JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 9
Special issue: .NET Technologies 2004 workshop, Pilsen

Bigloo.NET: compiling Scheme to .NET CLR

Yannis Bres, INRIA Sophia-Antipolis, France
Bernard Paul Serpette, INRIA Sophia-Antipolis, France
Manuel Serrano, INRIA Sophia-Antipolis, France

This paper presents the compilation of the Scheme programming language to .NET. This platform 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. As such, the CLR is the first execution environment that offers type safety, managed memory, tail recursion support and several flavors of pointers to functions. Therefore, 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 speed of these programs when compiled to C, JVM and .NET. We show that in term of speed performance of the Mono implementation of .NET, the best implementing running on both Windows and Linux, still lags behind C and fast JVMs such as the Sun's implementations.

1 INTRODUCTION

Introduced by Microsoft in 2001, the .NET framework has many similarities with the Sun Microsystems Java Platform [11]. 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) [10]. 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 [1,21] 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 multi-dimensional arrays, etc. The CLR supports tail calls, i.e. calls that do not consume stack space. The CLR supports closures through delegates. At last, pointers to functions can be used although doing so 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 [20]. Rotor was released for research and educational purposes.

Cite this article as follows: Yannis Bres, Bernard Serpett, Manuel Serrano: "Bigloo.NET: compiling Scheme to .NET CLR", in *Journal of Object Technology*, vol. 3, no. 9, October 2004, Special issue: .NET Technologies 2004 workshop, pp. 71–94, http://www.jot.fm/issues/issue_2004_10/article4

- 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 [24].

- From Novell, the Mono Open Source project offers a quite complete runtime and good performances. In term of performance, Mono is the best implementation of .NET that runs on both Windows and Linux.

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.

Bigloo

Bigloo is an optimizing compiler for the Scheme programming language (see R^{5}RS [9]). 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 performances are 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 [14].

Bigloo offers several extensions to Scheme [9] such as: modules for separate compilation, object extensions *à la* Clos [5] with extensible classes [17] and optional type annotations for compile-time type verification and optimization.

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.

Outline of this paper

Section 2 presents the main techniques used to compile Bigloo programs to CIL and JVM bytecode. Section 3 enumerates the new functionalities of the .NET framework that could be used to improve the performances of produced code. Section 4 details some practical issues that we faced while using the .NET framework. Section 5 compares the run times of several benchmarks 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 the CLR and the JVM are built upon similar concepts, the techniques deployed for both platforms are quite close. The compilation to JVM being thoroughly presented in a previous paper [14], a more shallow presentation is given here.

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 increases memory usage, we aim at avoiding boxing as much as possible. Thanks to the Storage Use Analysis [18] or user-provided type annotations, numerals or characters are usually passed as values and not boxed, i.e. not allocated on the heap any more. Note that in the C backend, boxing of integers is always avoided using usual tagging techniques [8]. 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 `byte`s, as Scheme strings are mutable sequences of 1 byte characters while the .NET built-in `System.String`s are non-mutable sequences of wide characters. Closures are instances of `bigloo.procedure`, as we will see in Section 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). 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.

Compilation of functions

In Bigloo programs, functions can be separated into 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 function call or used as a return value of function.
- Generic functions are compiled as static methods and use an *ad hoc* framework for resolving late binding.

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
}
```

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 [16]).

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.

In the C backend, closures are implemented as structures that hold their arity, two pointers to functions used depending on whether the function accepts a variable or a fixed number of arguments, and an inlined environment that contains the variables captured by the closure:

```
struct procedure {
  int              arity;
  bigloo_object (*entry)();
  bigloo_object (*va_entry)();
  bigloo_object   env0;
  bigloo_object   env1;
  ...
} procedure_t;
```

Therefore, the size of a closure structure depends on the size of its environment. In other terms, each closure has its own "type". The functions that implement the closure are invoked with a pointer to the closure structure as their first argument

so that they can check their arity and access the captured variables. The total cost of a closure invocation is two function calls and one indirect memory access to read the function pointer. For captured variables, the cost is that of an indirect memory access that goes through the environment (`env0`, `env1`, etc.).

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 produced by the compilation of 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. 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:

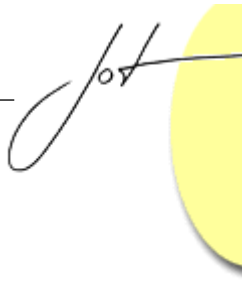
```
(module klist)

(define (make-klist n)
   (lambda (x) (cons x n)))

(map (make-klist 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));
  }
  ...
}
```

Compiling Generic Functions

The Bigloo object model [17] is inspired from Clos [5]: classes only encapsulate data and 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 and single dispatch. Bigloo does not support the Clos Meta Object Protocol.

In both the JVM and the 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, the JVM or the 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.

Continuations

Scheme allows a program to capture the *continuation* of a computation. This continuation can be used to escape from a pending computation, as an exception does, or it can be used to suspend, resume, or even *restart* the computation, as coroutines or threads! Continuations are captured by the Scheme function `call-with-current-continuation`, or `call/cc` in short. This function accepts one argument which is, itself, a function of one argument. When a form `(call/cc f)` is evaluated, the function `f` is invoked and the formal parameter of `f` is bound to the continuation of the `call/cc` form. The program of Figure 1 illustrates how continuations are used. The function `same-fringe` compare the fringe of two trees.

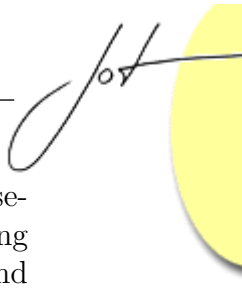
```
(define (same-fringe l1 l2)
   (define (iterate l ret)
      (if (pair? l)
          (iterate (cdr l) (iterate (car l) ret))
          (call/cc (lambda (k) (ret (cons l k))))))
   (define (start l)
      (call/cc (lambda (k) ((iterate l k) 'EOT))))
   (define (next l)
      (call/cc (lambda (k) ((cdr l) k))))
   (let loop ((v1 (start l1)) (v2 (start l2)))
      (cond
         ((and (pair? v1) (pair? v2) (eq? (car v1) (car v2)))
          (loop (next v1) (next v2)))
         ((or (pair? v1) (pair? v2))
          #f)
         (else
          #t))))
```

Figure 1: **Playing with continuations. The function `same-fringe` compare the fringe of two trees. It uses continuations to suspend/resume the tree traversals on each leaf.**

Executing this program produces:

```
same-fringe: 1, 1 = #t
same-fringe: 1, 2 = #f
same-fringe: 1, (1) = #f
same-fringe: (1 . 2), (1 . 2) = #t
same-fringe: (1 . 2), (1 . 3) = #f
same-fringe: (1 2 3), ((1 . 2) 3) = #t
same-fringe: (1 2 3), ((1 . 3) 2) = #f
```

Continuations are difficult to implement especially efficiently. The hardest part is to save and restore the context from which `call/cc` is invoked. For most language



implementations this context is made of a stack and a set of registers. In consequence, for these implementations, saving and restoring the context implies saving and restoring the stack. As already mentioned, Bigloo targets C, JVM bytecode and .NET bytecode. If the specification of the C programming language does not impose a stack, in practice C is always implemented with a stack. The JVM and the .NET virtual machines are specified as using a stack. Since none of these platforms support for explicit stack manipulation, one solution for implementing `call/cc` could be to not use the native stack. We have rejected this approach because we consider its drawbacks too important:

- Managing a private stack is possible but it slows down the performance of the foreign function interface. It slows down the calls to the native language (namely C, Java, and C#) and the calls from the native language to Scheme. This is in contradiction with the Bigloo's philosophy which could be qualified as *zero overhead foreign function calls*.

- Not using the *natural style* [19] of a platform. That is, not using C code that looks like hand-written C code, not using JVM bytecode that looks like bytecode produce by a Java compiler, or not using .NET bytecode that resembles bytecode produced by a C# compiler jeopardizes performance because platforms are always optimized for the code they are expected to run. This is specially true when a JIT is involved. With these compilers, the compilation takes place at runtime. Hence long lasting compilations cannot be afforded. C, JVM and .NET except codes that use the native stack. Hence, it might be expected that C compilers and JITs are far less efficient on codes that manages their own stack. Doing so would thus slows down the speed of Bigloo programs, which seems unacceptable for us.

Since we have decided to use the *natural style* for each platform Bigloo compiles to and since this style uses the native stack, we have to face the problem of saving and restoring it for implementing continuations. This is not possible on all platforms. If a tricky implementation in C using `setjmp`, `longjmp` and `memcpy` [15] allows to save and restore the stack, in the JVM and the CLR, the stack is read-only and thus cannot be restored. As a consequence, amongst its three backends, Bigloo only supports full continuations on its C backend. On the JVM and the CLR platforms a restricted version is proposed: continuations are first-class citizens but they can only be used within the dynamic extent of their capture. In other words, on these platforms, continuations can be used to implement escape operations such as exceptions but they cannot be used to implement operations such as suspending and resuming computations. As such, neither BiglooJVM nor Bigloo.NET conform to Scheme R^{5}RS. In particular, the program of Figure 1 does not run on the JVM and the CLR. It only runs with the C backend. On the JVM and the CLR, Bigloo's `call/cc` is implemented by the means of linked exceptions. Capturing a continuation installs an exception handler. Applying a continuation raises an exception

that is intercepted by the previously installed handler. On these platforms Bigloo's `call/cc` has the expressiveness of Java and C# exceptions.

3 .NET NEW FUNCTIONALITIES

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

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) plus a method call.
- 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 [22]. We have not explored this last possibility because it leads to unverifiable code.

Declaring a new type for each closure

Declaring a new type for each closure, as presented in Section 2, 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 we can expect real programs to embed several thousands closures. For instance, the Bigloo compiler itself, which is a large Scheme program, embeds more than *4000* closures. 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 expensive: 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%.

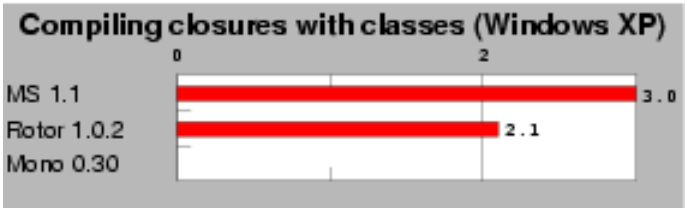


Figure 2: **Declaring one class per closure. This test compares the performance of two techniques for invoking closures: *declaring one type per closure* and *indexed functions*. Scores are relative to *index functions*, which is the 1.0 mark. Lower is better.**

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 2 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 one new type for each closure does not seem to really be a good choice for performances.

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.

```
delegate void SimpleDelegate( int arg );
```

Second, methods whose signature match its declaration are registered. This is illustrated by the following example:

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

Figure 3 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.

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

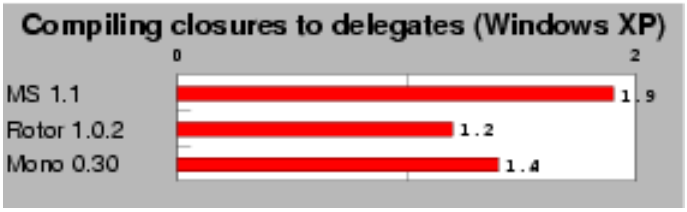


Figure 3: **Compiling closures to delegates. This test compares the performance of two techniques for invoking closures: *delegates* and *indexed functions*. Scores are relative to *index functions*, which is the 1.0 mark. Lower is better.**

languages do not support it. A well knonw technique called trampolining technique [2,23,7,13] allows tail-recursive calls. Since this technique 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.

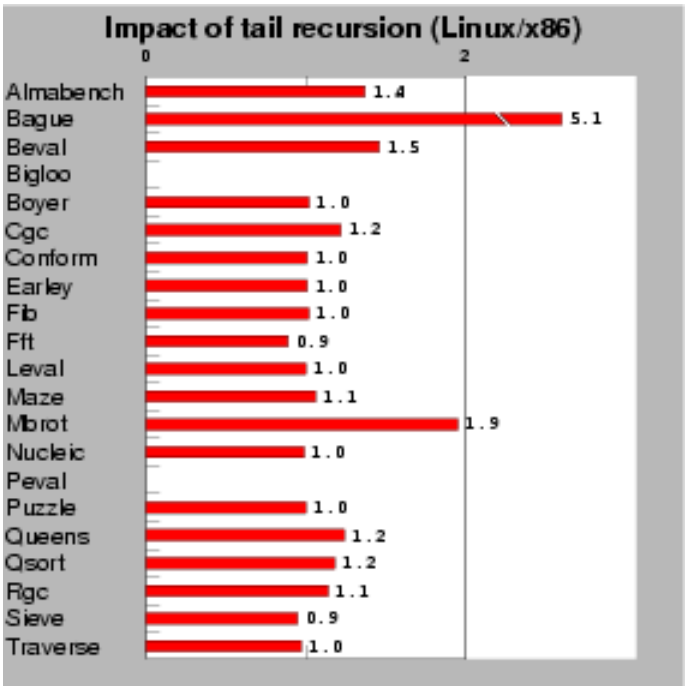


Figure 4: **This test measures the impact of tail recursion on .NET (mono-0.30) executions. Scores are relative to Bigloo.NET, which is the 1.0 mark. Lower is better.**

In the CIL, a function call that precedes 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 4, the impact of tail calls on the overall performance of Bigloo programs (see §7 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.

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.

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! Note that this experiment does measure the startup time of programs.

4 PRACTICAL ISSUES

Working from the JVM backend, it took approximately less than 6 men-monthes to get a mature .NET backend, properly bootstrap the compiler and have the recette working on all 4 implementations of the .NET framework. Apart from the time needed to generate correct `.il` files and to port the Bigloo runtime, a lot of time has been lost in supporting the 4 platforms, mostly because two of them (Portable.NET and Mono) were quite immature at that time and contained a lot of bugs that we had to report and find temporary workarounds for. Although the CIL is portable, the

format of .NET object files is not fully specified. Thus, we have to produce `.il` files that must be assembled and then linked. Unfortunately, the Microsoft assemblers require references to external types to mention the assembly in which the types are defined, although finding this information is the job of a linker. As Mono does not provide an assembler yet, we had to use Portable.NET one. Unfortunately again, assemblies produced by Portable.NET are not accepted by the Microsoft CLR nor Rotor, although Microsoft disassemblers accept them. Therefore, in order to be run by the Microsoft CLR, our assemblies have to be disassembled and then reassembled by Microsoft tools, so that they can at last run on all 4 platforms!

5 PERFORMANCE EVALUATIONS

We have used a large set of benchmarks for estimating the performance of the .NET CLR platform. They are described in Figure §7, 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 the Microsoft CLR (when ran on Windows).

Bigloo vs C#

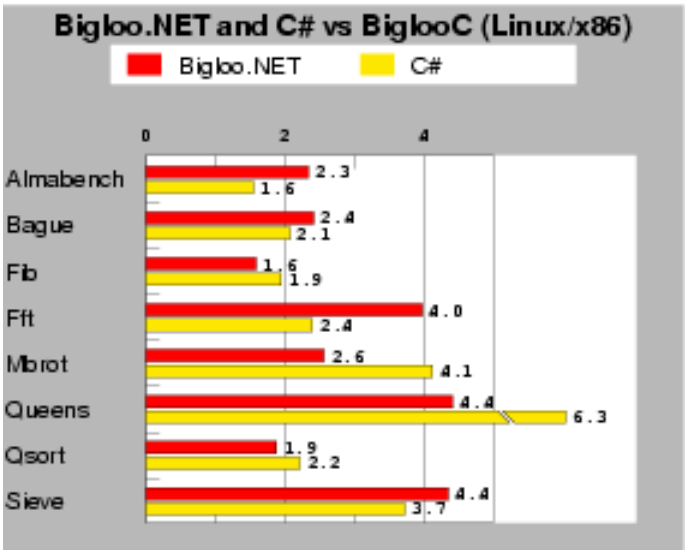


Figure 5: **This test compares the performance of Bigloo.NET vs C# running on mono-0.30. Scores are relative to BiglooC, which is the 1.0 mark. Lower is better.**

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 within reasonable time. For this experiment we use *managed* CIL code. That is, bytecode that complies the byte code verification rules of the CLR. Figure 5 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.

Platform and backend benchmarks

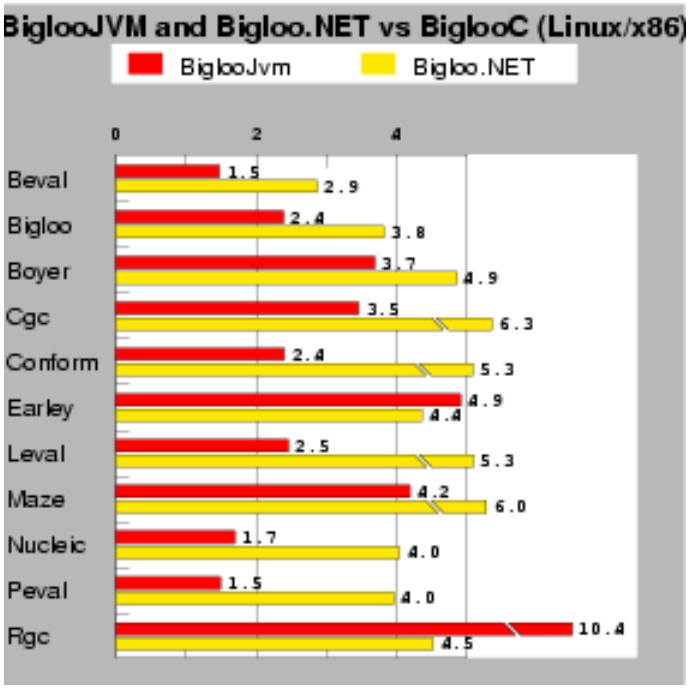


Figure 6: **This test compares BiglooJVM (running on Sun JDK 1.4.2) and Bigloo.NET (running on mono-0.30). Scores are relative to BiglooC, which is the 1.0 mark. Lower is better.**

Figure 6 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 12 presents figures for *unmanaged* and *managed* CIL bytecode.)

In general, Bigloo.NET programs run from 1.5 to 2 times slower than their Bigloo-JVM 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 that of 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 BiglooJVM and Bigloo.NET is stable. This uniformity shows that BiglooJVM avoids traps introduced by JIT*ted* architectures [14]. 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!

Impact of the memory management

	Memory allocation in MegaBytes	
Bench	BiglooC	Bigloo.NET
Beval	**113** (1 ×)	*226* (2 ×)
Bigloo	**80** (1 ×)	*204* (2.55 ×)
Boyer	**346** (1 ×)	*692* (2 ×)
Cgc	**6** (1 ×)	*10* (1.66 ×)
Conform	**400** (1 ×)	*770* (1.92 ×)
Earley	**569** (1 ×)	*1134* (1.99 ×)
Fft	**9** (1 ×)	*12* (1.33 ×)
Leval	**360** (1 ×)	*720* (2 ×)
Maze	**284** (1 ×)	*490* (1.72 ×)
Nucleic	**934** (1 ×)	*1162* (1.24 ×)
Peval	**308** (1 ×)	*618* (2.00 ×)
Queens	**611** (1 ×)	*1221* (1.99 ×)
Qsort	*40* (1 ×)	**39** (0.97 ×)
Rgc	**237** (1 ×)	*553* (2.33 ×)
Sieve	**487** (1 ×)	*1029* (2.11 ×)
Traverse	**29** (1 ×)	*96* (3.31 ×)

Figure 7: **Memory consumption on 32 bits architectures. Amount of memory are expressed in MegaBytes. The CLR implementation is mono-0.30.**

Both BiglooC (native) runtime system and Mono VM use the garbage collector developed by H-J Boehm [6]. Hence, it is relatively easy to compare the memory management of the two backends. This comparison concerns two different aspects: the memory consumption and the speed of allocation/deallocation.

As reported in Sections 2, BiglooC uses traditional C techniques for minimizing the memory space of common objects [8]. C enables a fine grain control over data memory layout. This enables BiglooC to be more compact than Bigloo.NET. Figure 7 presents the memory allocated when the benchmarks are ran with these two backends. The benchmarks that allocate too few objects such Almabench, Bague, or

Mbrot have been eliminated from the table. As demonstrated by this experiment, in average, BiglooC allocates about twice less memory than Bigloo.NET. The main reason comes from *tagging* which is preferred, in BiglooC, to *boxing*. The boxing technique consists in allocating value in the heap and using pointers to these memory locations. The tagging technique consists in using some bits (in general the least significant ones) for encoding the type of the values, the remaining ones being used for representing the values themselves. More precisely, in BiglooC, integers, pairs and constants (e.g., booleans, characters, ...) are tagged using the two least significant bits. Of course, this downgrades arithmetic precision because integers are only 30 bits long on a 32 bits architecture but this also significantly improves performance of program using fixed arithmetic. The pair objects are also represented using an *ad-hoc* memory layout which requires exactly two memory words: one word for the head and another word for the tail. The type of the pair is encoded in the least significant bits of the pointer to the pair.

This compact representation of pairs explains the memory consumption gap between BiglooC and Bigloo.NET on many benchmarks. For instance, on benchmarks such as Beval, Boyer, Leval, Peval, Queens, Rgc, Sieve or Traverse pairs represent more than 90% of the overall allocations! The result of figure 7 are thus unsurprising.

For other benchmarks such as Bigloo and Cgc tagged integers also play an important role in the economic memory consumption of BiglooC. These two programs make extensive use of bit vectors which are encoded using vectors of integers. Hence they consume more memory when integers are boxed (Bigloo.NET) than tagged (BiglooC).

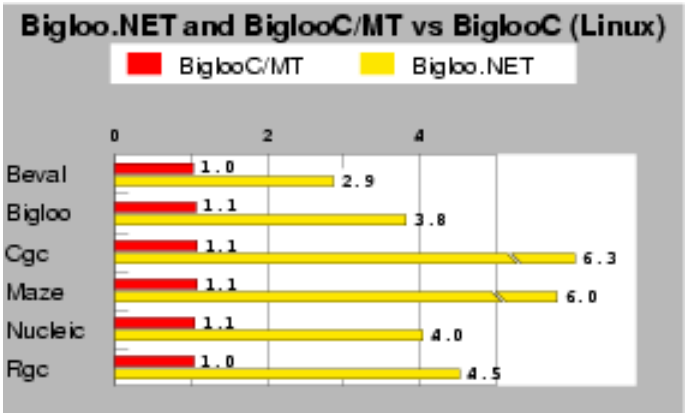


Figure 8: **This test measures the impact of multi-threading on mono-0.30. Smaller is better.**

In addition to using compact memory layout, 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 8 BiglooC/MT is no more than 10% slower than BiglooC on real benchmarks. Therefore, we can conclude that if the multi-threaded memory management slightly decreases the performance, by itself, it is not the culprit for the weaker performance of Bigloo.NET.

.NET framework implementations

When measuring the performance of the four implementations of the .NET framework (Microsoft CLR, Microsoft Rotor, Novell Mono, DotGNU Portable.Net) we have found important disparities. We have found that the Microsoft's CLR is the best implementation of the four platforms. The second one is Novell's Mono. In general the Microsoft's CLR delivers performance that are 20%~30% faster than Mono. On few peculiar benchmarks, the difference of performance is even much more important. Unfortunately, because Microsoft does not allow to publish benchmarking results of .NET, we cannot report more precise information in this paper[1]. Next to Mono, comes the Microsoft's Rotor implementation. Rotor was released for research and educational purposes. As such, Rotor's JIT and GC are simplified and stripped-down versions of the commercial CLR, which leads to poorer performances. At last, the Portable.Net GPL shows the weakest performance because it does not provide a full-fledged JIT [24]. Hence, its speed cannot compete with other implementations so we will not show performance figures for this platform.

Performance is not everything. If the Microsoft's CLR has the performance lead, Mono has the advantage of being a portable implementation. It runs on both Microsoft Windows and Unix. As Java, it supports for runtime portability. This makes Mono an interesting platform to benchmark. Novell's claims constant evolution for the Mono implementation. From a performance point of view, this is not strongly noticeable. The longly awaited version 1.0 does not significantly improve the performance. When it beats the former version 0.30, it reduces the execution times of 10% to 20%. Unfortunately, on other benchmarks, it loses a lot (80% on the Rgc). For this reason, all the time figures reported in this paper have been collected with the former version 0.30 of Mono.

Related Work

Besides Bigloo, several projects have been started to provide support for Scheme in the .NET framework. (*i*) Dot-Scheme [12] is an extension of PLT Scheme that gives PLT Scheme programs access to the Microsoft .NET framework. (*ii*) Scheme.NET[2],

[1] When installing Microsoft.NET framework SDK, one has to sign the following agreement: *Benchmark Testing. You may not disclose the results of any benchmarkk test of the .NET framework component of the Software to any third party without Microsoft's prior written approval.* We have never been able to get such an approval from Microsoft.

[2] http://www.cs.indiana.edu/ jgrinbla

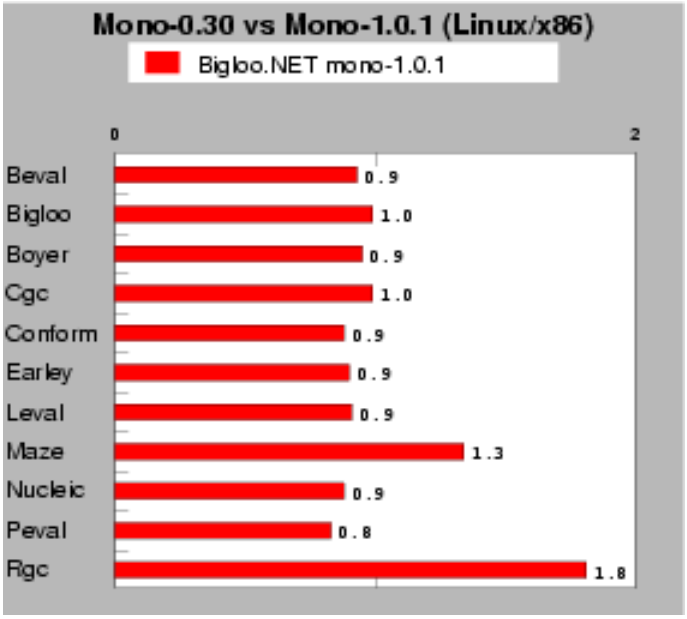


Figure 9: **This test compares the performance of Bigloo.NET performance on mono-0.30 and mono-1.0.1. Scores of mono-1.0.1 are relative to mono-0.30, which is the 1.0 mark. Lower is better.**

from the Indiana University. (*iii*) Hotdog[3], 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#[4] is an implementation of the core of the CAML programming language.(*ii*) From Microsoft Research and the University of Cambridge, SML.NET [4] is a compiler for Standard ML that targets the .NET CLR and which supports language interoperability features for easy access to .NET libraries. Only SML.NET seems to be implemented with high efficiency as a stated goal. Hence, we only present performance comparison with this system in Figure 10. These performance comparison must be read carefuly because it compares equivalent programs written in different languages. Even if Scheme and SML belong to the same programming language familly (strict functional languages supporting side effects) the Scheme writting style is different from the SML writting style. Even if we have tried our best at writting in the natural style of Scheme and SML it might be that some language or implementation idiosynchrasies disturb the experiment.

No paper currently describes the internal of the SML.NET compiler. From private discussion with the authors, it appears that SML.NET being based on the former

[3] http://rover.cs.nwu.edu/ scheme
[4] http://research.microsoft.com/projects/ilx/fsharp.aspx

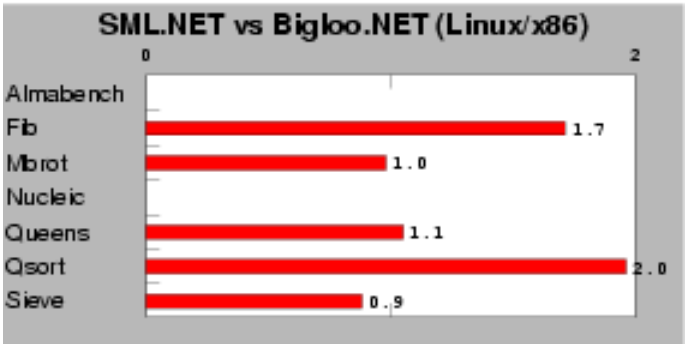


Figure 10: **This test measures the performance of SML.NET Version 1.1 build 671 on mono-0.30. Scores are relative to Bigloo.NET, which is the 1.0 mark. Lower is better.**

MLj compiler [3], it shares many similarities with this older compiler. Contrarly to Bigloo, the SML.NET and MLj compilers do not support separate compilation. If this might improve the performance of resulting programs because it enables global optimization, it also imposes upon users the burden of long compilation times. As presented in Figure 10 Bigloo.NET and SML have close performance. Unsurprisingly, because the benchmarks are single-file programs using very few library functions, we do not see evidence that the *whole-program* compilation strategy of SML.NET significantly improves performance of this particular class of applications. More interestingly, this experiment shows that in the current state of the art of compiling functional languages, the dynamic type checking of Scheme incurs no or little performance penalty over languages that rely on static type checking.

6 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 performance 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. .NET performance is improving form version to version. Bigloo.NET programs still run significantly slower on the Mono implementation than BiglooC and BiglooJVM programs. The same Bigloo.NET programs runs better on the Microsoft's CLR. Sadly, Microsoft does not allow to report on performance evaluation of this platform (see Section 5). In consequence, this paper does not report any precise information regarding the performance of

Bigloo.NET on the Microsoft CLR.

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

[4] Benton, N. and Kennedy, A. Russo, V. – **Adventures in Interoperability: the SML.NET Experience** – 6th Acm sigplan International Conference on Principles and Practice of Declarative Programming (PPDP), Verona, Italy, Aug, 2004, pp. 215–226.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

[14] 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.

[15] Serrano, M. – **Vers une compilation portable et performante des langages fonctionnels** – Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France, Dec, 1994.

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

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

[18] 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.

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

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

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

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

[23] 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.

[24] Weatherley, R. and Gopal, V. – **Design of the Portable.Net Interpreter** – Dot-GNU, Jan, 2003.

ABOUT THE AUTHORS

Yannis Bres is a former research scientist at INRIA Sophia-Antipolis, France, now working as software architect for a public company. See http://www.yannis.bres.name for additional information.

Bernard Paul Serpette is a research scientist at INRIA Sophia-Antipolis, France. See http://www.inria.fr/oasis/Bernard.Serpette for additional information.

Manuel Serrano is a research scientist at INRIA Sophia-Antipolis, France. See http://www.inria.fr/mimosa/Manuel.Serrano for additional information.

7 APPENDIX: THE BENCHMARKS

Benchmark	*Lines*	*Description*
Almabench	300	Tests floating point arihtmetic.
Bague	105	Tests function calls, fixnum arihtmetic, and vectors.
Beval	582	The regular Bigloo Scheme evaluator.
Bigloo	99,376	The bootstrap of the Bigloo compiler and the runtime library.
Boyer	626	Tests symbols and conditional expressions.
Cgc	8,128	A simple compiler for a C like language that produces Mips assembly code.
Conform	596	Uses lists, vectors and numerous small inner functions.
Earley	672	An implementation of the Earley parser.
Fft	120	A Gabriel's benchmark. Fast Fourier transform.
Fib	18	Fibonacci numbers.
Leval	555	A Scheme evaluator using λ-expressions.
Maze	809	Uses arrays, fixnum operations and iterators.
Mbrot	47	The Mandelbrot curve that tests floating point arithmetic.
Nucleic	3,507	Floating point intensive computations.
Peval	639	A partial evaluator that uses nested functions and allocates many lists and symbols.
Puzzle	208	A Gabriel's benchmark.
Queens	131	Ported from Lml to Scheme, tests list allocations.
Qsort	124	Tests arrays and fixnum arithmetic.
Rgc	348	The Bigloo regular grammar that implements the Bigloo reader.
Sieve	53	Fixnum arithmetic and list allocations.
Slatex	2,827	A LaTeX preprocessor that tests Input/Output capacities.
Traverse	136	Allocates and modifies lists.

Figure 11: **Benchmark Descriptions.**

Figure 11 is a short description of the BglStone benchmark suite used in this paper. The numbers of lines are always given for the Bigloo version of the source files. Figure 12 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 Java virtual machine is Sun's JDK1.4.2. The .NET implementation is mono-0.30. The JVM and the 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 single-threaded C programs were the same.

	Wall clock time in seconds				
Bench	BiglooC	BiglooJvm	BiglooJvm (vrf)	Bigloo.NET	Bigloo.NET (mgd)
Almabench	**5.54** (1.0 ×)	10.29 (1.85 ×)	*20.96* (3.78 ×)	8.72 (1.57 ×)	12.99 (2.34 ×)
Bague	**4.71** (1.0 ×)	7.51 (1.59 ×)	7.61 (1.61 ×)	*11.52* (2.44 ×)	11.42 (2.42 ×)
Beval	**5.98** (1.0 ×)	8.88 (1.48 ×)	9.17 (1.53 ×)	17.2 (2.87 ×)	*24.16* (4.04 ×)
Bigloo	**19.2** (1.0 ×)	45.91 (2.39 ×)	46.34 (2.41 ×)	73.48 (3.82 ×)	*84.59* (4.40 ×)
Boyer	**8.43** (1.0 ×)	31.14 (3.69 ×)	30.61 (3.63 ×)	41.03 (4.86 ×)	*57.07* (6.76 ×)
Cgc	**1.97** (1.0 ×)	6.82 (3.46 ×)	6.91 (3.50 ×)	12.4 (6.29 ×)	*19.26* (9.77 ×)
Conform	**7.41** (1.0 ×)	17.82 (2.40 ×)	18.97 (2.56 ×)	39.4 (5.31 ×)	*48.44* (6.53 ×)
Earley	**8.31** (1.0 ×)	40.86 (4.91 ×)	*41.91* (5.04 ×)	36.27 (4.36 ×)	40.61 (4.88 ×)
Fib	**4.54** (1.0 ×)	7.32 (1.61 ×)	*7.34* (1.61 ×)	7.27 (1.60 ×)	7.22 (1.59 ×)
Fft	**4.29** (1.0 ×)	7.8 (1.81 ×)	8.11 (1.89 ×)	15.26 (3.55 ×)	*17.1* (3.98 ×)
Leval	**5.6** (1.0 ×)	13.8 (2.46 ×)	13.81 (2.46 ×)	29.91 (5.34 ×)	*35.11* (6.26 ×)
Maze	**10.36** (1.0 ×)	43.5 (4.19 ×)	43.69 (4.21 ×)	62.18 (6.00 ×)	*64.2* (6.19 ×)
Mbrot	**19.82** (1.0 ×)	49.82 (2.51 ×)	49.62 (2.50 ×)	*51.47* (2.59 ×)	51.03 (2.57 ×)
Nucleic	**8.28** (1.0 ×)	14.1 (1.70 ×)	14.29 (1.72 ×)	33.53 (4.04 ×)	*37.85* (4.57 ×)
Peval	**7.57** (1.0 ×)	11.28 (1.49 ×)	11.87 (1.56 ×)	30.01 (3.96 ×)	*32.47* (4.28 ×)
Puzzle	**7.59** (1.0 ×)	12.96 (1.70 ×)	13.03 (1.71 ×)	20.96 (2.76 ×)	*29.26* (3.85 ×)
Queens	**10.47** (1.0 ×)	36.97 (3.53 ×)	37.75 (3.60 ×)	42.76 (4.08 ×)	*46.37* (4.42 ×)
Qsort	**8.85** (1.0 ×)	13.26 (1.49 ×)	13.39 (1.51 ×)	*16.93* (1.91 ×)	16.72 (1.88 ×)
Rgc	**6.94** (1.0 ×)	72.3 (10.41 ×)	*73.11* (10.53 ×)	31.43 (4.52 ×)	33.48 (4.82 ×)
Sieve	**7.37** (1.0 ×)	25.44 (3.45 ×)	25.17 (3.41 ×)	31.01 (4.20 ×)	*32.19* (4.36 ×)
Traverse	**15.19** (1.0 ×)	43.02 (2.83 ×)	43.24 (2.84 ×)	76.4 (5.02 ×)	*81.59* (5.37 ×)

Figure 12: **Benchmarks timing of Bigloo2.6e on an AMD Tbird 1400Mhz/512MB, running Linux 2.4.21, Sun JDK 1.4.2, and mono-0.30.**