Running Lua Scripts on the CLR through Bytecode Translation

Fabio Mascarenhas[1*], **Roberto Ierusalimschy**[1]

[1]Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente, 225 – 22453-900
Rio de Janeiro, RJ, Brasil

mascarenhas@acm.org, roberto@inf.puc-rio.br

Abstract. *The .NET Common Language Runtime (CLR) aims to provide interoperability among code written in several different languages, but porting scripting languages to it, so that scripts can run natively, has been hard. This paper presents our approach for running scripts written in Lua, a scripting language, on the .NET CLR.*
Previous approaches for running scripting languages on the CLR have focused on extending the CLR, statically generating CLR classes from user-defined types in the source languages. They required either language extensions or restrictions on the languages' dynamic features.
Our approach, on the other hand, focused on keeping the syntax and semantics of the original language intact, while giving the ability to manipulate CLR objects. We implemented a translator of Lua virtual machine bytecodes to CLR bytecodes. Benchmarks show that the code our translator generates performs better than the code generated by compilers that use the previous approaches.

Resumo. *O objetivo do Common Language Runtime .NET (CLR) é permitir a interoperabilidade entre código escrito em diversas linguagens, mas a execução nativa de linguagens de script no CLR tem sido difícil. Este artigo apresenta nossa abordagem para executar scripts escritos em Lua, uma linguagem de script, no .NET CLR.*
Abordagens anteriores para se executar linguagens de script no CLR enfatizaram a extensão do CLR, gerando estaticamente novas classes CLR a partir de tipos definidos naquelas linguagens. Aquelas abordagens precisavam fazer extensões àquelas linguagens, ou restringir suas características dinâmicas.
Nossa abordagem, entretanto, enfatiza a conservação da sintaxe e semântica da linguagem original, enquanto dá a ela a habilidade de manipular objetos do CLR. Nós implementamos um tradutor de bytecodes da máquina virtual de Lua para bytecodes do CLR. Testes mostram que o código que nosso tradutor gera tem desempenho melhor do que o do código gerado por compiladores que usam as abordagens anteriores.

1. Introduction

The aim of the Microsoft .NET Framework is to provide interoperability among several different languages, through a Common Language Runtime [Meijer and Gough, 2002].

*Supported by CAPES.

The .NET CLR specification is an ISO and ECMA standard [Microsoft, 2002]. Microsoft has a commercial implementation of the CLR for its Windows platform, and non-commercial implementations already exist for other platforms [Stutz, 2002, Ximian, 2005]. Some languages already have compilers for the CLR, and compilers for other languages are in several stages of development [Bock, 2005].

Lua [Ierusalimschy, 2003, Ierusalimschy et al., 1996] is a scripting language that is easy to embed, small, fast, and flexible. It is interpreted and dynamically typed, has a simple syntax, and has several reflexive facilities. Lua also has first-class functions, lexical scoping, and coroutines. It is widely used in the development of computer games.

Scripting languages are often used for connecting components written in other languages ("glue" code). They are also used for building prototypes, and as languages for configuration files. The dynamic nature of these languages lets them use components without previous type declarations and without the need for a compilation phase. Although they lack static type checks, they perform extensive type checking at runtime and provide detailed information in case of errors. Ousterhout argues that the combination of these features can increase developer productivity by a factor of two or more [Ousterhout, 1998].

This paper presents an approach for running Lua scripts natively on the CLR, by translating bytecodes of the Lua Virtual Machine to bytecodes of the Common Intermediate Language. The Common Intermediate Language, or CIL, is the underlying language of the CLR. Our approach leaves the syntax and semantics of the Lua scripts intact, while achieving adequate performance. The bytecode translator is called Lua2IL.

Porting scripting languages to the CLR has been hard. ActiveState has tried to build Perl and Python compilers, but abandoned both projects [ActiveState, 2000, Hammond, 2000]. Smallscript Inc. has been working on a Smalltalk compiler for the CLR since 1999 [Smallscript Inc., 2000], but there was no version of it available on February 2005.

A common trend among those projects is their emphasis on extending the CLR. They map user-defined types in the source languages to new types in the CLR, and then generate these types during compilation. That emphasis makes porting harder, as dynamic creation and modification of types is a common feature of scripting languages, Lua included. Those projects ended up extending the syntax and semantics of the languages, or removing some of their dynamic features.

Our approach, on the other hand, emphasizes the full implementation of the features of the original language, without impairing its ability as a consumer. Lua scripts that go through our translator are very restricted in their ability to extend the CLR with new types, even though they run natively on the CLR. The scripts have full access to existing CLR types, but this access is mediated by an interface layer. This interface layer has the capabilities of a full CLS consumer.

The Common Language Specification (CLS) is a subset of the CLR specification that establishes a set of rules for language interoperability [Microsoft, 2002, CLI Partition I Section 7.2.2]. Compilers that generate code capable of using CLS-compliant libraries are called *CLS consumers*. Compilers that can produce new libraries or extend existing ones are called *CLS extenders*. A full CLS consumer must be able to call any CLS-compliant method or delegate, even methods with names that are keywords of the language; to call distinct methods of a type with the same signature but from different interfaces; to instantiate any CLS-compliant type, including nested types; to read and write

any CLS-compliant property; and access any CLS-compliant event. All of these features are supported by the interface layer of Lua2IL, and are available to Lua scripts.

The rest of this paper is structured as follows: Section 2 describes the bytecode translator and the interface layer. Section 3 presents some related work and performance evaluations, and Section 4 presents some conclusions and future developments.

2. Translating Lua scripts to the CLR

Translating a Lua script to the Common Language Runtime involves several issues, the actual translation of the bytecodes being just one of them. First there should be a way to represent Lua types using the types in the CLR; we cover this on Section 2.1. Then there is the implementation of the VM instructions (the translation itself), covered in Section 2.2. Then there are the features of the Lua language that the Lua runtime environment implements: coroutines and weak tables. We cover coroutines in Section 2.3 and weak tables in Section 2.4. Finally, Lua scripts need to manipulate other CLR objects (instantiate them, access their fields, call their methods, and so on). Section 2.5 details the implementation of Lua wrappers for other CLR objects.

2.1. Representing Lua types in the CLR

A naive approach to represent Lua types in the CLR would be to map Lua numbers[1], strings, booleans and *nil* directly to their respective CLR types (*double*, *string*, *bool*, and *null*). Two new CLR types would represent tables (associative arrays) and functions. The advantage is that CLR code written in other languages would work with Lua types directly, and vice-versa.

There is a severe disadvantage, though: Lua is dynamically typed, so the code that Lua2IL produces would have to use the lowest common denominator among CLR types, the *object* type. Most operations would require a type check (with the `isinst` instruction of the CLR) and a type cast. The code would have to box and unbox all numbers and booleans operate on them, wasting memory and worsening performance.

Lua2IL does not use this naive representation. Internally, the code that it generates deals with instances of the `LuaValue` structure. This structure has two fields, `O`, of type `LuaReference`, and `N`, of type `double`. `LuaValue` instances with the `O` field set to `null` represent Lua numbers. Subclasses of `LuaReference` represent all the other Lua types.

Instances of `LuaString`, which use a CLR `string` internally, represent Lua strings. Instances of `LuaTable` represent tables, implementing a C# version of the algorithm that the Lua interpreter uses. This algorithm breaks a table in an array part and a hash part, to optimize the use of tables as arrays.

Each Lua function gets its own class, subclassed from a common ancestor, the `LuaClosure` class. The classes have a `Call` virtual method that executes the function (the translated bytecodes). Lua functions are first class values, so the actual functions are instances of their respective classes. The main body of the script is also a function, represented by a class named `MainFunction`. Instantiating this class and then calling its `Call` method runs the script.

Booleans and nil are a special case. There is a singleton object for each boolean value (the singleton instances of `TrueClass` and `FalseClass`). The same happens with nil (the singleton instance of `NilClass`).

[1] Lua numbers are floating point numbers with double precision.

Userdata are a Lua type that represents data from a host application or a library. The `LuaWrapper` class represents userdata, and instances of this class are proxies to CLR objects. We cover this class in more detail in Section 2.5.

The representation we use does not need type casts, as there are common denominators among all types (`LuaValue` and `LuaReference`). To check if a value is a number, for example, the code just checks whether its `O` field is `null`. If the `O` field is null, the number is stored in the `N` field. As another example, the code to index a value just checks if its `O` field is not `null`, then calls the `get_Item` method of the `O` field. If the value does not support this operation, the implementation of this operation in the value's class throws an exception (the Lua interpreter would flag an error in this case).

2.2. Translation of the bytecodes

When Lua2IL translates a Lua script (previously compiled to Lua Virtual Machine bytecodes), it first reads the script and builds an in-memory tree structure of it. Each function the script defines is a node of this tree, and the body of the script is the root. Lua2IL walks this tree, in preorder, compiling each node to a subclass of `LuaClosure`. The end result is a library containing all those subclasses.

Calls to Lua functions do not use the standard CLR parameter passing mechanism. When Lua2IL compiles a call to a Lua function, it has no way of knowing how many parameters there are in the function being called, nor how many values it will return (Lua functions can return multiple values). One possible way to pass parameters to Lua functions would be using an array, with return values collected using another array. The downside is that two arrays must be instantiated and filled in every function call, so we used an alternative way.

This alternative way is to have a Lua stack, an auxiliary stack parallel to the CLR execution stack. Lua2IL uses the Lua stack for parameter passing and collecting return values. Each function receives this stack when it is called, along with how many arguments it is receiving, and returns how many return values it pushed on the stack. For example, a function `foo` calling another function `bar` with *10* and *3* as arguments would push both arguments to the stack, then call `bar` passing the stack and the number *2* (for two arguments). If `bar` wants to return the values *3* and *1*, it would push them into the stack and return the number *2* (for two return values).

Lua functions also use the Lua stack to store their local variables, instead of using CLR locals. This is required by our implementation of lexical closures. The code does not use strict stack discipline when operating on locals, however. For example, an addition operation of two locals gets their values directly from their stack positions, storing it in another stack position. There is no need to push the values to the top of the stack before operating on them.

The `LuaClosure` class also defines a helper method that receives an array of arguments, builds a new Lua stack, calls the function with this stack, pops the return values, and then returns them in another array. Other CLR programs can use this helper method as a more natural interface to Lua functions.

The Lua stack is implemented as an array of `LuaValue` instances. The stack starts small and automatically grows as needed, doubling in size each time it is filled. The stack does never shrink, although object references are cleared as the stack unwinds.

Using an auxiliary stack mimics the way that the Lua interpreter implements the Lua Virtual Machine. The Lua VM is register-based, but its registers are actually virtual,

mapped to positions in the Lua execution stack [Ierusalimschy, 2002]. Parameter passing and return in the Lua interpreter works just as described earlier in this section. The Lua stack also lets Lua2IL reuse the interpreter's implementation of lexical closures.

Due to the similarity between the execution models of the Lua interpreter and of Lua2IL, we could do, for most of the Lua VM instructions, a straightforward translation from the original ANSI C implementation of the Lua interpreter to the Common Intermediate Language of the CLR. The translation of some instructions is not as straightforward, though. The Lua interpreter implements function calls, tail calls and function returns by creating and maintaining its own activation records for each function call. Lua2IL uses the CLR stack to do this, letting the CLR keep track of activation records for each Lua function call, as each Lua function call is also a CLR method call.

The implementation of the function call instruction invokes the `Call` method of the callee, passing the stack and number of arguments (pushed into the stack by previous instructions). A preamble in the `Call` method adjusts the arguments in the stack to the number of arguments that the function expects, then clears the stack space that the function will use (possibly growing the stack). The implementation of tail calls is slightly different: Lua2IL first copies the arguments to the beginning of the stack of the caller, then invokes the `Call` method using the tail call instruction of the CIL. The implementation of function return copies the return values to the end of the caller's stack space, then unwinds the Lua stack and does a CIL method return.

The first prototype of Lua2IL translated each instruction as a call to a helper method, like a threaded interpreter. The helper methods were implemented in C#. This approach was slower, but easier to debug. After implementations of all the VM instructions were done and debugged, we changed Lua2IL to directly emit CIL code instead of just calling helper methods, effectively inlining those methods. This inlining allowed a few more optimizations. Many of the Lua VM instructions can operate on either literal values or registers. In the threaded translator, the helper method that implemented the instruction did the tests to see whether the operands were literals or registers. The inlined translator does these tests at translation time, and the CIL code that it generates is specialized to operate either on a literal or a register.

All instructions are inlined, but a few of them are partially implemented by C# helper methods. In these cases, the inlined portion deals with the common case, and delegates other cases to a helper. For example, the inlined implementation of arithmetic instructions does the arithmetic operation itself when both operands are numbers, delegating to a helper method when dealing with operands of other types.

2.3. Coroutines

Lua supports full asymmetric coroutines [Moura et al., 2004]. A Lua coroutine is a first-class value. During its execution, the coroutine can yield control back to its caller at any time, including deep inside nested function calls. When a coroutine yields, its execution is suspended. It can be later resumed from any point in the script, even inside other coroutines. Returning from the main function of a coroutine also yields control back to the caller, but the coroutine is marked as dead and can no longer be resumed. If an error occurs during the execution of a coroutine, this error is captured and returned to the caller, and then the coroutine is marked as dead.

Lua2IL implements coroutines on top of CLR threads, using semaphores for synchronization. Each coroutine has its own Lua stack, plus a CLR thread and two binary semaphores. The semaphores are called *resume* and *yield*, and are initially closed. When

the script creates a coroutine, the thread of the coroutine is started. The first action of this thread is to try to decrement its *resume* semaphore, making CLR suspend it.

When another thread resumes a coroutine, it increments the *resume* semaphore of the coroutine, restarting the execution of the coroutine's thread. Then the caller thread decrements the *yield* semaphore of the coroutine, suspending itself.

When a coroutine yields back to its caller, it increments its *yield* semaphore, restarting the execution of the caller thread. Then the coroutine decrements its *resume* semaphore, suspending itself. When the coroutine returns (finishes executing), it increments its *yield* semaphore, again restarting the caller thread, then the coroutine is flagged as terminated. Any exception occurring during execution of a coroutine is trapped and terminates the coroutine.

The downside of this implementation is the overhead caused by context switches and synchronization, as each CLR thread is an OS thread, and swapping among them involves a full context switch. This overhead is not present in the coroutine implementation of the Lua interpreter. However, this is the only way of implementing coroutines on the CLR using managed code (that is, in a portable way). A native code implementation exists that uses Windows fibers (cooperative threads), but it is not portable, it has problems interacting with the CLR garbage collector and exception handling subsystems, and it uses undocumented API calls [Shankar, 2003].

2.4. Weak Tables

The Lua VM supports weak references through *weak tables*. A weak table may have weak keys, weak values, or both. If a weak key or value is collected then its pair is removed from the table. The garbage collector of the Lua interpreter puts weak tables in a list during the mark phase; in the end of this phase the collector traverses the tables and removes all pairs with unmarked weak references.

The Lua2IL runtime implements weak tables by storing a CLR weak reference to the key (or value) instead of the key itself. A CLR weak reference is an instance of `System.WeakReference`; the runtime wraps weak keys and values with instances of this type.

This implementation introduces overhead in every table access, unlike the implementation the interpreter uses. Besides this added overhead, the current implementation does not remove a weak reference from the table after the object it references is collected. The only event the CLR associates with garbage collection is object finalization, through a `Finalize` method, which adds overhead to garbage collection (objects with this method are collected differently). Implementing a notification system on top of `Finalize` is possible: each object can keep a list of the tables that have weak references to them, and the `Finalize` method of each object can go through this list, removing the pairs that contain the object. Besides the lack of elegance of this solution, it also implies a performance hit over the whole Lua2IL system, as every object would have a `Finalize` method, even if the object is never put inside a weak table.

A better way would be if the CLR notified the weak reference when it became invalid, or if it let applications register methods that would be executed after each garbage collection cycle. Another possible mechanism would be the one present in the Java Virtual Machine: associate a queue with each weak reference, and when the reference becomes invalid it is added to this queue.

2.5. Working with CLR objects

Our approach manages to keep the syntax and semantics of the Lua language intact. This comes with a price, though, as the scripts are isolated from the rest of the CLR; they have no direct notion of external CLR types. But we can give them access to these types through a layer that sits between the Lua environment and the rest of the CLR, automatically translating from Lua types to CLR types and vice-versa, all at runtime.

This integration layer is a full CLS consumer. It lets lets Lua scripts manipulate CLR objects. The scripts can get references to CLR types and use these references to instantiate objects, then access fields of those objects and call their methods. The scripts do al these operations with the standard Lua syntax. They can even pass Lua functions to methods that expect delegates, to handle events with Lua code, for example.

Lua2IL represents types and objects from the CLR with the `LuaWrapper` class, which has two other subclasses; one of them represents types, and is responsible for object instantiation and access to static members, while the other represents instances, and is responsible for access to instance members. The `LuaWrapper` class and its subclasses have methods that implement indexing (both to read and write values) and function invocation. For example, an expression like `obj:foo(arg1, arg2)` is translated by the Lua parser to the equivalent expression `obj["foo"](obj, arg1, arg2)`. The `obj["foo"]` subexpression emits a bytecode for an indexing operation, and Lua2IL translates this bytecode to a call to an indexing method on `obj`. If `obj` is a Lua table, this method returns the value stored in the table under the `"foo"` key. If `obj` is an instance of `LuaWrapper`, its indexing method searches for a method `foo` in the CLR object represented by `obj`, using the CLR reflection API. If the search finds the method then the indexing method returns a proxy to it, otherwise it returns `nil`.

Continuing the previous example, the compilation of the call to the value returned by `obj["foo"]` emits a call (or tail call) bytecode, which Lua2IL translates as a call to the method `Call` of the proxy. The proxy's `Call` method pops the arguments from the Lua stack, converts them to the types that the CLR method requires, and then calls it. If the method is overloaded, the proxy tries to call each of the methods, in the order they are defined, and throws an exception if all the calls fail because of incompatible arguments.

The cost of searching for a method with the reflection API is high, so the instances of `LuaWrapper` cache proxies. This cache is shared by all instances of a same type. Proxies to overloaded methods cache the last successful method that was called; on the next call the proxy tries this method first.

Going back to the `obj["foo"]` example, if `foo` is a field, the indexing method of `obj` finds `foo`, using reflection, and returns the value of `foo` in the CLR object represented by `obj`. The proxy caches the field (not its value), so the next access does not need a new reflexive search. Properties are treated in a similar manner. Writing to a field, like in `obj.foo = bar`, emits an indexing bytecode that sets the value at the index. This is translated by Lua2IL to a call to an indexing method that sets the value. This method finds the `foo` field, using reflection, converts `bar` to its type, and assigns to the field. The proxy caches the `foo` field, as mentioned in the previous paragraph. Writing to properties is again treated in a similar manner.

The Lua2IL runtime automatically converts Lua functions to delegates, if a method expects a delegate as a parameter. A script can use this to register Lua functions as event handlers, for example. The runtime dynamically generates a new class that implements a method with the delegate's signature. This method dispatches to a Lua function. The

runtime instantiates this class with the function being converted, and creates a delegate from this instance. The dynamic classes are generated with the Reflection.Emit API and kept in a temporary, memory-only library.

3. Related Work

During the years 1999 and 2000, Microsoft sponsored the development of a Python compiler for the CLR, called *Python for .NET* [Hammond, 2000]. Python for .NET traverses the abstract syntax tree generated by the CPython interpreter, emitting Common Intermediate Language code through the `Reflection.Emit` API. The implementation has some similarities to the implementation of Lua2IL: Python for .NET defines a `PyObject` structure for its values, and a `IPyType` interface that define what operations can be done on those values (the Lua2IL equivalents are the `LuaValue` structure and `LuaReference` class, respectively).

Around 95% of the Python core is implemented, according the author. Missing features are primitive types without a direct mapping to the CLR (arbitrary size integers, complex numbers and ellipses), and built-in methods of Python classes, used for dynamic extension of classes and objects. The language syntax was not modified. The development of this compiler halted about two years ago. The last available prototype is dated April 2002, with parts of it dated April 2000.

Perl for .NET is a Perl compiler for the CLR, and was developed by ActiveState between 1999 and 2000 [ActiveState, 2000]. The compiler works as a back-end to the Perl interpreter, generating C# code (not CIL) that calls a Perl runtime for its operations. There is no information about how much of the Perl language is covered by the compiler, and the source code for it is not available. The last available prototype is dated June 2000, and does not work with released versions of the CLR, only with betas.

JScript.NET is an extension of the JScript language (or EcmaScript) with a compiler for the CLR, and is part of the Microsoft .NET Software Development Kit. The language extensions include classes and optional type declarations. The dynamic features of JScript are still available, although interoperation with other CLR code is compromised if the extensions are not used: delegates must be declared with the correct signature (including type declarations), and declared inside a class. The code generated by the compiler uses CLR types natively, requiring type checking and casts in every operation with dynamic typing.

S# is a dialect of Smalltalk developed by SmallScript Corporation, and S#.NET is a S# compiler for the CLR. According to its author, the compiler and the language runtime are ready, but still need to be integrated with the Visual Studio.NET development environment before being released to the public. The compiler has been under development since 1999.

IronPython is a new Python compiler for the CLR, and is being developed by Jim Hugunin [Hugunin, 2004]. It uses its own parser, written in C#, and supports all of the Python core. IronPython does some aggressive optimizations on its generated code, specially if some of the more dynamic features of Python are not used. Like the JScript.NET compiler, it uses native CLR types whenever possible, but does not use type annotations, and any Python function can be a delegate.

3.1. Performance Evaluation

Our first performance test is the execution of six scripts from *The Great Win32 Computer Language Shootout* [Bagley, 2005], mainly involving arithmetic operations, recursion,

Script	Description
ack	Ackermann function, arguments 3 and 8
fibo	Fibonacci numbers, the 30^{th} number
random	Random number generation, generate 1,000,000 numbers between 0 and 100
sieve	Sieve of Eratosthenes, from 2 to 8,192, 100 runs
matrix	30×30 matrix multiplication, 100 runs
heapsort	Heap sort on an array of 100,000 random numbers

Table 1: Scripts for the first compiler performance test

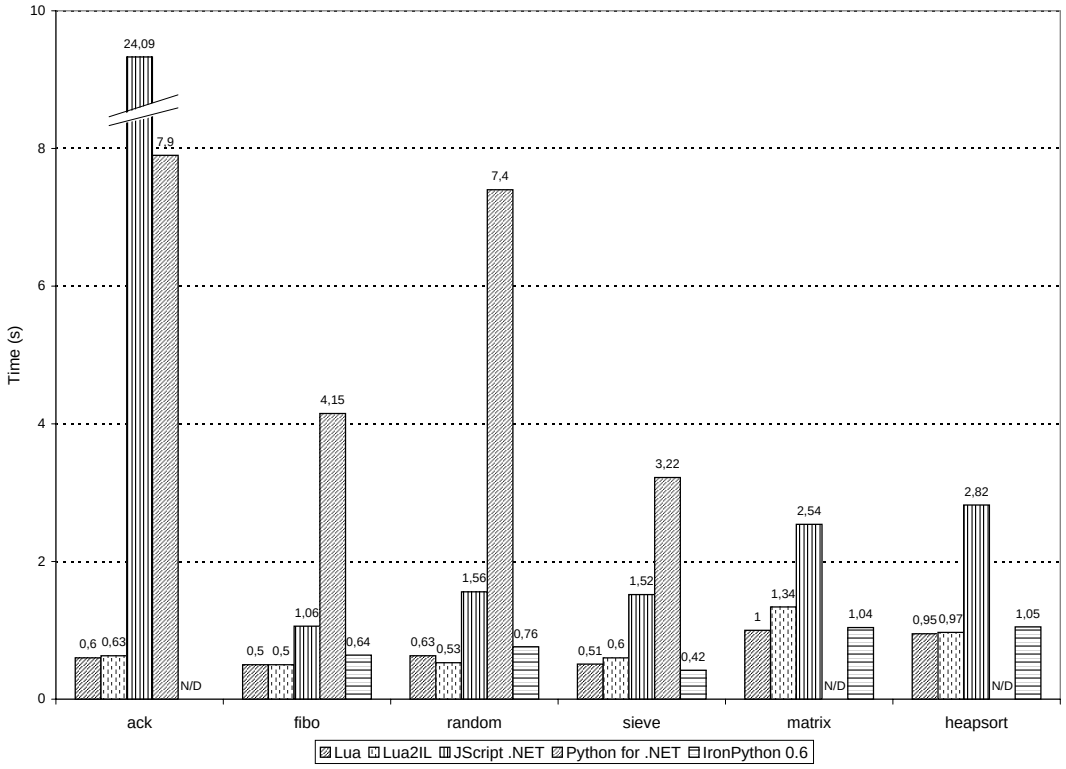

Figure 1: Results for the first performance test

and array accesses. The goal is to evaluate the performance of the code generated by the compilers when running the primitive operations of the languages. A description of each test script and the arguments of its execution is on Table 1.

We tested the Lua2IL, JScript.NET, Python for .NET and IronPython 0.6 compilers. The same scripts compiled by Lua2IL were also executed by the Lua 5.0.2 interpreter. We did not test the Perl for .NET compiler, as it did not work with the version of the CLR we used.

The results are shown on Figure 1. The times are in seconds, and all the scripts were run on the same machine, under the same conditions[2].

Python for .NET did not compile the *matrix* and *heapsort* scripts, even though these scripts were syntactically correct. IronPython successfully compiled the *ack* script,

[2]Pentium 2.8GHz HT, with 512Mb memory, running Windows XP Professional with version 1.1 of the .NET Common Language Runtime. The Lua interpreter was compiled by the Microsoft 32-bit C/C++ optimizing compiler, version 13.10.3077 for 80x86, with the /O2 switch.

but it ran out of stack space during execution and crashed.

The Python for .NET compiler lags behind the others, as its authors halted the development of the compiler before writing an optimizer. Next comes the JScript.NET compiler, penalized by the inefficient code it generates for numerical operations. Binary operations, except addition, are computed by a generic evaluator object that receives a numeric code for the operation and both operands. These evaluator objects are created in the heap, at each execution of the function, so heavily recursive numerical code is memory-intensive and very demanding on the garbage-collector.

Both Lua2IL and IronPython show close results, with an advantage for Lua2IL in numerical code, probably due to the type checks present in the code IronPython generates. IronPython is at a slight advantage in code that uses arrays. Arrays are an optimization of tables in Lua2IL, and the Lua2IL runtime must check, at each array access, if the index is an integer and if it is in the bounds of the array part of the table, defaulting to use the hash part if each of these tests fail. The Lua interpreter is the fastest overall.

The second performance test is a measuring of the time it takes to complete a method call to a CLR object. The test was done with code generated by the Lua2IL, JScript.NET (using late binding, with no type declarations), and IronPython compilers. The Python for .NET compiler could not instantiate the types in the assembly used in this test. We evaluated times for calls to six distinct methods. They vary by the number and types of their parameters. Three of the methods have all parameter and return values of type `Int32`, and are called with zero, one, and two parameters. The other three methods have parameters and return values of type `object`.

The results of the test are show on Figure 2, and are in microseconds. They were collected on the same machine and under the same conditions of the first performance test. The *Lua* columns show the times for calls from the Lua interpreter, using the LuaInterface [Mascarenhas and Ierusalimschy, 2004] library. The other columns show the times for calls from code generated by the respective compilers.

For this test, the code generated by JScript.NET and Lua2IL are very close, within 10% of each other. This shows that any overhead introduced by the peculiarities of the code generated by each compiler is dwarfed by the time for the actual reflexive invocation of the method. IronPython, on the other hand, clearly does not optimize method calls as well as it optimizes the execution of Python code.

The higher times for the calls from the Lua interpreter are a result of the overhead involved in passing values from the environment of the Lua interpreter to the managed environment of the CLR. This shows the performance advantage of code running directly under the CLR, which needs much less scaffolding.

4. Conclusions

This paper presented an approach for running scripts from Lua, a dynamically typed language, on the Common Language Runtime. The approach woks by translating the bytecodes of the Lua virtual machine to bytecodes of the CLR. The goal was to keep the syntax and semantics of the language unchanged; any script that the Lua interpreter executes, as long as it does not use library code, should be translatable to CIL code that with the same behavior. There is also an integration layer that lets scripts freely manipulate CLR types.

Previous attempts at creating CLR compilers for scripting languages have focused on static generation of classes, either by extending the language, in the case of JScript,

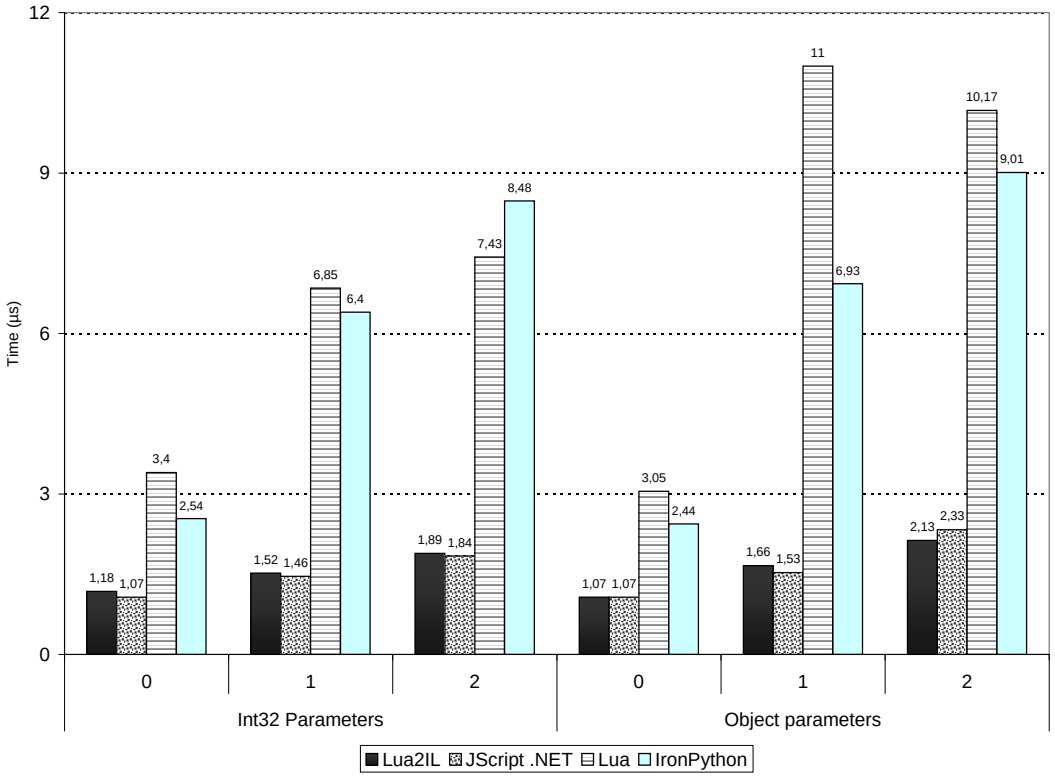

Figure 2: Times for method calls

or by restricting dynamic features, in the case of the Python for .NET compiler. Our approach focuses on reproducing the semantics of the language and offering access to CLR objects. We think the role of a consumer, instead of a creator, of CLR types is more suited to scripting languages. A recent Python compiler for the CLR, IronPython, uses a similar approach to ours, and matches some of our results.

The goal of keeping the semantics of the language was almost fulfilled, with only weak tables having a different semantic, due to the absence of any mechanism in the CLR that notifies when a weak reference becomes invalid.

Lua2IL does some optimizations in the generated code, like generating specialized implementations of Lua bytecodes. The integration layer also optimizes calls to methods of CLR objects, caching the methods that are discovered through reflection.

The performance of the code generated by Lua2IL was compared with code generated by three other compilers for dynamically typed languages: a commercial compiler of the JScript language (developed by Microsoft), and two open source prototype implementations of Python compilers. We also compared the performance with that of the same code executed by the latest release of the Lua interpreter. The results are mixed, with the code generated by Lua2IL performing better than the others in tests that are not dominated by array accesses. Lua2IL, like the Lua interpreter, implements arrays as an optimization of tables, not as a dedicated array type. Lua2IL performs well even in tests dominated by array accesses, though, coming close to the fastest compiler.

Performance evaluation of the time taken by calls to the methods of other CLR objects shows that the code generated by Lua2IL performs similarly to code generated by JScript.NET, and better than the code generated by IronPython and a Lua to CLR bridge. The overall time is dominated by reflexive invocation, on code generated by both Lua2IL

and JScript.NET.

For the future, we are doing a more efficient coroutine implementation that does not depend on threads. We also plan on making the CLR garbage collector more flexible, so it can better adapt to languages with finalization semantics different from the one used by C#. Another plan is to investigate how to enable faster execution of scripting languages by the CLR, to bring the performance nearer the performance of statically-typed languages.

References

ActiveState (2000). Release Information for the ActiveState Perl for .NET compiler. Available at `http://www.activestate.com/Corporate/Initiatives/NET/Perl_release.html`.

Bagley, D. (2005). The Great Computer Language Shootout. Available at `http://dada.perl.it/shootout/`.

Bock, J. (2005). .NET Languages. Available at `http://www.dotnetlanguages.net/DNL/Resources.aspx`.

Hammond, M. (2000). Python for .NET: Lessons Learned. Available at `http://www.activestate.com/Corporate/Initiatives/NET/Python_for_.NET_whitepaper.pdf`.

Hugunin, J. (2004). IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004 International Python Conference*.

Ierusalimschy, R. (2002). The Virtual Machine of Lua 5.0. In *Lightweight Languages 2003 Workshop*. Available at `http://www.inf.puc-rio.br/~roberto/talks/lua-ll3.pdf`.

Ierusalimschy, R. (2003). *Programming in Lua*. Lua.org.

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (1996). Lua — An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652.

Mascarenhas, F. and Ierusalimschy, R. (2004). LuaInterface: Scripting the .NET CLR with Lua. *Journal of Universal Computer Science*, 10(7):892–908.

Meijer, E. and Gough, J. (2002). Technical Overview of the Common Language Runtime. Technical report, Microsoft Research. Available at `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`.

Microsoft (2002). ECMA C# and Common Language Infrastructure Standards. Available at `http://msdn.microsoft.com/net/ecma/`.

Moura, A. L. d., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925.

Ousterhout, J. (1998). Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30.

Shankar, A. (2003). Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API. *MSDN Magazine*, 18(9). Available at `http://msdn.microsoft.com/msdnmag/issues/03/09/CoroutinesinNET/default.aspx`.

Smallscript Inc. (2000). S#.NET Tech-preview Software Release. Available at `http://www.smallscript.com/Community/calendar_home.asp`.

Stutz, D. (2002). The Microsoft Shared Source CLI Implementation. Available at http://msdn.microsoft.com/library/en-us/Dndotnet/html/mssharsourcecli.asp.

Ximian (2005). The Mono Project. Available at http://www.go-mono.com/.

All of the links in these references have been verified and are working as of February 2005.