminating transient stack variables so as to be independent from Suif. Furthermore, we are studying the development of a transformation phase, based on control flow information, aimed at rebuilding loop constructs. As was shown earlier, structured programs are usually better compiled.

Finally, we also plan to eliminate some type and bound checks since close examination of the C code generated has demonstrated that most of could be evaluated statically by means of a simple intra-procedural analysis.

Acknowledgments

We would like to thank the Compose Group at Irisa and, in particular, Julia Lawall and Renaud Marlet for their helpful suggestions.

Availability

A binary version of our system is freely available by WWW and can be down-loaded from: `http://www.irisa.fr/compose/harissa-/harissa.html`

References

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

[2] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[3] OSF. The J-lite project. URL: http://www.gr.osf.org/java/jlite/-index.htm, 1996.

[4] David Intersimone. Interview with Régis Crelier. URL: http:/www.borland.com/internet/java/interviews/-regis2.html, 1996.

[5] Symantec Corporation. Just-in-time compiler for Windows 95/NT. URL: http://cae.symantec.com/cafe/-fs_jit.html, 1996.

[6] Softway. Introduction to Guava. URL: http://guava.softway.com.au/-introduction.html, 1996.

[7] 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.

[8] T. Keishiro. J2c Java .class to C translator. URL: http://www.webity.co.jp/-info/andoh/java/j2c.html, 1995.

[9] 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 *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[10] Pendragon Software. Caffeinemark 2.5. URL: http://www.webfayre.com-/pendragon/cm2/index.html, 1996.

[11] 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.

[12] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *OOPSLA' 96 Conference*, pages 93–100, San Jose (CA), October 1996.

[13] D. Ford. Jive: A Java decompiler. Technical Report RJ 10022, IBM Research Center, Yorktown Heights, May 1996.

[14] H.P. van Vliet. Mocha - Java decompiler. URL: http://www.inter.nl.net/users-/H.P.van.Vliet/mocha.htm, 1996.

[15] B. Case. Java virtual machine should stay virtual. *Microprocessor Report*, pages 14–15, April 1996.

[16] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 94.

[17] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[18] 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.

[19] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[20] F. Yellin. Low level security in Java. URL: http://java.sun.com/sfaq/verifier.html, December 1995.

[21] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *Proceedings of OOPSLA '95*, pages 108–123, Austin, TX, October 1995.

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

[23] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5),ACM Press.

[24] 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.

[25] M. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, Austin, TX, June 1995.

[26] G. Aigner and U. Holzle. Eliminating virtual calls in C++ programs. In *Proceedings of ECOOP '96*, Linz, Austria, August 1996. Springer-Verlag.

[27] U. Holzle and D. Ungar. Optimizing dynammically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language and Design Implementation*, pages 29(6):326–336. SIGPLAN Notices, June 1994.

[28] S. Voges. Othello game. voges@informatik.uni-muenchen.de, 1997.

[29] R.A. Milowski and P. Bothner. The Kawa scheme interpreter project. URL: http://www.copsol.com/kawa/index.html, 1996.

[30] KB Sriram. Free tools for java. URL: http://www.blackdown.org/~kbs/. 1996.

[31] J. Leach. John's java page. URL: http://lenna.easynet.it/~jhl/java.html, 1996.