Compiling Scheme to JVM bytecode: a performance study

Bernard Paul Serpette
Inria Sophia-Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
Bernard.Serpette@sophia.inria.fr
http://www.inria.fr/oasis/Bernard.Serpette

Manuel Serrano
Inria Sophia-Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
Manuel.Serrano@sophia.inria.fr
http://www.inria.fr/mimosa/Manuel.Serrano

ABSTRACT

We have added a Java virtual machine (henceforth JVM) bytecode generator to the optimizing Scheme-to-C compiler Bigloo. We named this new compiler BiglooJVM. We have used this new compiler to evaluate how suitable the JVM bytecode is as a target for compiling strict functional languages such as Scheme. In this paper, we focus on the performance issue. We have measured the execution time of many Scheme programs when compiled to C and when compiled to JVM. We found that for each benchmark, at least one of our hardware platforms ran the BiglooJVM version in less than twice the time taken by the Bigloo version. In order to deliver fast programs the generated JVM bytecode must be carefully crafted in order to benefit from the speedup of just-in-time compilers.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*evaluation strategies*

General Terms

Languages, Experimentation, Measurement, Performance

Keywords

Functional languages, Scheme, Compilation, Java virtual Machine

1. INTRODUCTION

Many implementors of high-level languages have already ported their compilers and interpreters to the Java Virtual Machine [21]. According to Tolksdorf's web page [34], there are more than 130 compilers targeting JVM bytecode, for all kinds of languages. In the case of functional languages (henceforth FLs), several papers describing such systems have been published (Haskell [36], Scheme [7, 23, 9], and SML [5]).

JVM bytecode is an appealing target because:

- It is highly portable. Producing JVM bytecode, it is possible to "compile once and run everywhere". For instance, the BiglooJVM Scheme-to-JVM compiler that is presented in this paper is implemented on Unix but the very same bootstrapped version also runs on Microsoft Windows.
- The standard runtime environment for Java contains a large set of libraries and features: widget libraries, database access, network connection, sound libraries, etc. A compiler which compiles a language to JVM bytecode could make these fancy features available to programs written in that language.
- A lot of programming environments and tools are designed for the JVM. Some have no counterpart for other languages (especially the ones that rely on the high-level features of Java such as its automatic memory management). Producing JVM bytecode allows these tools to be reused. For instance, we have already customized the Jinsight system [8] for profiling Scheme compiled programs.
- The JVM is designed for hosting high-level, object-oriented languages. It provides constructions and facilities such as dynamic type testing, garbage collection and subtype polymorphism. These runtime features are also frequently required for FLs.

1.1 Performance of JVM executions

In spite of the previously quoted advantages, compiling to JVM (or to Java) raises an important issue: what about performance? Is it possible to deliver fast applications using a FL-to-JVM compiler? Current Java implementations are known not to be very fast and, in addition, because the JVM is designed and tuned for Java, one might wonder whether it is possible to implement a correct mapping from a FL to JVM without an important performance penalty. The aim of this paper is to provide answers to these questions.

We have added a new code generator to the existing optimizing Scheme-to-C compiler Bigloo. We call this new compiler BiglooJVM. Because the only difference between Bigloo and BiglooJVM is the runtime system, BiglooJVM is a precise tool to evaluate JVM performance. We have used it as a test-bed for benchmarking. We have compared the

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

performance of a wide range of Scheme programs when compiled to native code via C and when compiled to JVM. We have used 21 Scheme benchmarks, implemented by different authors, in different programming styles, ranging from 18 lines long programs to 100,000 lines long programs (see Appendix A for a short description). We have tested these programs on three architectures (Linux/x86, Solaris/Sparc and Digital Unix/Alpha). We have found the ratios Bigloo-JVM/Bigloo quite similar on Linux/x86 and Sparc. This is not surprising as both configurations use the same JVM (HotSpot™ 2.0). We have thus decided not to present the Sparc results in this paper.

It would be pointless to measure the performance penalty imposed by the JVM if our compiler is a weak one. In order to "validate" the BiglooJVM compiler we have compared it to other compilers. In a first step, we have compared it to two other Scheme compilers: *i)* Kawa [7] because it is the only other Scheme compiler that produces JVM code and *ii)* Gambit because it is a popular, portable and largely used Scheme-to-C compiler. In a second step, we have compared BiglooJVM to other compilers producing JVM code: *iii)* MLj [5] and, *iv)* Sun's Java compiler. We present these comparisons in Section 2.

1.2 The Bigloo and BiglooJVM compilers

Bigloo is an optimizing compiler for the Scheme programming language [18]. It compiles *modules* into C files. Its runtime library uses the Boehm and Weiser garbage collector [6]. Each module consists in a sequence of Scheme definitions and top level expressions. After reading and expanding a source file, Bigloo builds an abstract syntax tree (AST) that passes from stage to stage. Many stages implement high-level optimizations. That is, optimizations that are hardware independent, such as *source-to-source* [27] transformations or *data flow* reductions [29]. Other stages implement simplification of the AST, such as the one that compiles Scheme closures [25, 30, 26]. Another stage handles polymorphism [20, 28]. That is, for each variable of the source code, it selects a type that can be polymorphic, *i.e.*, the type denoting *any* possible Scheme value, or monomorphic which can denote the type of primitive hardware values, such as *integers* or *floating point double precision.* At the C code generation point, the AST is entirely type-annotated and closures are allocated as data structures. Producing the C code requires just one additional transformation: the expressions of the AST are turned into C statements. In the BiglooJVM, the bytecode generator takes the place of this C statement generator (Figure 1).

Reusing all the Bigloo compilation stages is beneficial for the quality of the produced JVM bytecode and for the simplicity of the new resulting compiler. The JVM bytecode only represents 12% of the code of the whole compiler (7,000 lines of Scheme code for the JVM generator to be compared to the 57,700 lines of Scheme code for the whole compiler). The JVM runtime system is only 5,000 lines of Java code which have to be compared with 30,000 lines of Scheme code that are common to both JVM and C back-ends. This can also be compared with 33,000 lines of C code (including the Garbage Collector which is 23,000 lines long) of the C runtime system.

The first versions of BiglooJVM, a mere 2,000 lines of Scheme code, were delivering extremely poor performance applications. For instance, the JVM version of the **Cgc** benchmark (see Appendix A), when ran on a Linux/x86 architecture was more than 100 times slower than the corresponding C version. Modifying the produced JVM class files, we have been able to speed up this benchmark and it is now "only" 3 times slower than the C version. Many *small* problems were responsible for the initial poor performance. For instance, we found that some sequences of JVM instructions prevent the just-in-time compiler (JIT) from compiling methods. We also found that on some 32-bit architectures, using 64-bit integers is prohibitive. We found that some JIT (Sun HotSpot and others) compilers have a size limit per function above which they stop compiling. Finally, we found that some JIT compilers refuse to compile functions that include functional loops, that is, loops that push values on the stack.

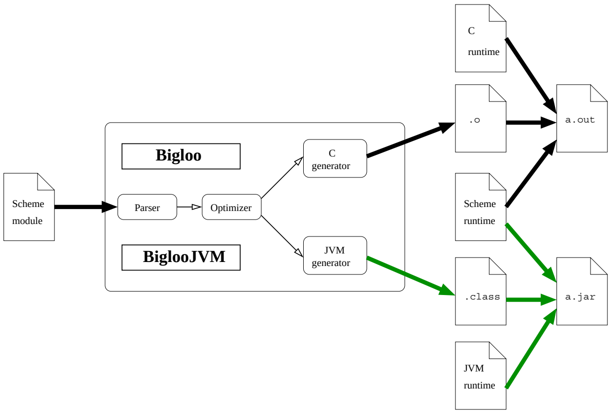

Figure 1: The architecture of the Bigloo compiler.

We present our experience with JVM bytecode generation in this paper. Most of the presented information is independent of the source language to be compiled so it could be beneficial to anyone wishing to implement a compiler producing JVM bytecode.

1.3 Overview

In Section 2 we present the performance evaluation focusing on the difference between Bigloo and BiglooJVM. In Section 3 we present the specific techniques deployed in the BiglooJVM code generator and the BiglooJVM runtime system. In Section 4 we present the important tunings we have applied to the code generator and the runtime system to tame the JIT compilers. In Section 5, we compare the BiglooJVM compiler with other systems.

2. PERFORMANCE EVALUATIONS

We present in this section the performance evaluation of BiglooJVM. In a first step, we compare it to Bigloo (that is the C back-end of Bigloo). Then, we compare it with other Scheme implementations. Finally, we compare it with non-Scheme implementations that produce JVM bytecode.

2.1 Settings

To evaluate performance, we have used two different architectures:

- Linux/x86: An AMD/Thunderbird 800Mhz, 256MB, running Linux 2.2, Java HotSpot™ of the JDK 1.3.0, used in "client" mode.

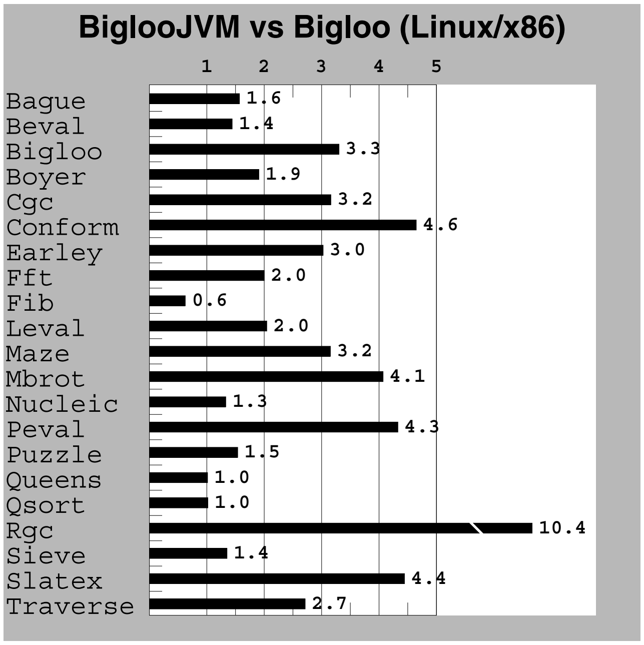

Figure 2: Scores are relative to Bigloo, which is the 1.0 mark. Lower is better.

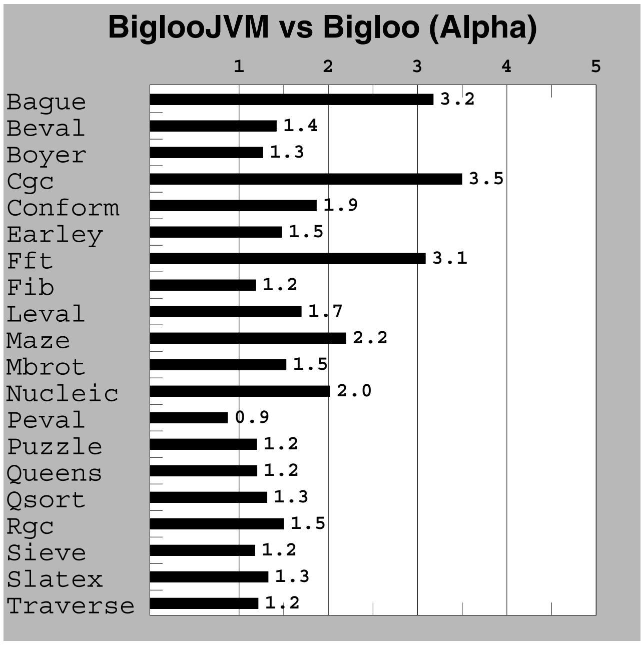

Figure 3: Scores are relative to Bigloo, which is the 1.0 mark. Lower is better.

- Alpha: An Alpha/21264 500Mhz, 512MB, running Digital Unix 4.0F, Classic VM version 1.2.2-3.

Each program was ran three times and the minimum sum of `cpu` + `system` time, as reported by the Unix `times` command, was collected. For JVM executions, the bytecode verifier is disabled (see Section 4.3).

In this paper we present ratios for which the base value is the execution time of the program when compiled to C by Bigloo. In consequence, when studying a configuration $\mathcal{K}$, we present the ratio of $\mathcal{K}$/Bigloo. The mark 1 is the execution time of Bigloo; lower is better.

2.2 BiglooJVM vs Bigloo

Figures 2 and 3 present the ratio of BiglooJVM/Bigloo for the Linux/x86 and Alpha platforms. The efficiency of JVM executions vary from one benchmark to another and from one platform to another. However, if for each benchmark, we consider the configuration with the smaller BiglooJVM/Bigloo ratio, we found that BiglooJVM measurements fall roughly between once and twice the time taken by Bigloo. The only three exceptions to this rule are: *i)* **Fib**, *ii)* **Bigloo** (the bootstrap of the Bigloo compiler) and, *iii)* **Cgc** (a toy compiler that produces Mips code for a C-like language). **Fib** is faster with the JVM implementation but tackling with C compilation options, Bigloo can reach the speed of BiglooJVM. More precisely if we prevent the C compiler from using a register as frame pointer (the gcc `-fomit-frame-pointer` option) then Bigloo performs as well as BiglooJVM. **Bigloo** and **Cgc** benchmarks are slower when compiled to JVM with a deficient ratio of 3.2. It would have been interesting to study the performance of **Bigloo** on the Alpha platform, unfortunately, the virtual machine crashes for that benchmark because of stack overflow and we have found no mean to extent the stack on this platform. We are currently testing a new version of BiglooJVM that reduces the stack consumption of the produced code. On the other hand, we explain in Section 3.3.1 the reason why **Cgc** is slower when ran with the JVM runtime. Comparing the "best of all platforms" (that is, considering the time gathered on the Linux platform when the ratio BiglooJVM/Bigloo is the smallest, considering the Alpha platform otherwise) demonstrates that there is no technical impossibility for a JVM implementation to deliver performance comparable to the one of C. The Compaq implementation on Alpha almost achieves this level of performance.

2.2.1 *Variations on the JVM implementation*

It exists many implementations of the JVM. Some are based on pure interpretation techniques [12]. Some are based on static compilation of JVM bytecode to native code [17]. Finally, some are based on just-in-time compilers [19, 1]. Amongst the last category some use adaptive compilation [32, 31, 3] which was pioneered by the Self project [14, 15]. It consists in an on-demand compilation of the "hot spots" of a program. Interpreters cannot compete with compilers for performance thus we have not tested any. In addition, we have not considered testing static JVM compilers because one of the most important advantage of JVM over C is the ability to compile once and to run everywhere. This property does not hold for static native compilation. In addition, previous experiments show that this compilation technique has a moderate impact on performance [16].

For the Linux operating system alone, more than ten implementations of the JVM exist. It is beyond the scope of this paper to evaluate all of them. According to the *Java performance report* [22], two implementations perform best: Sun's HotSpot (JDK 1.3) [32] and IBM JDK 1.3 [31]. HotSpot seems to be the most reliable one on Linux so it is our base reference. In Figure 4 we present the performance evaluations of BiglooJVM on Linux/x86 using the different JVM implementations. Only HotSpot/client is able to

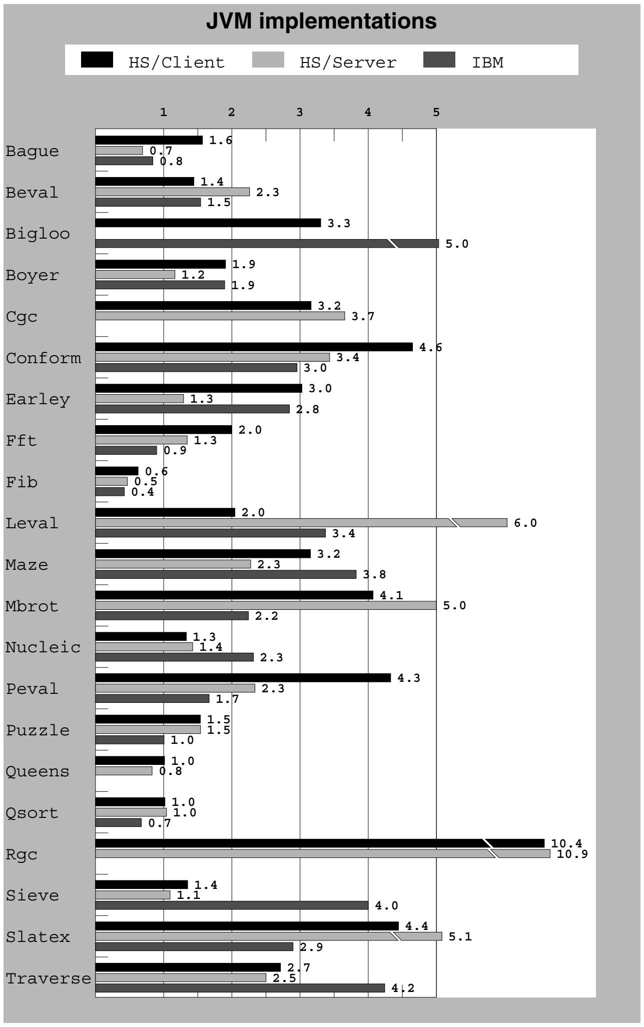

Figure 4: BiglooJVM with Sun's JDK, with Sun's JDK/server and with IBM's JDK vs Bigloo on Linux/x86 architecture. Lower is better.

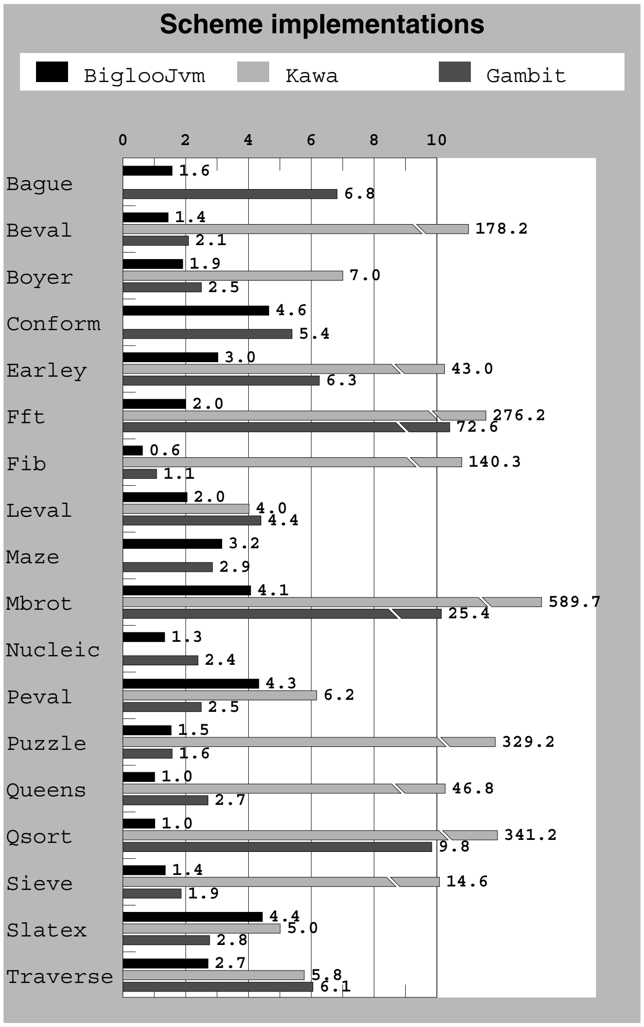

Figure 5: BiglooJVM, Kawa and Gambit vs Bigloo on Linux/x86 architecture. Lower is better.

run all the benchmarks. Missing values denote a JVM failure. It seems that these three machines implement different adaptive compilation strategies because they deliver different performance. We cannot conclude that one machine *always* delivers the best performance. We can thus only use this test to emphasize that: *i)* performance of JVM is hardly predictable and highly dependent of the hardware/software platforms, and *ii*) comparing all the "best" JVM measured execution times to the C execution times, we found that only one benchmark took more than twice as long when run on the JVM.

2.3 BiglooJVM vs Scheme implementations

Figure 5 presents a comparison of BiglooJVM, Kawa v1.6.7 and Gambit v3.0 (unsafe mode and optimizations enabled) on Linux/86. Kawa produces JVM bytecode, Gambit produces C code. For all tested benchmarks BiglooJVM is significantly faster than Kawa. One may notice that since Kawa unfortunately does not provide specialized arithmetic operations, slow executions are therefore to be expected. The Kawa compiler does not seem able to automatically turn generic arithmetic function calls to specialized arithmetic calls. So, Kawa programs *always* use generic arithmetic. This is a huge handicap.

For all but two programs, BiglooJVM is faster than Gambit. Gambit is not mainly designed for maximal efficiency. It implements the *exact* definition of Scheme when Bigloo does not[1]. In particular, Gambit performance is affected by the implementation of tail recursive calls. This speed test between Gambit and Bigloo thus only demonstrates that BiglooJVM deliver programs with "acceptable" performance since Gambit is considered by the Scheme community as an efficient enough system. In consequence, we think that if speed is not an implementor's highest priority, the JVM should be considered.

2.4 BiglooJVM vs other languages

It is always difficult to compare compilers for different

[1] Bigloo is not *fully* Scheme compliant mainly because it is not properly tail-recursive and because it does not implement arithmetic bound checking.

programming languages. Because the programs cannot be the very same, we can fear that a small difference in the implementation invalidates the test. For instance, Scheme and SML share a lot of features and constructions: they both are functional languages, promoting first class closures and polymorphism, using garbage collection, etc. However the programming styles of these two languages differ. It is thus to be expected that a compiler for Scheme and a compiler for SML optimize different patterns of source expressions. It is likely that a Scheme program written using an SML style would not be efficiently optimized by a Scheme compiler and vice-versa. In order to avoid these pitfalls we have only tested small and simple programs that we have tried to implement using the natural style of each programming languages. In addition, we have slightly modified some benchmarks in order to prevent the Bigloo compiler from applying too aggressive optimizations. For instance, the original version of the **Mbrot** benchmark was running 40 times faster when compiled by BiglooJVM than when compiled by Javac, the Sun's standard Java compiler. This was only due to inlining and constant folding. Since we do not want, in this paper, to emphasize compiler techniques and optimizations but runtime system efficiency, we have found relevant to write a "neutral" benchmark version that prevents Bigloo optimizations. However, one should note that because of the Java object model and because of dynamic class loading, a compiler for FLs to JVM bytecode is likely to have more opportunities to optimize the source code than a Java compiler. We present the comparison between BiglooJVM, MLj and Javac compilers Figure 6.

On the five small programs we have used, we have found that BiglooJVM and MLj (version 0.2c) deliver programs with comparable performance.

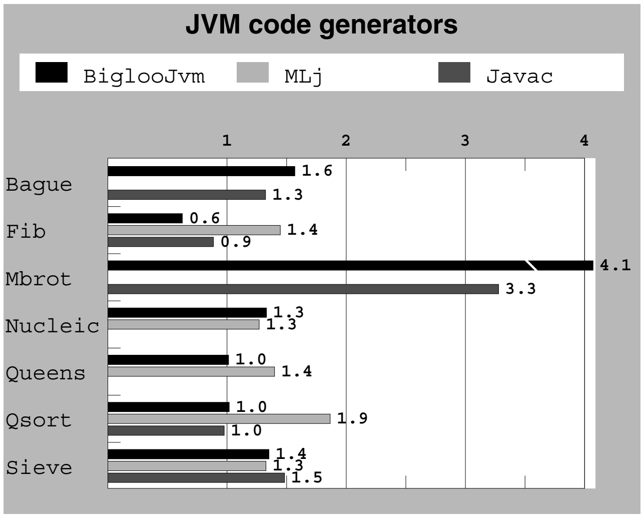

Figure 6: BiglooJVM, MLj and Javac vs Bigloo on Linux/x86 architecture. Lower is better.

We decided to perform these tests for Java too, even though we assumed that a Java compiler would produce better code for the JVM than BiglooJVM. Surprisingly, our programs performed comparably to those produced by Sun's Javac (JDK 1.3). In fact, on the **Sieve** benchmark Sun's code is even slower. To us, this demonstrates that BiglooJVM makes efficient use of the JVM.

3. THE BiglooJVM BACK-END

In this section we present the most important points of the compilation of Scheme, *i.e.* a functional programming language using dynamic type checking, to JVM bytecode. We start, in Section 3.1, by presenting the compilation of functions. We then discuss in Section 3.2 the implementation of dynamic type checking. We conclude in Section 3.3 with a description of data representation.

3.1 Compiling functions

Bigloo uses three different frameworks for compiling functions. We first present in Section 3.1.1 how local functions can be compiled to loops. Then in Section 3.1.2 we show the general framework applied to functions. In Section 3.1.3 we present how does Bigloo handle closures. Finally, in Section 3.1.4 we present limitations of our JVM back-end.

3.1.1 *Compiling functions to loops*

Closure analysis aims at compiling Scheme functions into JVM loops. This framework applies when local Scheme functions do not escape (that is, when functions are not used as first-class values) and when these functions are always invoked tail-recursively (see Section 3.1.4 for a discussion of generalized tail recursion). Here is an example of two mutually recursive functions that map to JVM loops:

```
(define (odd? x)
  (letrec ((even? (λ (m)
                    (if (= m 0) #t (odd? (- m 1)))))
           (odd? (λ (n)
                    (if (= n 0) #f (even? (- n 1))))))
    (odd? x)))
```

is compiled as:

```
(method odd? (x) (n m)
    (iload x)
    (istore n)
L1  (iload n)       ;; beginning of odd?
    (iconst_0)
    (if_icmpne L2)  ;; compare the argument n with 0
    (iconst_0)      ;; return (for odd?) false
    (ireturn)
L2  (iload n)       ;; prepare the actual value for
    (iconst_1)      ;; the even? function
    (isub)
    (istore m)
    (iload m)       ;; even? is inlined here
    (iconst_0)
    (if_icmpne L3)  ;; compare the argument m with 0
    (iconst_1)      ;; return (for even?) true
    (ireturn)
L3  (iload m)       ;; otherwise prepare the call to odd?
    (iconst_1)
    (isub)
    (istore n)
    (goto L1))      ;; goto odd?
```

As illustrated by the **Qsort** or **Sieve** benchmarks that use loops extensively, this compilation framework delivers good performance.

3.1.2 *Compiling functions to JVM methods*

When the previous scheme does not apply, non-escaping n-ary functions map to JVM n-ary static methods. Consider the Scheme `fib` definition:

```
(define (fib x)
   (if (< x 2)
       1
       (+ (fib (- x 1)) (fib (- x 2)))))
```

It is compiled as:

```
 1:  (method fib (x) ()
 2:      (iload x)
 3:      (iconst_2)
 4:  ;; test with numeral 2
 5:      (if_icmpge 9)
 6:      (iconst_1)
 7:      (ireturn)
 8:  ;; first recursive call
 9:      (iload x)
10:      (iconst_1)
11:      (isub)
12:      (invokestatic fib)
13:      (iload x)
14:  ;; second recursive call
15:      (iconst_2)
16:      (isub)
17:      (invokestatic fib)
18:      (iadd)
19:      (ireturn))
```

The good performance of the **Fib**, **Beval** and other benchmarks using non tail recursive calls shows the efficiency of this compilation framework. The JVM's static method invocation is as fast as C's function invocation.

3.1.3 *Higher-order and Heap-allocated closures*

Scheme is a higher-order language: functions are first-class objects which can be passed as arguments, returned by functions and stored in memory. The front-end of the compiler resolves the lexical visibility of variables inside functions by making explicit closures on demand. The back-end still has to implement an abstract type *pointer-to-function* (defined in a straightforward manner for the C back end). Fortunately, this abstract type was designed to appear only in specific functions (closure creators) and is used indirectly by specific special forms (`funcall` and `apply` nodes). This allows us to implement *pointer-to-function* as an index (integer) into a switch tables for the JVM back-end. This technique is similar to the one deployed in Gambit to implement tail recursive calls [11]. One unique index is allocated for each static closure creation on a compilation unit (a Bigloo module). Indexes may be identical for 2 different modules (since one switch table has been allocated per module). Consider the simple following Bigloo module example:

```
(module test (export app0 app1))
(define (app0 f l) (f l))
(define (app1 f l) (f (reverse l)))
```

The compiler generates a class file similar to:

```
public class test extends bigloo.procedure {
   public static procedure app0 = new test( 2, 0 );
   public static procedure app1 = new test( 2, 1 );
   public test( int arity, int index ) {
      super( arity, index );
   }
   public Object funcall2( Object a1, Object a2 ) {
      switch( this.index ) {
         case 0: return anonymous0( this, a1, a2 );
         case 1: return anonymous1( this, a1, a2 );
         default: funcall_error2( this, a1, a2 );
      }
   }
   private static Object anonymous0(
     procedure fun, Object a1, Object a2 ) {
      return ((procedure) a1).funcall1( a2 );
   }
   private static Object anonymous1(
     procedure fun, Object a1, Object a2 ) {
      return ((procedure) a1).funcall1( reverse( a2 ) );
   }
}
```

Where `procedure` is a runtime Bigloo class defined as:

```
package bigloo;
public abstract class procedure {
   public int index, arity;
   public Object[] env;
   public procedure( int arity, int index ) { ... };
   public abstract Object funcall1( Object a1 );
   public abstract Object funcall2( Object a1, Object a2 );
}
```

One should keep in mind that according to Sections 3.1.1 and 3.1.2, this generic compilation framework is used only when the compiler is unable to statically discover which functions are called at an application site. However, we found that some JITs are unable to produce code with a constant complexity for the `tableswitch` instruction (see Section 4.4.3).

An alternative compilation is used by Kawa [7], where each function used as value is compiled into a new JVM class. This compilation framework gets rid of `tableswitch`es. Our different solution is motivated by reducing the number of generated classes. For instance, the Kawa technique would yield to more than 4,000 classes to compile the closures of the Bigloo bootstrap, with each class been compiled into a separate `.class` file!

Moreover, in the current Bigloo version, each global that is declared to be accessible from the Scheme interpreter, is associated with two Bigloo functions; one to get the value of the global, one to set a new value. This would also increase significantly the number of generated classes.

3.1.4 *Call/cc and tail recursion*

BiglooJVM imposes one restriction to Bigloo: continuations can only be invoked in the dynamic extent of the `call/cc` expression from which they have been reified. This restriction comes from the design of the JVM instruction set and from the bytecode verifier. The JVM does not provide a mean to save and restore execution stacks.

The lack of general stack operators also makes very difficult a "correct" JVM implementation of tail calls. In the current BiglooJVM version, only recursive calls to local functions are correctly handled, *i.e.*, without stack consumption (see Section 3.1.1 for an example and a previous article [29] by the authors for more details). The same restriction applies to the C Bigloo runtime. In addition, we think that the trampoline technique [4, 33, 11] is infeasible for the current JVM implementation, as it would cause an excessive performance penalty (see Section 3.1.3). Performance of JVM executions highly depend on the performance of embedded JIT compilers. Because executed at application runtime, these compilers have to be extremely fast. They only have the opportunity to deploy simple and local optimizations. It is thus likely that JIT compilers will fail at optimizing *pure* trampolined code, which is an unusual style of JVM bytecode. On the other hand, a recent study shows a variant of trampolined code that appears to fit the requirement of JIT compilers and that could be used to implement tail recursion under the JVM [24].

3.2 Dynamic typing

The Scheme type system may be viewed as a huge, unlimited union type. Since the JVM does not provide union types or parametric types, the usual way to mimic union types is to use subtyping. In order to specify a union type T which is a sum of some T_i, it is usual to declare a class T and implement all T_i as subclasses of T. Type-dependent behaviors take the form of methods added to each subclass. This strategy impacts earliest stages of the compiler. This would have forced widespread changes to the existing Bigloo source code, and we decided not to use this design. Instead, we have used the JVM instruction `instanceof` that implements dynamic type checking. As in the case of the `tableswitch` instructions, we have found that some JITs fail to compile `instanceof` efficiently. For instance, Sun's JDK 1.3 on Linux/x86 is 40 times slower to answer "no" than to answer "yes" to `instanceof`. This explains some distortions on the benchmarks results. As for `tableswitch`, the Alpha implementation of the JDK does not seem to suffer from this problem.

We plan to get rid of the `instanceof` instruction by adding a specific tag to all Scheme objects. The drawback of this implementation is that all foreign values require wrapping in a specific Scheme type. We hope that the SUA optimization [28] will do the same nice job as for boxed integers (see Section 3.3.1).

3.3 Data representation

We present in this section the BiglooJVM data representations. We start showing in Section 3.3.1 the main differences in the representation of data structures between the C runtime and the JVM runtime. Then, in Section 3.3.2, we focus on the representation of integers in the JVM runtime system.

3.3.1 *Tagging, boxing and pointers alignment*

Some well-known C tricks enable faster implementations of dynamic type identification [13]. These techniques rely on hardware alignment requirements. Because pointers are aligned on four- or eight-byte boundaries, the two or three least significant bits of values can be used to encode type information. In the Bigloo C back-end we use these bits for two purposes: *i)* using tagged instead of boxed representation for integers, and *ii)* using lightweight two-words representation for pairs. Because Java is a safe high-level language it does not provide operations on pointers and thus, these C tricks do not apply to the BiglooJVM runtime system. In order to estimate the impact of these techniques, we have built two additional versions of the regular Scheme-to-C version of Bigloo: *i)* the first one, called Bigloo32, is identical to Bigloo but integers are 32 bits long and boxed; *ii)* the second one, called Bigloo32+ is identical to Bigloo32 but pairs are implemented using 3 memory words (one more than with Bigloo and Bigloo32).

Figure 7 compares BiglooJVM and these two new versions. We see that **Cgc** is highly sensitive to integer boxing. When compiled to C using boxed integers, this program is no longer faster than its JVM counterpart. **Cgc**. Most of its execution time is spent in a library of bit-vectors. These are used for data flow analyses such as liveness or reachability property computations. In **Cgc** the bit-vectors are implemented using vectors of integers. Because of the programming style of **Cgc**, the Bigloo compiler is unable to demonstrate that these vectors cannot contain non-integer values. Thus, it cannot optimize these vectors. That is, it cannot replace them with native integer vectors. With the C tagged version of Bigloo this is not a performance problem. The main consequence of tagging is that a couple of bits is wasted for each of the word of the bit-vectors, making the vectors slightly larger. With the boxed version of integers, bit-vector operations become expensive because new integers have to be allocated. That is, each time a set operation is computed such as an union, a disjunction, etc., new integers are allocated. One may argue that this problem is due to the poor implementation of **Cgc**. It is clear that **Cgc** could be easily improved. However, we wanted to use this program "as is" because this current version points out one real problem of the JVM version. If integers cannot be unboxed by a compiler, the JVM runtime system performance could be much slower than the one of C.

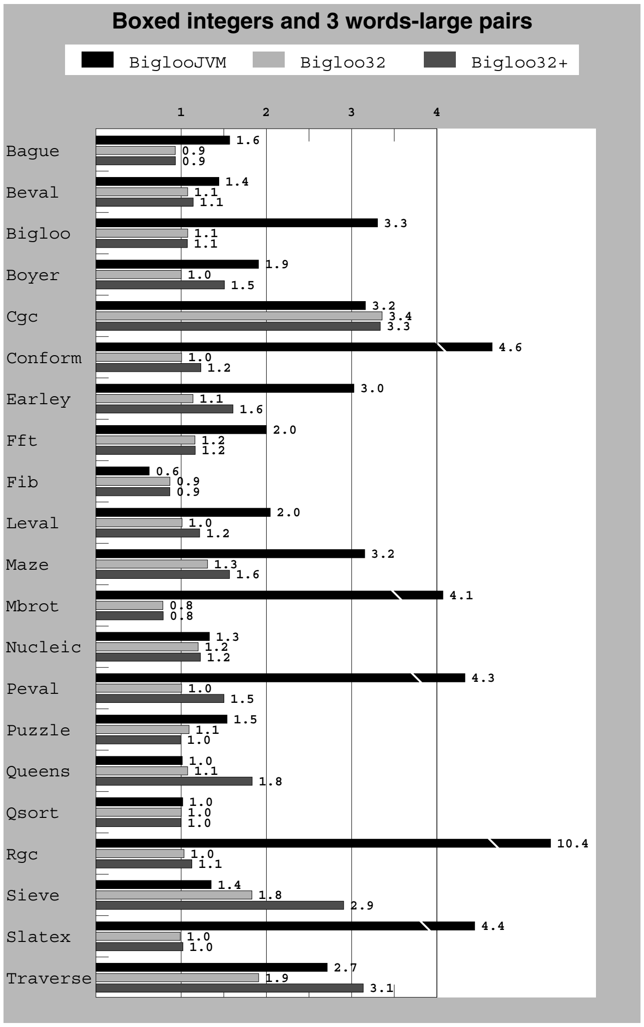

Figure 7: BiglooJVM, Bigloo C with boxed integers (Bigloo32) and Bigloo C with boxed integers and 3 words long pairs (Bigloo32+) vs Bigloo on Linux/x86 architecture. Lower is better.

3.3.2 *Integer arithmetic*

As presented Section 3.3.1, BiglooJVM uses heap-allocated boxed integers such as Java `Integer` instances. Boxing integers is expensive but preallocating small ones avoid memory consumption and memory allocation in most cases (for example all 256 Scheme characters are preallocated). For 32-bit integers we have preallocated integers in the range

[-100...2048]. For the **Bigloo** benchmark, only 3.8% of integers are outside this range. Thanks to the SUA [28], an optimization that enables unboxing for polymorphic languages, fewer than 4% of the static arithmetic operations (including loading constants) need boxing for this benchmark.

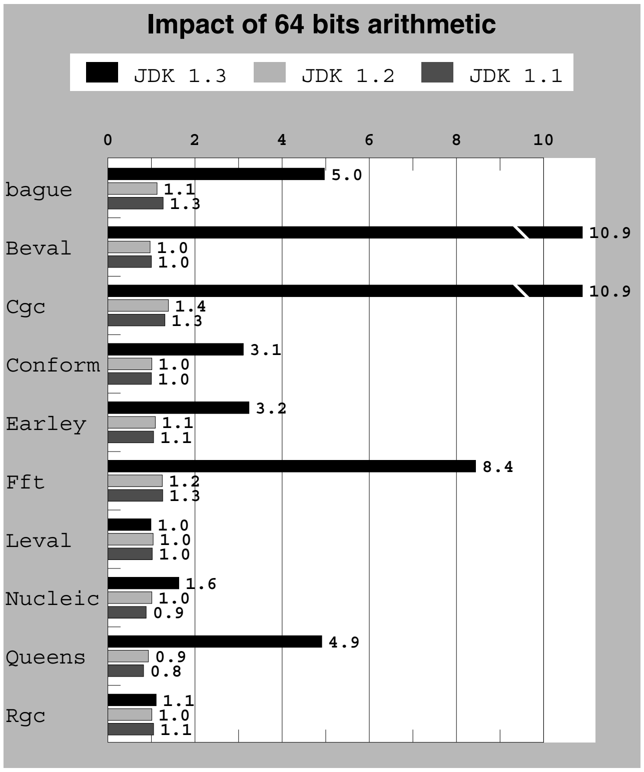

Figure 8: BiglooJVM-64 bits/BiglooJVM-32 bits on Linux/86. Lower is better.

Aside from boxing, some choices have to be made for the size of basic types. These choices are directed by the target languages. For C, sizes are unspecified but the `long` type is usually large enough to contain any address. By contrast, the JVM specifies the length of all basic types. In order to choose between 32-bit integers and 64-bit arithmetic integers we have tested the two settings with the Sun's JDK on Linux/x86. This test is presented Figure 8. Because 64-bit arithmetic in JDK1.3 has such a major performance penalty, we decided to use 32-bit arithmetic on all platforms. Using this configuration, JDK 1.3 is the fastest JVM we have found on Linux/x86 and Solaris/Sparc.

4. JVM BYTECODE TUNING

Because of the currently leading technology relying on dynamic compilation for executing JVM programs, special attention must be paid to produce JVM bytecode that pleases the JIT compilers. We report the most important facts we have learned during our work on BiglooJVM.

4.1 Data flow optimizations

Data flow optimizations turn out to be important. We have found that certain JVM bytecode sequences disable some JIT compilers. Since JIT compilers such as HotSpot use methods as compilation units, it is important to remove such sequences; otherwise, the whole function that contains it is left uncompiled, *i.e.*, interpreted.

The Bigloo compiler already implements some data flow analysis. However, we have added three new transformation rules for the JVM bytecode generator that are enabled when the compiler is able to prove that expressions are side-effect free:

1. Let bindings with no bound variables are removed:

$$\mathcal{R}_1 \left[\!\!\left[\begin{array}{c} \ddots \\ \boxed{\texttt{(let () } expr\texttt{)}} \\ \ddots \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{c} \ddots \\ \boxed{expr} \\ \ddots \end{array} \right]\!\!\right]$$

2. Let-bound variables that are only used in a function called are reduced:

$$\mathcal{R}_2 \left[\!\!\left[\begin{array}{c} \ddots \\ \boxed{\begin{array}{l}\texttt{(let (... (x } expr\texttt{) ...)} \\ \quad\texttt{(}f\texttt{ ... x ...))}\end{array}} \\ \ddots \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{c} \ddots \\ \boxed{\begin{array}{l}\texttt{(let (... ...)} \\ \quad\texttt{(}f\texttt{ ... } expr\texttt{ ...))}\end{array}} \\ \ddots \end{array} \right]\!\!\right]$$

3. Let-bound variables used only in a test position are reduced:

$$\mathcal{R}_3 \left[\!\!\left[\begin{array}{c} \ddots \\ \boxed{\begin{array}{l}\texttt{(let (... (x } expr\texttt{) ...)} \\ \quad\texttt{(if x ...))}\end{array}} \\ \ddots \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{c} \ddots \\ \boxed{\begin{array}{l}\texttt{(let (... ...)} \\ \quad\texttt{(if } expr\texttt{ ...))}\end{array}} \\ \ddots \end{array} \right]\!\!\right]$$

None of these rewriting rules are difficult to implement and they are well-known [2, 35]. However, they are important for our JVM back-end. For instance, disabling rule $\mathcal{R}_2$ slows down the **Bigloo** benchmark by more than 20%, because the pattern reduced by rule $\mathcal{R}_2$ is frequently introduced by earlier Bigloo compilation stages.

4.2 Functional loops

On the one hand, functional languages are expression oriented languages. On the other hand, Java, as C, is statement-oriented. Scheme loops (*e.g.*, Scheme inner functions) are expressions, that is, there evaluations produces a value. This feature is depicted in this naive example:

```
(define (test a n b) ;; compute a + b * n
   (define (mul m r)
      (if (= m 0) r (mul (- m 1) (+ b r)) ))
   (+ a (mul n 0)))
```

In a first version of the back-end we had left the value of `a` in the JVM stack. The generated bytecode was:

```
(method test (a n b) (r m)
     (iload a)        ;; Prepare a for addition
     (iload n)        ;; Prepare arguments for mul: m
     (iconst_0)       ;; ...: r
muls (istore r)       ;; stack entry for mul, store in reg r
     (istore m)       ;; ... reg m
mul  (iload m)        ;; reg entry for mul
     (iconst_0)
     (if_icmpne l1)   ;; m == 0 ?
     (iload r)        ;; return loop with r
     (iadd)           ;; continuation of the loop
     (ireturn)
l1   (iload m)        ;; Prepare arg 1 of mul: m-1
     (iconst_1)
     (isub)
     (iload b)        ;; Prepare arg 2 of mul: b+r
     (iload r)
     (iadd)
     (goto muls))     ;; back to stack entry of mul
```

We have found that some JITs do not compile this programming style and we now require that all loops (backward branches) must be done in a "statement" style, with an empty stack. (One should note that such loops used as expressions cannot be produced by a Java compiler because Java is a statement-based language.) As a result, the current BiglooJVM version now produces:

```
(method test (a n b) (reg1 r m save)
     (iload a)       ;; Prepare a for addition
     (iload n)       ;; Prepare arguments for mul: n
     (iconst_0)      ;; ...: r
muls (istore r)      ;; stack entry for mul: store in reg r
     (istore m)      ;; .. reg m
     (istore save)   ;; SAVE THE STACK
mul  (iload m)       ;; reg entry for mul
     (iconst_0)
     (if_icmpne l1)  ;; m == 0 ?
     (iload r)       ;; return loop with r
     (istore reg1)   ;; RESTORE THE STACK
     (iload save)    ;; RESTORE THE STACK
     (iload reg1)    ;; RESTORE THE STACK
     (iadd)          ;; continuation of the loop
     (ireturn)
l1   (iload m)       ;; Prepare arg 1 of mul: m-1
     (iconst_1)
     (isub)
     (iload b)       ;; Prepare arg 2 of mul: b+r
     (iload r)
     (iadd)
     (istore r)      ;; prepare for reg entry point of mul
     (istore m)
     (goto mul))     ;; back to reg entry of mul
```

In order to perform this transformation, the code generator must know the type of stack elements at each loop entry. The size of the stack is not enough since for each element we have to know if we have to generate an `istore` or an `astore` instruction. Even if more complex, this second version runs 18 times faster on SUN JDK1.3 or 1.2 than the former one!

4.3 Bytecode verification

Since all functions (or more generally methods) must have a prototype (declaration of arguments and return type), the JVM may be considered as a strongly typed language. These prototypes are used for two main reasons. First, it is possible to define two different functions with the same name and use the prototype as an additional key for linking purposes. This is not to be confused with Java *overloading* which is resolved statically at compile time. Second, prototypes are used by the bytecode verifier at runtime. Bytecode verifier compliant programs must explicitly, at runtime, check downward casts. Event if for instance, in order to comply to the JVM bytecode verifier the following Scheme function definition:

```
1:  (define (first x)
2:     (cond
3:        ((pair? x) (car x))
4:        ((vector? x) (vector-ref x 0))
5:        (else #f)))
```

must be compiled such as:

```
1:  (method first (x) ()
2:    (aload x)
3:    (instanceof pair)
4:    (ifeq 9)
5:    (aload x)
6:    (checkcast pair)
7:    (getfield car)
8:    (areturn)
9:    (aload x)
10:   (instanceof [jobject)
11:   (ifeq 17)
12:   (aload x)
13:   (checkcast [jobject)
14:   (iconst_0)
15:   (aaload)
16:   (areturn)
17:   (getstatic bfalse)
18:   (areturn))
```

That is, if the compiler is able to prove that the type of `x` is a `pair` line *3* or a `vector` line *4* depending on the branch of the `cond`, it still has to enforce a dynamic type test when fetching values from the data structure, such as lines *6* and *13*. One may enable or disable the bytecode verifier when running JVM programs. Because the bytecode generated produced by BiglooJVM is not bytecode verifier compliant all the previously given time figures have been gathered with the bytecode verifier disabled. BiglooJVM provides an option to generate code that complies with the verifier. We have not yet implemented the optimization which removes unnecessary `checkcast` instructions. We have noticed that the current overhead of the bytecode verifier compliance ranges from 10% to 55%. We think that this slowdown is acceptable, so we have postponed the work on this topic.

4.4 JVM idiosyncrasies

We have noticed that current JVM implementations are highly sensitive to the shape of the generated bytecode. In this section we present three JIT related problems we have encountered and the solution we have applied for two of them.

4.4.1 *Startup time*

In order to gather accurate time informations, we have considered long lasting benchmarks. All but **Bigloo** and **Cgc** benchmarks, when compiled to C, last more than 10 seconds on our AMD Athlon 800 Mhz. However, we must point out that JVM applications have long startup times. For instance, when **Bigloo** is invoked with the `-help` option which displays command line options and exits, the C time is about 0.02s while the JVM time is 2.52s. That is, the startup time of the JVM version is 126 slower than the C one! This extremely slow startup time explains the poor result of the overall **Bigloo** benchmark for the JVM version. Prohibitive startup times are a problem for short-lasting applications. For instance, it is currently impossible to use BiglooJVM to implement shell-like commands such as `ls`, `cat`, etc. We have no solution to this problem yet.

4.4.2 *Method size considerations*

We have noticed that the current JIT compilers use a compilation threshold. In particular, we have experimentally found out that Sun's HotSpot stops compiling functions for which the body is larger than 8000 JVM bytecode instructions. Expectingly, some of our benchmarks suffer from this strict limitation. The most significant one is **Rgc** which performs miserably with HotSpot (see Figure 2). This benchmark uses Bigloo regular grammars. These lex-like grammars are compiled to deterministic finite automata that are, in turn, compiled into Scheme functions. Each grammar is compiled into one global Scheme function for which the states of the automaton are implemented using local functions. Because all function calls to these local functions are tail calls, the Bigloo compiler is able to inline or *integrate* them (see Section 3.1.1). In consequence, for one regular grammar, Bigloo produces exactly one JVM method. The **Rgc** benchmark uses a large grammar that implements a complete Scheme reader (the actual Bigloo reader). When compiled to JVM this grammar uses 8252 JVM instructions. That is 252 instructions more than the accepted number! In consequence the JVM method resulting of the compilation of the regular grammar is left uncompiled by HotSpot. All the execution time of the benchmark is spent in the compiled regular grammar. This explains why that benchmark performs so badly with this Java virtual machine. We hope that this problem will be fixed in the next HotSpot version. A JVM flag enabling user customization of the compilation threshold would be a workaround to this problem.

4.4.3 *JVM compilation of tableswitch*

We found that some JITs are unable to produce code with a constant complexity for the `tableswitch` instruction. For example, on Linux/x86, using Sun's JIT, `tableswitch` implementation is linear in time. More precisely, we found that the execution time of `tableswitch` is $17.68 * index + 34.39$ nanoseconds. On the same platform, `iadd` was benchmarked around 14 nanoseconds. When *index* is not in the specified range, the longest time is required. This poor compilation of `tableswitch` explains some distortions on the benchmarks results. One should note that stable timing results were measured on the Alpha platform. The efficiency of this compilation framework is demonstrated by the good performance of benchmarks using higher order functions (*e.g.*, **Queens**, **Leval**).

5. RELATED WORK

In this section we compare BiglooJVM with two others Scheme to JVM compilers, Kawa and with a ML to JVM compiler, MLj.

5.1 The Kawa Scheme compiler

Kawa is another Scheme compiler to JVM. Kawa implementation techniques differ from BiglooJVM in many respects. For instance, Kawa closures are implemented by the means of classes (see Section 3.1.3). Kawa strings differ from BiglooJVM strings. Kawa maps Scheme strings into Java strings, while BiglooJVM maps them into array of bytes. Java strings are not suitable for implementing Scheme strings because the former are immutable while the later are mutable.

The Kawa `eq?` predicate has a different meaning than the BiglooJVM one's. Kawa `eq?` is false for integers, which is a correct implementation, according to the Scheme Revised^5 Report [18]. Bigloo implements exact `eq?` for integers. Thus, for the sake of compatibility between the two runtime systems, we have decided to provide the same semantics for `eq?` with BiglooJVM. Providing integers `eq?` impacts the performance of `eq?`. As reported Section 3.3.2, integers are allocated which leads to the following implementation of `eq?`:

```
boolean EQ( Object o1, Object o2 ) {
  if( o1 == o2 ) return true;
  return( ( o1 instanceof BINT ) &&
          ( o2 instanceof BINT ) &&
          ((BINT)o1.value == (BINT)o2.value) );
}
```

If equality is not considered for integer, `eq?` could be:

```
boolean EQ( Object o1, Object o2 ) {
  return o1 == o2;
}
```

When, thanks to the SUA, the Bigloo front-end succeeds in demonstrating that either one of the `eq?` arguments is not an integer, this fast implementation is preferred to the former one. In order to minimize the performance slow down of the compatibility between Bigloo and BiglooJVM we have added a compilation flag that switches from compatible to incompatible `eq?`. For all benchmarks, we have disable `eq?` compatibility.

5.2 The MLj compiler

MLj is an optimizing compiler for Standard ML producing JVM bytecode. BiglooJVM and MLj share a main goal: efficiently compiling a strict functional language to JVM. The paper [5] concentrates on the front-end of that compiler and does not detail its bytecode production. The present paper focuses on the runtime system. The two papers are thus complementary.

It seems that MLj uses type information to avoid boxing when Bigloo uses a global analysis. The MLj analysis is too briefly described. For this reason, we cannot compare the respective accuracy of the two approaches. The most important difference between MLj and BiglooJVM seems to be that MLj does not support separate compilation while Bigloo does [5]. The authors of MLj state that "...the reasonable [MLj] performance has only been achieved at the price of high compile times and a limitation on the size of programs which may reasonably be compiled. Our decision to do whole-program optimization is certainly controversial...". As reported in Figure 6 it seems that BiglooJVM produces comparable performance without this limitation. In addition, BiglooJVM supports separate compilation. So, the BiglooJVM, which is a program of about 100,000 lines of code (the compiler plus the runtime system) is bootstrapped. On the other hand, the good performance of BiglooJVM is achieved at the price of producing code that does not comply with the bytecode verifier while MLj does not suffer from this limitation. Using a compilation option, BiglooJVM produces bytecode verifier compliant code but this currently slows down executions.

6. CONCLUSION

We have added a new JVM bytecode generator to the Bigloo Scheme-to-C optimizing compiler. In order to estimate, from a performance point of view, how suitable JVM is as a target for compiling strict functional languages, we compared the applications when run on the C-based and the JVM-based runtime systems. We have tested several implementations of the JVM on several different hardware architectures. Given a code generator that is carefully written and aware of the JVM's idiosyncrasies, we found that most of the JVM compiled applications are no more than twice as slow as compiled C applications. The performance of JVM has not ceased to increase significantly since the first implementation. If JVM implementations keep improving at this pace, we think that in a near future, JVM bytecode may become a true performance challenger.

ACKNOWLEDGMENTS

Many thanks to Christian Queinnec, Xavier Leroy, John Clements, and to Céline for their helpful feedbacks on this work.

7. REFERENCES

[1] A. Adl-Tabatabai, M. Cierniak, G-Y. Lueh, V. Parikh, and J. Stichnoth. **Fast, Effective Code Generation in a Just-In-Time Java Compiler**. In *Conference on Programming Language Design and Implementation*, pages 280–190, June 1998.

[2] A. Aho, R. Sethi, and J. Ullman. **Compilers: Principles, Techniques and Tools**. Addison-Wesley, 1986.

[3] M. Arnold, D. Fink, M. Hind, and P. Sweeney. **Adaptive Optimization in the Jalapeño JVM**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, USA, October 2000.

[4] H. Baker. **CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A <1>**. SIGPLAN *Notices*, 30(9):17–20, September 1995.

[5] N. Benton and G. Kennedy, A. Russel. **Compiling Standard ML to Java Bytecodes**. In *Int'l Conf. on Functional Programming*, 1998.

[6] H.J. Boehm and M. Weiser. **Garbage Collection in an Uncooperative Environment**. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[7] P. Botner. **Kawa: Compiling Scheme to Java**. In *Lisp users conference*, Berkeley, California, USA, November 1998.

[8] W. De Pauw and G. Sevitski. **Visualizing Reference Patterns for Solving Memory Leaks in Java**. In *Proceedings ECOOP'99*, pages 116–134, Lisbon, Portugal, June 1999.

[9] M. DePristo. **SINTL: A Strongly-Typed Generic Intermediate Language for Scheme**. Northwestern University, Computer Science Honors Thesis, 2000.

[10] P. H. Hartel *et al.* **Pseudoknot: a Float-Intensive Benchmark for Functional Compilers**. *Journal of Functional Programming*, 6(4):621–655, 1996.

[11] M. Feeley, J. Miller, G. Rozas, and J. Wilson. **Compiling Higher-Order Languages into Fully Tail-Recursive Portable C**. Technical Report Rapport technique 1078, Université de Montréal, Département d'informatique et r.o., August 1997.

[12] E. Gagnon and L. Hendren. **SableVM: A Research Framework for the Efficient Execution of Java Bytecode**. Technical Report 2000-3, McGill University, School of Computer Science, November 2000.

[13] D. Gudeman. **Representing Type Information in Dynamically Typed Languages**. Technical report, University of Arizona, Departement of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, AZ 85721, April 1993.

[14] U. Hölzle. **Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming**. PhD thesis, Stanford University, August 1994.

[15] U. Hölzle and D. Ungar. **Reconciling responsiveness with performance in pure object-oriented languages**. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[16] M. Honeyford. **Weighing in on Java native compilation**. Technical Report developerWorks, IBM, January 2002.

[17] C.-H. Hsieh, J. Gyllenhaal, and W. Hwu. **Java bytecode to native code translation: The Caffeine prototype and preliminary results**. In *IEEE/ACM Int'l Symposium on Microarchitecture*, 1996.

[18] R. Kelsey, W. Clinger, and J. Rees. **The Revised(5) Report on the Algorithmic Language Scheme**. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

[19] A. Krall. **Efficient JavaVM Just-in-Time Compilation**. In *Proceedings PACT'98*, Paris, France, October 1998.

[20] X. Leroy. **Unboxed objects and polymorphic typing**. In *Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.

[21] T. Lindholm and F. Yellin. **The Java Virtual Machine**. Addison-Wesley, 1996.

[22] O. Pinali Doederlein. **The Java Performance Report – Part IIA**. http://www.javalobby.org/features/jpr, August 2000.

[23] C. Queinnec. **The influence of browsers on evaluators**. In *Int'l Conf. on Functional Programming*, pages 23–33, Montréal, Canada, September 2000.

[24] M. Schinz and M. Odersky. **Tail call elimination of the Java Virtual Machine**. In *Proceedings of Babel'01*, Florence, Italy, September 2001.

[25] N. Séniak. **Théorie et pratique de Sqil: un langage intermédiaire pour la compilation des langages fonctionnels**. PhD thesis, Université Pierre et Marie Curie (Paris VI), November 1991.

[26] M. Serrano. **Control Flow Analysis: a Functional Languages Compilation Paradigm**. In *10th Symposium on Applied Computing*, pages 118–122, Nashville, Tennessee, USA, February 1995.

[27] M. Serrano. **Inline expansion: *when* and *how*?** In *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, pages 143–147, Southampton, UK, September 1997.

[28] M. Serrano and M. Feeley. **Storage Use Analysis and its Applications**. In *1fst Int'l Conf. on Functional Programming*, pages 50–61, Philadelphia, Penn, USA, May 1996.

[29] M. Serrano and P. Weis. **Bigloo: a portable and optimizing compiler for strict functional languages**. In *2nd Static Analysis Symposium*, Lecture Notes on Computer Science, pages 366–381, Glasgow, Scotland, September 1995.

[30] O. Shivers. **Control Flow Analysis in Scheme**. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[31] T. Suganama, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. **Overview of the IBM Java Just-in-time compiler**. *IBM Systems Journal*, 39(1), 2000.

[32] Sun Microsystems. **The Java HotSpot Performance Engine**, April 1999.

[33] D. Tarditi, A. Acharya, and P. Lee. **No assembly required: Compiling Standard ML to C**. *ACM Letters on Programming Languages and Systems*, 2(1):161–177, 1992.

[34] Tolksdorf. **Compiling to JVM**. `http://grunge.cs.tu-berlin.de/~tolk`, 2000.

[35] T. VanDrunen, A. Hosking, and J. Palsberg. **Reducing loads and stores in stack architectures**, 2000.

[36] D. Wakeling. **Compiling Lazy Functional Programs for the Java Virtual Machine**. *Journal of Functional Programming*, 9(6):579–603, November 1999.

	CPU+SYS seconds					
Bench	Bigloo C	BiglooJvm	Gambit	Kawa	MLj	Javac
Bague	**10.12** (1.00 δ)	15.88 (1.56 δ)	*69.05* (6.82 δ)	_	_	13.39 (1.32 δ)
Beval	**10.61** (1.00 δ)	15.30 (1.44 δ)	22.18 (2.09 δ)	*1890.9* (178.22 δ)	_	_
Bigloo	**3.89** (1.00 δ)	*12.85* (3.30 δ)	_	_	_	_
Boyer	**12.73** (1.00 δ)	24.28 (1.90 δ)	31.82 (2.50 δ)	*89.17* (7.00 δ)	_	_
Cgc	**2.35** (1.00 δ)	*7.43* (3.16 δ)	_	_	_	_
Conform	**11.01** (1.00 δ)	51.18 (4.64 δ)	*59.33* (5.38 δ)	_	_	_
Earley	**10.51** (1.00 δ)	31.81 (3.02 δ)	65.75 (6.25 δ)	*452.08* (43.01 δ)	_	_
Fft	**8.96** (1.00 δ)	17.92 (2.00 δ)	650.46 (72.60 δ)	*2474.9* (276.22 δ)	_	_
Fib	13.58 (1.00 δ)	**8.49** (0.62 δ)	14.59 (1.07 δ)	*1905.4* (140.31 δ)	19.65 (1.44 δ)	12.03 (0.88 δ)
Leval	**9.26** (1.00 δ)	18.94 (2.04 δ)	*40.73* (4.39 δ)	37.21 (4.01 δ)	_	_
Maze	**13.87** (1.00 δ)	*43.74* (3.15 δ)	39.58 (2.85 δ)	_	_	_
Mbrot	**13.72** (1.00 δ)	55.85 (4.07 δ)	347.95 (25.36 δ)	*8090.0* (589.65 δ)	_	44.98 (3.27 δ)
Nucleic	**9.12** (1.00 δ)	12.14 (1.33 δ)	*21.82* (2.39 δ)	_	11.58 (1.27 δ)	_
Peval	**14.50** (1.00 δ)	62.77 (4.32 δ)	36.20 (2.49 δ)	*89.50* (6.17 δ)	_	_
Puzzle	**13.13** (1.00 δ)	20.20 (1.53 δ)	20.62 (1.57 δ)	*4322.4* (329.21 δ)	_	_
Queens	**12.22** (1.00 δ)	12.38 (1.01 δ)	33.13 (2.71 δ)	*571.82* (46.79 δ)	17.09 (1.39 δ)	_
Qsort	19.38 (1.00 δ)	19.74 (1.01 δ)	190.66 (9.83 δ)	*6611.7* (341.17 δ)	36.15 (1.86 δ)	**18.92** (0.97 δ)
Rgc	**10.64** (1.00 δ)	*110.45* (10.38 δ)	_	_	_	_
Sieve	**11.88** (1.00 δ)	16.06 (1.35 δ)	22.05 (1.85 δ)	*173.34* (14.59 δ)	15.76 (1.32 δ)	17.61 (1.48 δ)
Slatex	**13.91** (1.00 δ)	61.81 (4.44 δ)	38.44 (2.76 δ)	*69.69* (5.01 δ)	_	_
Traverse	**18.68** (1.00 δ)	50.69 (2.71 δ)	*113.06* (6.05 δ)	107.89 (5.77 δ)	_	_

Figure 9: Benchmarks timing on an AMD Tbird 800Mhz/256MB, running Linux 2.2.8

Appendix A: The benchmarks

Here is a short description of the benchmarks we have been using so far. The numbers of lines are always given for the Bigloo version of the source files. Figure 9 presents all the numerical values on Linux/x86.

•**Bague** by *P. Weis* (105 lines). Tests fixnum arithmetic and vectors. •**Beval** (582 lines). The regular Bigloo Scheme evaluator. •**Bigloo** (99,376 lines). The bootstrap of the BiglooC compiler and the runtime library. •**Boyer** (626 lines) by *B. Boyer* and modified by *B. Shaw* and *W. Clinger*. Tests symbols and conditional expressions. •**Cgc** (8,128 lines). A simple compiler for a C like language that produces Mips assembly code. •**Conform** (596 lines). It uses lists, vectors and numerous small inner functions. •**Earley** by *M. Feeley* (672 lines). An implementation of the Earley parser. •**Fft** (120 lines). Yet another Gabriel's benchmark. Fast Fourier transform ported to Scheme by *Harry Barrow*. •**Fib** (18 lines). Fibonacci numbers. •**Leval** by *M. Feeley* (555 lines). A Scheme evaluator using lambda expressions. •**Maze** by *O. Shivers* (809 lines). Uses arrays fixnum operations and iterators. •**Mbrot** (47 lines). The Mandlebrot curve that tests floating point arithmetic. •**Nucleic** (3,507 lines). Described in [10], this benchmark measures the efficiency of numerical computations. •**Peval** by *M. Feeley* (639 lines). A partial evaluator that uses a lot of nested functions and allocates many lists and symbols. •**Puzzle** by *F. Baskett* (208 lines). Another Gabriel's benchmark. •**Queens** by *L. Augustsson* (131 lines). Ported from Lml to Scheme, tests list allocations. •**Quicksort** by *P. Weis* (124 lines). It tests arrays and fixnum arithmetic. •**Rgc** (348 lines). The Bigloo regular grammar that implements the Bigloo reader. •**Sieve** (53 lines) Fixnum arithmetic and list allocations. •**Slatex** by *D. Sitaram* (2,827 lines). This is a LaTeX preprocessor implemented by Rice University that tests Input/Output capacities. •**Traverse** modified by *J. Siskind* (136 lines). It allocates and modifies lists.