Compiling Scheme programs to .NET Common Intermediate Language

Yannis Bres, Bernard Paul Serpette, Manuel Serrano
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93, F-06902 Sophia Antipolis, Cedex, France
{Yannis.Bres,Bernard.Serpette,Manuel.Serrano}@inria.fr

ABSTRACT

This paper presents the compilation of the Scheme programming language to .NET platform. .NET provides a virtual machine, the Common Language Runtime (CLR), that executes bytecode: the Common Intermediate Language (CIL). Since CIL was designed with language agnosticism in mind, it provides a rich set of language constructs and functionalities. Therefore, the CLR is the first execution environment that offers type safety, managed memory, tail recursion support and several flavors of pointers to functions. As such, the CLR presents an interesting ground for functional language implementations.

We discuss how to map Scheme constructs to CIL. We present performance analyses on a large set of real-life and standard Scheme benchmarks. In particular, we compare the performances of these programs when compiled to C, JVM and .NET. We show that .NET still lags behind C and JVM.

1. INTRODUCTION

Introduced by Microsoft in 2001, the .NET Framework has many similarities with the Sun Microsystems Java Platform [9]. The execution engine, the Common Language Runtime (CLR), is a stack-based Virtual Machine (VM) which executes a portable bytecode: the Common Intermediate Language (CIL) [8]. The CLR enforces type safety through its bytecode verifier (BCV), it supports polymorphism, the memory is garbage collected and the bytecode is Just-In-Time [1,17] compiled to native code.

Beyond these similarities, Microsoft has designed the CLR with language agnosticism in mind. Indeed, the CLR supports more language constructs than the JVM: the CLR supports enumerated types, structures and value types, contiguous multidimensional arrays, etc. The CLR supports tail calls, i.e. calls that do not consume stack space. The CLR supports closures through delegates. Finally, pointers to functions can be used although this leads to unverifiable bytecode. The .NET framework has 4 publicly available implementations:

- From Microsoft, one commercial version and one whose sources are published under a shared source License (Rotor [16]). Rotor was released for research and educational purposes. As such, Rotor JIT and GC are simplified and stripped-down versions of the commercial CLR, which lead to poorer performances.
- Ximian/Novell's Mono Open Source project offers a quite complete runtime and good performances but has only a few compilation tools.
- From DotGNU, the Portable.Net GPL project provides a quite complete runtime and many compilation tools. Unfortunately, it does not provide a full-fledged JIT [20]. Hence, its speed cannot compete with other implementations so we will not show performance figures for this platform.

1.1 Bigloo

Bigloo is an optimizing compiler for the Scheme (R^5RS [7]) programming language. It targets C code, JVM bytecode and now .NET CIL. In the rest of this presentation, we will use BiglooC, BiglooJvm, and Bigloo.NET to refer to the specific Bigloo backends. Benchmarks show that BiglooC generates C code whose performance is close to human-written C code. When targeting the JVM, programs run, in general, less than 2 times slower than C code on the best JDK implementations [12].

Bigloo offers several extensions to Scheme [7] such as: modules for separate compilation, object extensions *à la* Clos [3] + extensible classes [14], optional type annotations for compile-time type verification and optimization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4
© UNION Agency - Science Press, Plzen, Czech Republic

Bigloo is itself written in Bigloo and the compiler is bootstrapped on all of its three backends. The runtime is made of 90% of Bigloo code and 10% of C, Java, or C# for each backend.

1.2 Motivations

As for the JVM, the .NET Framework is appealing for language implementors. The runtime offers a large set of libraries, the execution engine provides a lot of services and the produced binaries are expected to run on a wide range of platforms. Moreover, we wanted to explore what the "more language-agnostic" promise can really bring to functional language implementations as well as the possibilities for language interoperability.

1.3 Outline of this paper

Section 2 presents the main techniques used to compile Bigloo programs to CIL. Section 3 enumerates the new functionalities of the .NET Framework that could be used to improve the performances of produced code. Section 4 compares the run times of several benchmark and real life Bigloo programs on the three C, JVM and .NET backends.

2. COMPILATION OUTLINE

This section presents the general compilation scheme of Bigloo programs to .NET CIL. Since CLR and JVM are built upon similar concepts, the techniques deployed for these two platforms are close. The compilation to JVM being thoroughly presented in a previous paper [12], only a shallow presentation is given here.

2.1 Data Representation

Scheme polymorphism implies that, in the general case, all data types (numerals, characters, strings, pairs, etc.) have a uniform representation. This may lead to boxing values such as numerals and characters, i.e., allocating heap cells pointing to numbers and characters. Since boxing reduces performances (because of additional indirections) and increase memory usage, we aim at avoiding boxing as much as possible. Thanks to the Storage Use Analysis [15] or user-provided type annotations, numerals or characters are usually passed as values and not boxed, i.e. not allocated in the heap any more. Note that in the C backend, boxing of integers is always avoided using usual tagging techniques [6]. In order to save memory and avoid frequent allocations, integers in the range [-100 ... 2048] and all 256 characters (objects that embed a single byte) are preallocated. Integers are represented using the `int32` type. Reals are represented using `float64`. Strings are represented by arrays of `bytes`, as Scheme strings are mutable sequences of 1 byte characters while the .NET built-in `System.Strings` are non-mutable sequences of wide characters. Closures are instances of `bigloo.procedure`, as we will see in Section 2.3.3.

2.2 Separate Compilation

A Bigloo program is made of several modules. Each module is compiled into a CIL class that aggregates the module definitions as static variables and functions. Modules can define several classes. Such classes are compiled as regular CIL classes (see §2.3.4). Since we do not have a built-in CIL assembler yet, we print out each module class as a file and use the Portable.Net assembler to produce an object file. Once all modules have been separately compiled, they are linked using the Portable.NET linker.

2.3 Compilation of functions

Functions can be separated in several categories:

- Local tail-recursive functions that are not used as first-class values are compiled as loops.
- Non tail-recursive functions that are not used as first-class values are compiled as static methods.
- Functions used as first-class values are compiled as real closures. A function is used as a first-class value when it is used in a non-functional position, i.e., used as an argument of another function call or used as a return value of another function.
- Generic functions are compiled as static methods and use an *ad hoc* framework for resolving late binding.

2.3.1 Compiling tail-recursive functions

In order to avoid the overhead of function calls, local functions that are not used as values and always called tail-recursively are compiled into CIL loops. Here is an example of two mutually recursive functions:

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

These functions are compiled as:

```
.method static bool odd(int32) cil managed {
  .locals(int32)
         ldarg.0       // load arg
  odd?:  stloc.0       // store in local var #0
         ldloc.0       // load local var #0
         ldc.i4.0      // load constant 0
         brne.s loop1  // if not equal go to loop1
         ldc.i4.0      // load constant 0 (false)
         br.s end      // go to end
  loop1: ldloc.0       // load local var #0
         ldc.i4.1      // load constant 1
         sub           // subtract
  even?: stloc.0       // store in local var #0
         ldloc.0       // load local var #0
         ldc.i4.0      // load constant 0
         brne.s loop2  // if not equal go to loop2
         ldc.i4.1      // load constant 1 (true)
         br.s end      // go to end
```

```
  loop2: ldloc.0      // load local var #0
         ldc.i4.1     // load constant 1
         sub          // subtract
         br.s odd?    // go to odd?
  end:   ret          // return
}
```

2.3.2 Compiling regular functions

As a more general case, functions that cannot be compiled to loops are compiled as CIL static methods. Consider the following Fibonacci function:

```
(define (fib n::int)
  (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2)))))
```

It is compiled as:

```
.method static int32 fib(int32) cil managed {
         ldarg.0        // load arg
         ldc.i4.2       // load constant 2
         bne.s loop     // if not equal go to loop
         ldc.i4.1       // load constant 1
         br.s end       // go to end
  loop:  ldarg.0        // load arg
         ldc.i4.1       // load constant 1
         sub            // subtract
         call int32 fib::fib(int32)
         ldarg.0        // load arg
         ldc.i4.2       // load constant 2
         sub            // subtract
         call int32 fib::fib(int32)
         add            // add
  end:   ret            // return
}
```

Note also that if their body is sufficiently small, these functions might get inlined (see [13]).

2.3.3 Compiling closures

Functions that are used as first-class values (passed as argument, returned as value or stored in a data structure) are compiled to closures.

The current closure compilation scheme for the JVM and .NET backends comes from two *de facto* limitations imposed by the JVM. First, the JVM does not support pointers to functions. Second, as to each class corresponds a file, we could not afford to declare a different type for each closure. We estimated that the overload on the class loader would raise a performance issue for programs that use closures intensively. As an example of real-life program, the Bigloo compiler itself is made of 289 modules and 175 classes, which produce 464 class files. Since we estimate that the number of real closures is greater than 4000, compiling each closure to a class file would multiply the number of files by more than 10.

In JVM and .NET classes corresponding to Bigloo modules extend `bigloo.procedure`. This class declares the arity of the closure, an array of captured variables, two kind of methods (one for functions with fixed arity and one for functions with variable arity), and the index of the closure within the module that defines it. In order to have a single type to represent closures, all the closures of a single module share the same entry-point function. This function uses the index of the closure to call the body of the closure, using a `switch`. Closure bodies are implemented as static methods of the class associated to the module and they receive as first argument the `bigloo.procedure` instance.

The declaration of `bigloo.procedure` is similar to:

```
class procedure {
  int index, arity;
  Object[] env;
  virtual Object funcall0();
  virtual Object funcall1(Object a1);
  virtual Object funcall2(Object a1, Object a2);
  ...
  virtual Object apply(Object as);
}
```

Let's see that in practice with the following program:

```
(module klist)
(define (make-klist n) (lambda (x) (cons x n)))
(map (make-adder 10) (list 1 2 3 4 5))
```

The compiler generates a class similar to:

```
class klist: procedure {
  static procedure closure0
    = new make-klist(0, 1, new Object[] {10});
  make-klist(int index, int arity, Object[] env) {
    super(index, arity, env);
  }
  ...
  override Object funcall1(Object arg) {
    switch (index) {
      case 0: return anon0(this, arg);
      ...
    }
  }
  ...
  static Object anon0(procedure fun, Object arg) {
    return make-pair(arg, fun.env[0]);
  }
  static void Main() {
    map(closure0, list(1, 2, 3, 4, 5));
  }
}
```

2.3.4 Compiling Generic Functions

The Bigloo object model [14] is inspired from Clos [3]: classes only encapsulate data, there is no concept of visibility. Behavior is implemented through generic functions, called generics, which are overloaded global methods whose dispatch is based on the dynamic type of their arguments. Contrarily to Clos, Bigloo only supports single inheritance, single dispatch. Bigloo does not support the Clos Meta Object Protocol.

In both JVM and CLR, the object model is derived from Smalltalk and C++: classes encapsulate data and behaviour, implemented in methods which can have different visibility levels. Method dispatch is based on the dynamic type of objects on which they are applied. Classes can be derived and extended with new data slots, methods can be redefined and

new methods can be added. Only single inheritance is supported for method implementation and instance variables, while multiple inheritance is supported for method declarations (interfaces).

Bigloo classes are first assigned a unique integer at run-time. Then, for each generic a dispatch table is built which associates class indexes to generic implementations, when defined. Note that class indexes and dispatch tables cannot be built at compile-time for separate compilation purposes. When a generic is invoked, the class index of the first argument is used as a lookup value in the dispatch table associated with the generic. Since these dispatch tables are usually quite sparse, we introduce another indirection level in order to save memory.

Whereas C does not provide direct support for any evolved object model, JVM or CLR do and we could have used the built-in virtual dispatch facilities. However, this would have lead to serious drawbacks. First, as generics are declared for all objects, they would have to be declared in the superclass of all Bigloo classes. As a consequence, separate compilation would not be possible any more. Moreover, this would lead to huge virtual function tables for all the Bigloo classes, with the corresponding memory overhead. Finally, the framework we chose has two main advantages: it is portable and it simplifies the maintenance of the system. For these reasons, the generic dispatch mechanism is similar in the C, JVM and .NET backends.

2.4 Continuations

Scheme allows to capture the continuation of a computation which can be used to escape pending computations, but it can also be used to suspend, resume, or even *restart* them! If in the C backend, continuations are fully supported using `setjmp`, `longjmp` and `memcpy`, in JVM and CLR, the stack is read-only and thus cannot be restored. Continuation support is implemented using structured exceptions. As such, continuations are first-class citizens but they can only be used within the dynamic extent of their capture.

One way to implement full continuation support in JVM and CLR would be to manage our own call stack. However, this would impose to implement a complex protocol to allow Bigloo programs to call external functions, while this is currently trivial. Moreover, we could expect JITs to be far less efficient on code that manages its own stack. Doing so would thus reduce performances of Bigloo programs, which seems unacceptable for us. Therefore, we chose not to be fully R^5RS compliant on this topic.

3. .NET NEW FUNCTIONALITIES

In this section we explore the compilation of Scheme with CIL constructs that have no counterpart in the JVM.

3.1 Closures

If we consider the C implementation of closures as a performance reference, the current JVM and .NET implementations have several overheads:

- The cost of body dispatching depending on closure index (in the C backend pointers to functions are directly available).
- An additional indirection when accessing a captured variable in the array (in the C backend, the array is inlined in the C structures representing the closures).
- The array boundaries verification (which are not verified at run-time in the C compiled code).

The CLR provides several constructs that can be used to improve the closure compilation scheme: delegates, declaring a new class per closure, and pointers to functions [18]. We have not explored this last possibility because it leads to unverifiable code.

3.1.1 Declaring a new type for each closure

Declaring a new type for each closure, as presented in §2.3.3, would get rid of the indexed function call and enables inlining of captured variables within the class instead of storing them in an array. However, as we have seen, each JVM class is stored in its own file and there are more than *4000* closures in the compiler. Hence, we could not afford to declare a new class for each closure in the JVM backend: loading the closures would be too much of a load for the class loader.

This constraint does not hold in the .NET Framework as types are linked at compile-time within a single binary file. However, loading a new type in the system is a costly operation: metadata have to be loaded, their integrity verified, etc. Moreover we noted that each closure would add slightly more than 100 bytes of metadata in the final binary file, that is about more than 400Kb for a binary which currently weights about 3.8MB, i.e. a size increase of more than 10%.

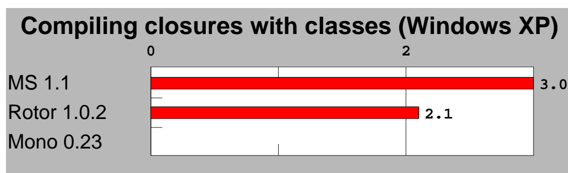

Fig. 1: Declaring a class per closure. This test compares the performance of two techniques for invoking closures: ***declaring a type per closure*** **and** ***indexed functions.*** **Lower is better.**

We have written a small benchmark program that declares 100 modules containing 50 closures each. For

each module, the program calls 10 times each 50 closures in a row. All closure functions are the identity, so this program focuses on the cost of closure invocations. Figure 1 shows that such a program always runs at least twice slower when we define a new type for each closure (Mono crashes on this test). Note that if the closures were invoked more than 10 times, these figures would decrease as the time wasted when loading the new types would be compensated by the acceleration of closure invocations. However, declaring a new type for each closure does not seem to really be a good choice for performances.

3.1.2 *Using Delegates*

The CLR provides a direct support for the Listener Design Pattern through Delegates which are linked lists of couples <*object reference*, *pointer to method*>. Delegates are a restricted form of pointers to functions that can be used in verifiable code while real pointer to functions lead to unverifiable code. Declaring delegates involves two steps. First, a delegate is declared. Second, methods whose signature match its declaration are registered. This is illustrated by the following example:

```
delegate void SimpleDelegate( int arg );
void MyFunction( int arg ) {...}
SimpleDelegate d;
d = new SimpleDelegate( MyFunction );
```

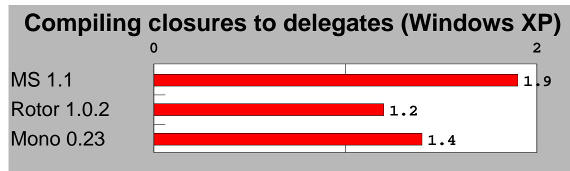

Fig. 2: Compiling closures to delegates. This test compares the performance of two techniques for invoking closures: *delegates* and *indexed functions*.

Figure 2 shows that our closure simulation program also runs slower when using delegates as surrogate pointers to functions instead of the indexed call. Such a result is probably due to the fact that delegates are linked lists of pointers to methods where we would be satisfied by single pointers to methods.

3.2 Tail Calls

The R^{5}RS requires that functions that are invoked tail-recursively must not allocate stack frames. In C and Java, tail recursion is not directly feasible because these two languages does not support it. The trampoline technique [2,19,5,11] allows tail-recursive calls. Since it imposes a performance penalty, we have chosen not to use it for Bigloo. As such, the Bigloo C and JVM backends are only partially compliant with the R^{5}RS on this topic.

In the CIL, function calls that precede a return instruction can be flagged as tail-recursive. In this case, the current stack frame is discarded before jumping to the tail-called function. The CLR is the first architecture considered by Bigloo that enables correct support of tail-recursive calls. For the .NET code generator, we have added a flag that enables or disables the CIL .tail call annotation. Hence, we have been able to measure, as reported Figure 3, the impact of tail calls on the overall performance of Bigloo programs (see §6 for a brief description of the benchmarks used). As demonstrated by this experiment, the slowdown imposed by flagging tail calls is generally small. However, some programs are severely impacted by tail recursion. In particular, the Bague program runs 5 times slower when tail recursion is enabled! This toy benchmark essentially measures function calls. It spends its whole execution time in a recursion made of 14 different call sites amongst which 6 are tail calls. This explains why this program is so much impacted by tail recursion.

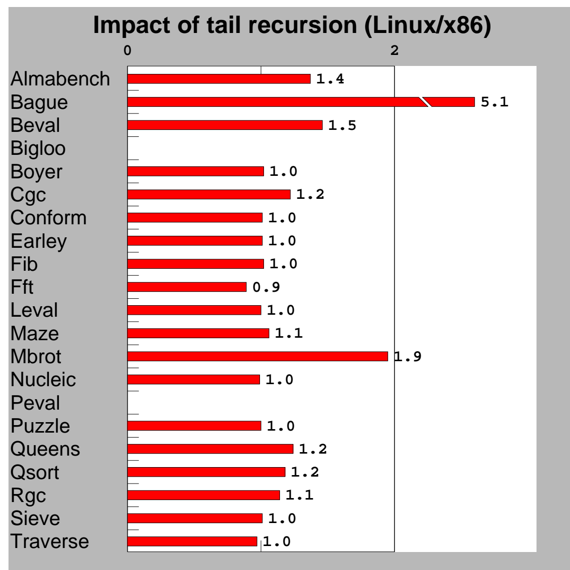

Fig. 3: This test measures the impact of tail recursion on .NET executions. Scores are relative to Bigloo.NET, which is the 1.0 mark. Lower is better.

The tail-call support of the .NET platform is extremely interesting for porting languages such as Scheme. However, since tail calls may slow down performance, we have decided not to flag tail calls by default. Instead we have provide the compiler with three options. One enabling tail-calls inside modules, one enabling them across modules, and a last one enabling them for all functions, including closures.

3.3 Precompiling binaries

With some .NET platforms, assemblies (executables and dynamic libraries) can be precompiled once for all. This removes the need for just-in-time compiling assemblies at each program launch and enables

heavier and more expensive program optimizations. Precompiled binaries are specifically optimized for the local platform and tuned for special features provided by the processor. Note that the original portable assemblies are not replaced by optimized binary versions. Instead, binary versions of assemblies are maintained in a cache. Since .NET assemblies are versioned, the correspondance between original portable assemblies and precompiled ones is straightforward. When an assembly is looked up by the CLR, preference is then given to a precompiled one of compatible version, when available.

Even if precompiling binaries is a promising idea for realistic programs such as Bigloo and Cgc (a simple C-like compiler), we have unfortunately measured no improvement using it. Even worse, we have even noticed that when precompiled these programs actually run slower!

4. PERFORMANCE EVALUATIONS

We have used a large set of benchmarks for estimating the performance of the .NET CLR platform. They are described in Figure §6, which also describes the platform we have used for these experiments. For measuring performance, we have used Mono .NET because it is the only implementation that is available on all main operating systems and because it delivers performance comparable to that of Microsoft CLR (when ran on Windows).

4.1 Bigloo vs C#

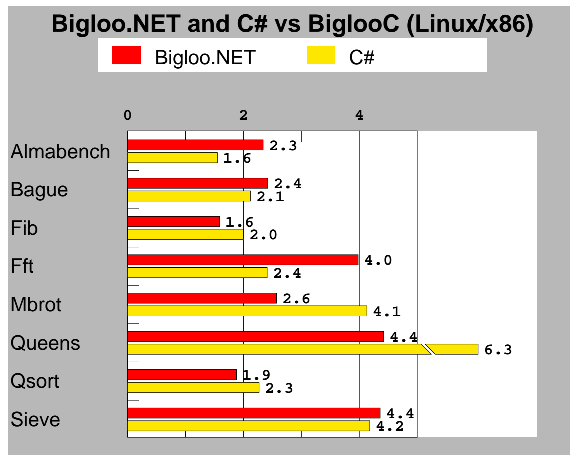

Fig. 4: This test compares the performance of Bigloo.NET vs C#. Scores are relative to BiglooC, which is the 1.0 mark. Lower is better.

To assess the quality of the CIL code produced by the Bigloo.NET compiler, we have compared the running times of the Bigloo generated code vs. regular human-written code in C# on a subset of our programs made of micro benchmarks that were possible to translate. For this experiment we use *managed* CIL code. That is, bytecode that complies the byte code verification rules of the CLR. Figure 4 shows that most Bigloo compiled programs have performances that are quite on par with their C# counterparts, but for Almabench and Fft. Actually the Bigloo version of these two benchmarks suffer from the same problem. Both benchmarks are floating point intensive. The Bigloo type inference is not powerful enough to get rid of polymorphism for these programs. Hence, many allocations of floating point numbers take place at run-time, which obviously slows down the overall execution time.

4.2 Platform and backend benchmarks

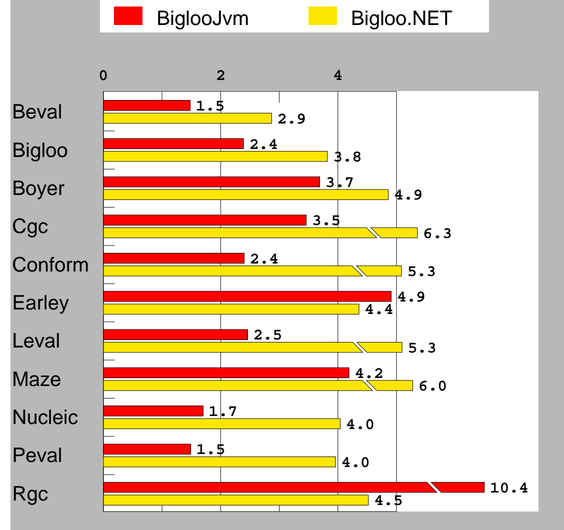

Fig. 5: This test compares BiglooJVM and Bigloo.NET. Scores are relative to BiglooC, which is the 1.0 mark. Lower is better.

Figure 5 shows the running times of several real-life and standard Scheme benchmarks for all three Bigloo backends. Since we are comparing to native code where no verification takes place, we have decided to measure the performance of *unmanaged* CIL bytecode and JVM bytecode that is not conform to the JVM bytecode verifier. (Figure 7 presents figures for *unmanaged* and *managed* CIL bytecode.)

In general, Bigloo.NET programs run from 1.5 to 2 times slower than their BiglooJvm counterpart. The only exceptions are Earley and Rgc for which Bigloo-.NET is faster. These two programs are also the only ones for which the ratio BiglooJvm/BiglooC is greater than 4. Actually these two programs contain patterns that cause trouble to Sun's JDK1.4.2 JIT used for this experiment. When another JVM is used, such as the one from IBM, these two programs run only twice slower than their BiglooC counterpart.

The benchmarks test memory allocation, fixnum and flonum arithmetics, function calls, etc. For all these topics, the figures show that the ratio between Bigloo-

Jvm and Bigloo.NET is stable. This uniformity shows that BiglooJVM avoids traps introduced by JIT*ted* architectures [12]. The current gap between Jvm and .NET performance is most likely due to the youth of .NET implementations. After all, JVM benefits from 10 years of improvements and optimizations. We also remember the time where each new JVM was improving performance by a factor of two!

4.2.1 Impact of the memory management

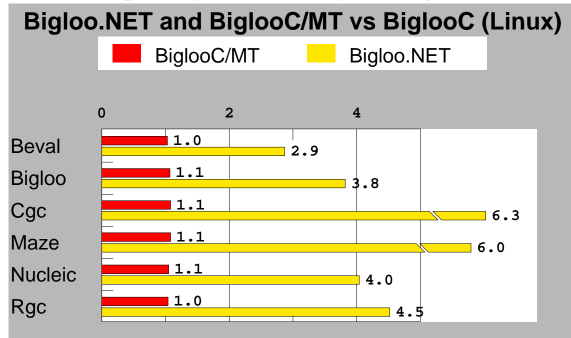

Fig. 6: This test measures the impact of multi-threading.

Both BiglooC (native) runtime system and Mono VM use the garbage collector developed by H-J Boehm [4]. However, as reported in Section 2.1, BiglooC uses traditional C techniques for minimizing the memory space of frequently used objects such as pairs or integers. In addition, BiglooC tunes the Boehm's collector for single threaded applications while Mono tunes it for multi-threading. In order to measure the impact of the memory management on performance, we have compiled a special BiglooC native version, called BiglooC/MT that used the very same collector as the mono one. As reported on Figure 6 BiglooC/MT is no more than 10% slower than BiglooC on real benchmarks. Therefore, we can conclude that memory management is not the culprit for the weaker performance of Bigloo.NET.

4.3 Related Work

Besides Bigloo, several projects have been started to provide support for Scheme in the .NET Framework. (*i*) Dot-Scheme [10] is an extension of PLT Scheme that gives PLT Scheme programs access to the Microsoft .NET Framework. (*ii*) Scheme.NET , from the Indiana University. (*iii*) Scheme.NET , from the Indiana University. (*iv*) Hotdog, from Northwestern University. Unfortunately we have failed to install these systems under Linux thus we do not present performance comparison in this paper. However, from the experiments we have conducted under Windows it appears that none of these systems has been designed and tuned for performance. Hence, they have a different goal from Bigloo.NET.

Beside Scheme, there are two main active projects for functional language support in .NET: (*i*) From Microsoft Research, F# is an implementation of the core of the CAML programming language.(*ii*) From Microsoft Research and the University of Cambridge, SML.NET is a compiler for Standard ML that targets the .NET CLR and which supports language interoperability features for easy access to .NET libraries. Unfortunately, as for the Scheme systems described above, we have failed in installing these two systems on Linux. Hence, we cannot report performance comparison in this paper.

5. CONCLUSIONS

We have presented the new .NET backend of Bigloo, an optimizing compiler for a Scheme dialect. This backend is fully operational. The whole runtime system has been ported to .NET and the compiler bootstraps on this platform. With the exception of continuations, the .NET backend is compliant to Scheme R^{5}RS. In particular, it is the first Bigloo backend that handles tail-recursive calls correctly. Bigloo.NET is available at: `http://www.inria.fr/mimosa/fp/-Bigloo`.

In conclusion, most of the new functionalities of the .NET Framework are still disappointing if we only consider performances as the ultimate objective. On the other hand, the support for tail calls in the CLR is very appealing for implementing languages that require proper tail-recursion. Currently .NET performance has not reached the one of Jvm implementation: Bigloo.NET programs run significantly slower than BiglooC and BiglooJvm programs. However there seems to be strong activity in the community of .NET implementors. Future will tell if next versions of .NET will bridge the gap with JVM implementations.

Bibliography

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

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

[3] Bobrow, D. and DeMichiel, L. and Gabriel, R. and Keene, S. and Kiczales, G. and Moon, D. – **Common lisp object system specification** – special issue, Sigplan Notices, (23), Sep, 1988.

[4] Boehm, H.J. – **Space Efficient Conservative Garbage Collection** – Conference on Programming Language Design and Implementation, Sigplan Notices, 28(6), 1993, pp. 197–206.

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

Bench	Bigloo	BiglooJvm	BiglooJvm (vrf)	Bigloo.NET	Bigloo.NET (mgd)	Bigloo.NET (tailc)
	Wall clock time in seconds					
Almabench	**5.54** (1.0 δ)	10.29 (1.85 δ)	*20.96* (3.78 δ)	8.72 (1.57 δ)	12.99 (2.34 δ)	12.01 (2.16 δ)
Bague	**4.71** (1.0 δ)	7.51 (1.59 δ)	7.61 (1.61 δ)	11.52 (2.44 δ)	11.42 (2.42 δ)	*58.81* (12.48 δ)
Beval	**5.98** (1.0 δ)	8.88 (1.48 δ)	9.17 (1.53 δ)	17.2 (2.87 δ)	24.16 (4.04 δ)	*25.2* (4.21 δ)
Bigloo	**19.2** (1.0 δ)	45.91 (2.39 δ)	46.34 (2.41 δ)	73.48 (3.82 δ)	*84.59* (4.40 δ)	*error*
Boyer	**8.43** (1.0 δ)	31.14 (3.69 δ)	30.61 (3.63 δ)	41.03 (4.86 δ)	*57.07* (6.76 δ)	41.9 (4.97 δ)
Cgc	**1.97** (1.0 δ)	6.82 (3.46 δ)	6.91 (3.50 δ)	12.4 (6.29 δ)	*19.26* (9.77 δ)	15.15 (7.69 δ)
Conform	**7.41** (1.0 δ)	17.82 (2.40 δ)	18.97 (2.56 δ)	39.4 (5.31 δ)	*48.44* (6.53 δ)	40.0 (5.39 δ)
Earley	**8.31** (1.0 δ)	40.86 (4.91 δ)	*41.91* (5.04 δ)	36.27 (4.36 δ)	40.61 (4.88 δ)	36.7 (4.41 δ)
Fib	**4.54** (1.0 δ)	7.32 (1.61 δ)	7.34 (1.61 δ)	7.27 (1.60 δ)	7.22 (1.59 δ)	*7.43* (1.63 δ)
Fft	**4.29** (1.0 δ)	7.8 (1.81 δ)	8.11 (1.89 δ)	15.26 (3.55 δ)	*17.1* (3.98 δ)	13.71 (3.19 δ)
Leval	**5.6** (1.0 δ)	13.8 (2.46 δ)	13.81 (2.46 δ)	29.91 (5.34 δ)	*35.11* (6.26 δ)	30.0 (5.35 δ)
Maze	**10.36** (1.0 δ)	43.5 (4.19 δ)	43.69 (4.21 δ)	62.18 (6.00 δ)	64.2 (6.19 δ)	*66.41* (6.41 δ)
Mbrot	**79.26** (1.0 δ)	199.26 (2.51 δ)	198.51 (2.50 δ)	205.9 (2.59 δ)	204.14 (2.57 δ)	*403.51* (5.09 δ)
Nucleic	**8.28** (1.0 δ)	14.1 (1.70 δ)	14.29 (1.72 δ)	33.53 (4.04 δ)	*37.85* (4.57 δ)	33.3 (4.02 δ)
Peval	**7.57** (1.0 δ)	11.28 (1.49 δ)	11.87 (1.56 δ)	30.01 (3.96 δ)	*32.47* (4.28 δ)	*error*
Puzzle	**7.59** (1.0 δ)	12.96 (1.70 δ)	13.03 (1.71 δ)	20.96 (2.76 δ)	*29.26* (3.85 δ)	21.0 (2.76 δ)
Queens	**10.47** (1.0 δ)	36.97 (3.53 δ)	37.75 (3.60 δ)	42.76 (4.08 δ)	46.37 (4.42 δ)	*53.33* (5.09 δ)
Qsort	**8.85** (1.0 δ)	13.26 (1.49 δ)	13.39 (1.51 δ)	16.93 (1.91 δ)	16.72 (1.88 δ)	*20.1* (2.27 δ)
Rgc	**6.94** (1.0 δ)	72.3 (10.41 δ)	*73.11* (10.53 δ)	31.43 (4.52 δ)	33.48 (4.82 δ)	35.96 (5.18 δ)
Sieve	**7.37** (1.0 δ)	25.44 (3.45 δ)	25.17 (3.41 δ)	31.01 (4.20 δ)	*32.19* (4.36 δ)	31.46 (4.26 δ)
Traverse	**15.19** (1.0 δ)	43.02 (2.83 δ)	43.24 (2.84 δ)	76.4 (5.02 δ)	*81.59* (5.37 δ)	74.21 (4.88 δ)

Fig. 7: Benchmarks timing on an AMD Tbird 1400Mhz/512MB, running Linux 2.4.21, Sun JDK 1.4.2, and Mono 0.30.

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

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

[8] Lidin, S. – **Inside Microsoft .NET IL Assembler** – *Microsoft Press*, 2002.

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

[10] Pinto, P. – **Dot-Scheme A PLT Scheme FFI for the .NET framework** – Scheme workshop, Boston, MA, USA, Nov, 2003.

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

[12] Serpette, B. and Serrano, M. – **Compiling Scheme to JVM bytecode: a performance study** – 7th Int'l Conf. on Functional Programming, Pittsburgh, Pensylvanie, USA, Oct, 2002.

[13] Serrano, M. – **Inline expansion: when and how** – Int. Symp. on Programming Languages, Implementations, Logics, and Programs, Southampton, UK, Sep, 1997, pp. 143–147.

[14] Serrano, M. – **Wide classes** – ECOOP'99, Lisbon, Portugal, Jun, 1999, pp. 391–415.

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

[16] Stutz, D. and Neward, T. and Shilling, G. – **Shared Source CLI Essentials** – *O'Reilly Associates*, March, 2003.

[17] Suganama, T. et al. – **Overview of the IBM Java Just-in-time compiler** – IBM Systems Journal, 39(1), 2000.

[18] Syme, D. – **ILX: Extending the .NET Common IL for Functional Language Interoperability** – Proceedings of Babel'01, 2001.

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

[20] Weatherley, R. and Gopal, V. – **Design of the Portable.Net Interpreter** – DotGNU, Jan, 2003.

6. APPENDIX: THE BENCHMARKS

Figure 7 presents all the numerical values on Linux 2.4.21/Athlon Tbird 1.4Ghz-512MB. Native code is compiled with Gcc 3.2.3. The JVM is Sun's JDK1.4.2. The .NET CLR is Mono 0.30. The JVM and CLR are multithreaded. Even single-threaded applications use several threads. In order to take into account the context switches implied by this technique we have preferred actual durations (wall clock) to CPU durations (user + system time). It has been paid attention to run the benchmarks on an unloaded computer. That is, the wall clock duration and the CPU duration of singled threaded C programs were the same.

Almabench (300 lines) floating point arithtmetic. **Bague** (105 l) function calls, fixnum arithtmetic, and vectors. **Beval** (582 l) the regular Bigloo Scheme evaluator. **Bigloo** (99,376 l) the bootstrap of the Bigloo compiler. **Boyer** (626 l) symbols and conditional expressions. **Cgc** (8,128 l) A simple C-to-mips compiler. **Conform** (596 l) lists, vectors and small inner functions. **Earley** (672 l) Earley parser. **Fft** (120 l) Fast Fourier transform. **Fib** (18 l) Fibonacci numbers. **Leval** (555 l) A Scheme evaluator using λ-expressions. **Maze** (809 l) arrays, fixnum operations and iterators. **Mbrot** (47 l) The Mandelbrot curve. **Nucleic** (3,507 l) floating point intensive compuations. **Peval** (639 l) A partial evaluator. **Puzzle** (208 l) A Gabriel's benchmark. **Queens** (131 l) tests list allocations. **Qsort** (124 l) arrays and fixnum arithmetic. **Rgc** (348 l) The Bigloo reader. **Sieve** (53 l) fixnum arithmetic and list allocations. **Slatex** (2,827 l) A LaTeX preprocessor. **Traverse** (136 l) allocates and modifies lists.