2
"Naive" Compilation

This chapter presents our basic approach for compiling Lua programs to the CLR, along with a series of variations of the basic approach that try to improve performance of generated code without having to perform non-trivial static analysis on the source code (type inference or data-flow based optimizations). We focus on different ways of mapping Lua's values and operations to the Common Language Runtime. Evaluation on the effectiveness of our choices is deferred to Chapter 4, where we present our benchmarks. Although benchmarking was intertwined with the process of coming up with the compilers in this chapter, and has informed this process, we believe that presenting the benchmarks and discussion about them separately leads to a better exposition.

The lack of static analysis and data-flow optimizations results in relatively naive code generation. But the code we generate still has to interact with the CLR's optimizer and JIT compiler, and this interaction is hard to predict even for relatively naive code. The CLR specification [Microsoft, 2005] has no performance model for its intermediate language, and even the performance of a particular implementation depends on the architecture of the machine where it is executing [Blajian et al., 2006, Morrison, 2008a]. The Java Virtual Machine, another managed runtime environment, has the same lack of a clear performance model [Gu et al., 2006, Georges et al., 2007]. We believe that the preferred approach with this lack of performance models is to generate straightforward code and then experiment to discover the impact of any changes, instead of trying to generate optimized code from what we think will be the behavior of the runtime.

Section 2.1 presents the basic compiler and how it is implemented. Section 2.2 presents several variations on this basic compiler, detailing their changes an the rationale for them. Section 2.3 reviews related work on compilation of Lua programs and on compilation of other dynamic languages to managed runtime environments.

Each compiler has a short name that we use to refer to it throughout this chapter and the rest of the dissertation. Table 2.1 lists the names for

each compiler along with its defining characteristic and the section where the compiler is described. References to these names in the text are quoted.

Name	Description
base	Basic Compiler, Section 2.1
single	Single Return Values, Section 2.2.1
box	Boxed Numbers, Section 2.2.2
intern	Interned Strings, Section 2.2.3
prop	Local Type Propagation, Section 2.2.4
infer	Type Inference, Section 3.2

Table 2.1: Compiler names

2.1 Basic Compiler

The basic LuaCLR compiler directly generates CLR Intermediate Language code from an annotated (and desugared) Lua abstract syntax tree. There is no separate intermediate language between Lua and IL, so describing the compiler is a matter of describing the representation of Lua's types by CLR types and of describing the IL code for Lua's operations. We present the representation of Lua values in the next section, and Section 2.1.2 covers how the operations are implemented.

2.1.1 Representing Lua Values

Lua is a dynamically typed language, so values of any of its types can be operands to its operations, and Lua will raise a runtime error if the operands are incompatible. This means that Lua types need an uniform representation and this representation must carry type information that is checked at runtime. Having to use an uniform representation automatically precludes directly using the CLR's primitive types (several types of numbers, and booleans), but still leaves several approaches for representing Lua types.

The approach we use in the basic compiler is essentially the same as the representation we used in Lua2IL [Mascarenhas and Ierusalimschy, 2005], and is similar to the representation the Lua interpreter uses [Ierusalimschy et al., 2005]. The Lua interpreter uses a C *struct* with a type tag and a C *union* with the actual value. The value itself may be a double precision floating point number, a boolean, an external pointer, or a GC-managed pointer. Other types of structs and unions represent values of the GC-managed types, such as functions and tables.

The CLR does not have unions, just structures and classes. We use a `Lua.Value` structure as our uniform representation for all Lua values. A field of this structure is a reference to an instance of a subclass of the abstract `Lua.Reference` class. The other field of this structure is a double precision floating point number (CLR type `double`). The first field acts as a tag that identifies if the value is a number or not; if the first field is `null` then the value is a number, with the number stored in the second field.

Each of the other Lua types is represented by a subclass of `Lua.Reference`. The *nil* type is the singleton class `Lua.Nil`, so the sole instance of this class is the representation of the value `nil`. The `Lua.Bool` class represents booleans, with a single instance for `true` and a single instance for `false`. Strings are represented by instances of the `Lua.String` class, which encapsulates CLR strings, as both Lua and CLR strings are immutable. An important difference between our representation and the Lua interpreter representation is that it is possible to have several instances of the same string with our representation, while the Lua interpreter *interns* strings so there is only one instance of each string. Our representation makes string creation more efficient, at the cost of slower equality testing (and consequently slower table lookups).

We represent tables with instances of the `Lua.Table` class. The internal implementation of this class is similar to the representation used by the Lua interpreter, which has separate hash table and array parts to optimize some accesses using integer indexes. A description of the algorithm is in Ierusalimschy et al. [2005], and implementing it for the CLR is straightforward.

Each function in the program source is represented by its own subclass of `Lua.Reference`. Instances of the function's class are the closures created during execution of the program. Lua functions are first class values that can reference and modify variables in their enclosing lexical scopes, so the closures we create have instance variables that hold references to variables in enclosing scopes that the function uses. In the next section we will cover the internals of closures in greater depth. Each closure also has an instance variable to hold the *environment table*, used to look up the value of global variables.

An important characteristic of our chosen representations is that Lua code never manipulates "native" CLR objects directly, only instances of the `Lua.Value` type. We will see in the next section that this leads to simpler code generation and more efficient generated code. The drawback is that it makes it harder to interface Lua with other CLR code. To interface with other code we can wrap CLR values in a subclass of `Lua.Reference` that delegates operations to the actual CLR type of the value (using reflection, for example).

Any Lua values passed to CLR functions can also be converted to equivalent primitive CLR values. This wrapping can be made transparent to the user, and we have already used this wrapping both in Lua2IL and in a bridge from the Lua interpreter to the CLR [Mascarenhas and Ierusalimschy, 2005, 2004].

2.1.2 Code Generation

The code the "base" compiler generates for most operations is similar: there is code to test if both operands are numbers, then code that does the operation inline if both operands are numbers, and finally a call to a function in the runtime library that does the operation if one of the operands is not a number. These functions of the runtime library are all virtual methods of the base `Lua.Reference` class. So each operation involves a type check and then either the inlined operation or a virtual dispatch to a method that implements the operation.

We inline arithmetic and relational operations on numbers because they are just a single IL instruction. Instead of generating tests and inlined operations we could just compile each operation to a call to a runtime library function that would do the checks, and rely on the CLR JIT's inliner to inline this code. But the type checks and inlined operations are straightforward to generate.

Lua functions are first-class values with lexical scoping, as we saw in Section 1.1. Distinct functions that use the same variable share the same memory location for the variable, instead of each having a distinct memory location that receives a copy of the variable's value at the moment the function is defined, like lexically scoped variables in Python [Mertz, 2001] and the "final" restriction of Java's inner classes [Gosling et al., 2005].

The "base" compiler uses the CLR stack for local variables and argument passing, but the CLR denies access to local variables outside of the current stack frame, and also does not allow references to local variable locations that escape the function where the local variable is defined. To get around these limitations we have to allocate in the heap all local variables that are used outside the function that defines them, and keep just references to these heap cells in the stack. Each closure also has a kind of *display* [Aho et al., 1986, Friedman et al., 2001] that holds references to heap cells of all variables in enclosing scopes that it uses. The basic compiler implements the display as fields in the function's type, so they are instance variables of the closures. When instantiating the function at runtime (at the point in the program where the function is defined) the closure's constructor takes one argument for each

field in its display, and sets the fields in the newly created instance.

For example, take this fragment:

```
local w
function f(x)
  return function (y)
    return function (z) return w + x + y + z end
  end
end
```

The class of the innermost function has fields for variables w, x, and y. The class of its enclosing function has fields for variables w and x, and the class of the outermost function has a field for variable w. All these fields hold a cell for a single Lua value. When instantiating the innermost closure, we emit code to call its constructor passing the value of the w and x fields of the enclosing closure, as well as the value of the y stack slot, which is also a reference to a heap cell holding the value of y. Using references to these heap cells allows two closures to share the same variable. The type of these heap cells is `Lua.UpValue`.

Lua does not have arity errors when calling functions; extra arguments are simply ignored, and missing arguments have value *nil*. Function calls in the basic LuaCLR compiler are compiled to a call to a virtual method of `Lua.Reference` called `Invoke`. `Invoke` is overloaded; `Lua.Reference` has different versions of `Invoke` for different numbers of arguments in the call site, from zero arguments to a maximum fixed by the compiler, plus a version of `Invoke` that takes an array of arguments. The version of `Invoke` that takes an array is used in call sites with more arguments than the maximum or with call sites where the number of arguments is unknown at compile-time. The latter case may happen because Lua functions can return a variable number of values, and when a function call is the last argument to another function call all the values returned by the first call get passed to other call (like when composing functions).

Each function always has a version of `Invoke` that matches the arity of the function, and this version has the code generated from the function body. Other versions of `Invoke` delegate to it, adjusting the number of arguments as necessary. This means that calling a function with a small number of arguments translates to a single CLR virtual call if the number of arguments matches the arity of function. If there is a mismatch there is an adjustment, but then the adjusted call to `Invoke` is a statically dispatched call, not another virtual call.

The classes of all three functions in the example shown in the previous page have `Invoke` methods similar to the ones shown in the fragment of C# code below[1]:

```
Lua.Value[] Invoke() {
  return this.Invoke(Lua.Nil.Instance);
}
Lua.Value[] Invoke(Lua.Value v1) {
  ... code that implements the function ...
}
Lua.Value[] Invoke(Lua.Value v1, Lua.value v2) {
  return this.Invoke(v1);
}
... versions of Invoke with more parameters ...
Lua.Value[] Invoke(Lua.Value[] vs) {
  if(vs.Length > 0)
    return this.Invoke(vs[0]);
  else
    return this.Invoke(Lua.Nil.Instance);
}
```


2.2 Variations of the Basic Compiler

There are several changes that we can make in how we represent Lua types and generate code for operations on these types that trade complexity in the implementation of the compiler for faster programs. In the following sections we present variations of the "base" compiler and the rationale for these variations.

2.2.1 Single Return Values

Lua functions can return any number of values, so all functions in the "base" compiler return an array of `Lua.Value`, even if they only return a single value or the function call needs just the first one. This means that every function call has to allocate an array on the heap to store the return values of the call.

There are several situations where the compiler can be sure that it only needs the first return value, or none at all. Some of these situations are when

[1] We use C# code just to make the fragment shorter, the compiler actually generates IL directly.

a function call is on the right side of an assignment, as in `x = f()`, when a function call is used as a statement, when a function call is not the last argument of another function call, as in `g(f(), x)`, and when a function call is used in a binary operation.

Our first variation of the basic compiler exploits this property by compiling each function twice; one version returns an array, just like in "base", while the other returns a single `Lua.Value` (possibly *nil*). So instead of a `Invoke` method the functions have a `InvokeM` method, for multiple values, and a `InvokeS` method, for single values. They are overloaded to take different number of arguments, as in the basic compiler. In the code for *f*'s `InvokeS`, the code generated for all `return` statements returns the first expression in the statement and just evaluates and discards the others, while in the code for *f*'s `InvokeM` a `return` statement allocates an array, stores the values of its expressions in this array, and then returns it.

For example, in the function call `g(f(), f())` the compiler emits a call to a `InvokeS` method in the first call to *f*, and a call to a `InvokeM` method in the second call to *f*, as all values returned by the second call have to be passed as arguments to the call to *g*.

2.2.2 Boxed Numbers

The previous compiler still uses a `Lua.Value` structure as a uniform representation for Lua values. This structure has two fields: one is used if the value is a number, and the other is a pointer to other types of values, which are all instances of subclasses of the `Lua.Reference` abstract class. This arrangement tries to mimic the representation used by the Lua interpreter (which uses a union instead of a structure), and avoids having to store numbers in the heap.

We will see in Section 2.3 that other compilers for dynamic languages on the CLR choose to represent all values as pointers to objects in the heap, with most compilers using the CLR base `object` type as the common denominator for the values of the language. We follow a similar approach in the variation we present in this section.

Instead of having a `Lua.Value` structure we will use `object` as the type of Lua values; all values are now pointers to either a boxed `double` in the heap, in case the value is a number, or a pointer to an instance of one of the subclasses of `Lua.Reference`, for all other types of values. The representation of these other types remain unchanged, except for trivial changes in the signatures of methods and the internal representation of tables to deal with `object` instead

of `Lua.Value`.

All operations involving numbers in the "base" and "single" compilers first check if the value is a number by checking if the `Lua.Reference` field of the `Lua.Value` structure is `null`, and then unpack the number from the structure, by loading the `double` field. In the "box" compiler these operations become a type test, to see if the value is of type `double`, and an unboxing operation if it is. These are the following fragment in the CLR intermediate language:

isinst `double`
brfalse *notnum*
unbox `double`

After doing the operation, the "base" and "single" compilers had to store `null` in the `Lua.Reference` field and the number that is the result of the primitive operation in the `double` field of the `Lua.Reference` structure that holds the result of the operation. The "box" compiler just does a **box** `double` IL operation that takes the result and boxes it.

In operations that do not involve numbers, the "base" and "single" compilers had to unpack the operand by loading the `Lua.Reference` field from the `Lua.Value` structure and then invoking the correct virtual method. In the "box" compiler this becomes a cast to `Lua.Reference` followed by the virtual dispatch.

This variation presents a case where the high-level nature of the intermediate language of a managed runtime environment and a lack of information on how the optimizer of this runtime translates this intermediate language to machine code makes it hard to assess if a particular change improves or not the execution time of compiled programs.

The number of intermediate language instructions to do each operation in the "base", "single", and "box" compilers is roughly the same, but the cost of these instructions is unpredictable. We have to use benchmarks to evaluate how each approach performs. The change of unboxed to boxed representations can have wildly different performance characteristics depending on how the runtime's heap allocator and garbage collector work, and even whether the runtime optimizes boxing and unboxing of some numbers. For example, one possible optimization a runtime can do is to pre-allocate a range of boxed numbers, like small integers, and keep reusing them instead of allocating new objects in the heap.

2.2.3 Interned Strings

The Lua interpreter *interns* all strings so each string has only one copy. This lets the test of whether two strings are equal be a simple test of pointer equality, instead of a test of the strings' contents. Tests of string equality are always used when indexing tables with string keys, so interning strings makes indexing using string keys faster. Indexing operations with string keys are a common operation in Lua, specially in OO-style code, as field accesses (`obj.field`) and method calls (`obj:method()`) get desugared to indexing operations with string keys.

Our previous compilers represent Lua strings with the `Lua.String` subclass of `Lua.Reference`, and the internal representation is just a CLR `string`. Like Lua strings, CLR strings are immutable, but the CLR specification does not dictate how strings should be implemented, so implementations are free to intern strings or not. Not interning is a better choice when string creation (as a result of concatenation, for example, or slicing) is more common than testing equality. In this section we present a variation of the "box" compiler that tries to optimize equality tests, and benchmarking this variation against "box" tells whether a particular CLR implementation is interning its strings or not.

One optimization we can do is to implement the equality test between two `Lua.String` objects as a pointer equality test first, with the CLR string equality test done only when the pointer test fails. We then make sure all string literals in the program, including desugared field and method names, have only one instance. This completely avoids regular string comparison in indexing operations if the particular `Lua.String` key is already in the table and there are no hash collisions.

The optimization we actually do in the "intern" compiler is applicable to a greater number of indexing operations; we add a new type for interned strings that we call *symbols*, the type `Lua.Symbol`. This is a subtype of `Lua.String`, the only difference being that symbols are interned in a global hashtable, so two symbols can safely be tested for equality by pointer equality. All string literals are symbols, and tables intern all strings that are used as keys, so all internal equality tests in a hash lookup are done between symbols. Any indexing operation that has a symbol key only needs pointer equality, even in the case of collisions or if the key is not in the hash. This includes all indexing operations caused by Lua's OO syntactical sugar.

Operations that create new strings still create instances of `Lua.String` instead of `Lua.Symbol`, so they do not have to pay the cost of interning strings.

If the CLR implementation does not intern strings then this variation should speed up OO-style code without slowing down code that does string processing.

2.2.4 Local Type Propagation

Working with boxed numbers can have a big performance impact on code that does a lot of arithmetic, even if the runtime tries to optimize boxing and unboxing operations and has a good memory allocator and garbage collector. The `Lua.Value` representation of the "base" and "single" compilers avoids boxing and unboxing, but wastes memory and uses a runtime feature, structured value types allocated in the stack, that also relies on optimizations by the runtime. If we statically know that we are only dealing with numbers we can avoid both boxing and structures and use the native `double` type of the CLR directly.

In Chapter 3 we present a way of extracting this information with *type inference*, and show how it can have other uses besides just avoiding boxing of numbers. In this section we present a much simplified, and local, version of type inference. It is local because it does not try to infer types across function call boundaries. It types variables and expressions in a function as one of seven simple types: nil, boolean, number, string, function, table, and the type *any*, which means that the variable (or expression) can have any type. There are no structural types: having a type "function" or "table" just means that operations can be dispatched to methods of `Lua.Function` or `Lua.Table` without doing type checks; all functions have type "function" and all tables have type "table". Function calls and indexing operations always have type "any", and "any" is also the type of the parameters of a function.

The typing rules are straightforward. For assignment there are three cases, which depends on the types of the lvalue (the left side of the assignment) and the rvalue (the right side of the assignment):

1. If the lvalue does not have a type yet, then it gets the type of the rvalue;
2. if the lvalue has the same type as the rvalue, then nothing changes;
3. if the lvalue and rvalue have different types, then the lvalue now has type "any".

The rules for binary arithmetic operations are that the type of the operation is "number" if the type of both operands is "number", otherwise it is "any". In Lua, as a consequence of the metatables we mentioned in Section 1.1, binary arithmetic operations are dispatched to user-defined operations if one

of the operands is not a number, and our type inference has to assume that these operations can return anything.

The representation that the "prop" compiler uses for most types is the same as the one the "intern" compiler uses. The only type that has a different representation is "number", which uses the `double` type of the CLR. The type "any" is an `object` pointer to either a boxed `double` or an instance of a subclass of `Lua.Reference`, and other types continue to be represented as subclasses of `Lua.Reference`.

We implement the type inference as an iterated traversal of each the abstract syntax tree of each function. The inference stops when the types converge. All the typing rules have the invariant that if the type of a term is "any" it cannot change, and if the type of a term is not "any" then it can either stay the same or change to "any", so convergence is guaranteed.

Changes to the code generator are straightforward, with simpler code for arithmetic expressions when the operands are numbers, and elimination of type checks prior to calls to methods of `Lua.Reference` whenever possible.

2.3 Related Work

In this section we review previous work on compiling dynamic languages to managed runtime environments. We focus on compilers targeting the Common Language Runtime first, as it was the first managed runtime specifically created as a shared runtime for different high-level languages, with emphasis on interoperation among these languages [Hamilton, 2003].

In 1999 and 2000 Microsoft sponsored the development of a compiler for the Python scripting language, written by Mark Hammond and Greg Stein. The compiler supported most of the Python language and allowed Python programs to interface with CLR code, but the authors judged the performance to be "so low as to render the current implementation useless for anything beyond demonstration purposes" [Hammond, 2000]. This poor performance was for "both the compiler itself, and the code generated by the compiler" [Hammond, 2000]. The authors abandoned the effort in 2002.

Python for .NET used a type mapping similar to the one we used on our basic compiler, with a CLR structure representing Python values, but the authors noted that "simple arithmetic expressions could take hundreds or thousands of Intermediate Language instructions", so the operations were inefficient. All the operations were dispatched to the Python for .NET runtime, as the compiler did not do any inlining.

Common Larceny [Clinger, 2005] was an early Scheme compiler for the

CLR that compiled MacScheme assembly code, used as an intermediate language in the Larceny family of Scheme compilers [Clinger and Hansen, 1994], to CIL instructions. It used instances of a `SchemeObject` class to represent Scheme values, so all values were pointers to objects in the heap; Common Larceny preallocated booleans, characters, and small integers. Common Larceny used its own stack for both values and control information, to make the implementation of Scheme's first-class continuations easier. The performance of the compiled Scheme code was found to be similar to the performance of the code running on the MzScheme interpreter [Clinger, 2005].

Bigloo.NET [Bres et al., 2004] was a later Scheme compiler for the CLR which, like Common Larceny, was part of a family of Scheme compilers that targeted different platforms (the Bigloo family [Serrano and Weis, 1995]). Bigloo.NET was heavily derived from the BiglooJVM compiler [Serpette and Serrano, 2002]. Its usual representation for Scheme values was a pointer (of `object` type) to a value in the heap that could be a boxed number, a byte array (for strings), or instances of classes that represented other Scheme types. Bigloo.NET could also use an interprocedural flow analysis [Serrano and Feeley, 1996] augmented with type annotations to use unboxed representations for scalar values.

In contrast to Common Larceny, Bigloo.NET used the CLR's stack for control-flow, argument passing, and local variable storage, and did not have a complete implementation of Scheme continuations. Bigloo.NET implemented all closures in the same Scheme module as instances of the same class derived from `bigloo.procedure`, with an index that identified the specific closure's entry point plus an array for the closure's display. The code for all the functions in a module was compiled to different entry points in the same CLR method, indexed by a switch statement, because the authors claimed this was better than having each closure be a separate subclass of `bigloo.procedure` for code that makes heavy use of closures. Performance of Bigloo.NET was found to be two to six times slower than the performance of BiglooC, a compiler of the Bigloo family that compiles Scheme to C code, and about twice as slow as BiglooJVM.

Lua2IL was another project for running Lua code inside the CLR, and worked by translating Lua 5.0 bytecodes to CIL instructions, with the help of a support runtime [Mascarenhas and Ierusalimschy, 2005]. Lua2IL used the same mapping from Lua types to CLR types as our "base" compiler, with a structure holding either a Lua number of a reference to other Lua types, and the other types represented by CLR classes with a common subclass. Lua2IL also inlined operations on numbers.

Where Lua2IL differs form our "base" compiler is on the treatment of local variables, function arguments and upvalues. Lua2IL kept a parallel stack for Lua values in the CLR heap and threaded this stack through the compiled Lua code, but still used the CLR stack for control. Code generated by Lua2IL had performance similar to the same code when executed by the Lua 5.0.2 interpreter.

The research prototype of IronPython, another Python compiler for the CLR, showed that Python could have good performance in the CLR if the compiler was designed with careful consideration to performance [Hugunin, 2004]. IronPython boxed numbers and cast other types to `object`, like our "boxed numbers" compiler, and also inlined common operations.

Later versions of IronPython have focused on generalizing its runtime to other dynamic languages, building a *Dynamic Language Runtime* on top of the CLR [Hugunin, 2008, Chiles and Turner, 2009]. The DLR uses a generalization of inline caches [Deutsch and Schiffman, 1984, Hölzle and Ungar, 1994, Hölzle et al., 1991] to implement operations, called *dynamic sites*. Its runtime system generates CIL code that makes heavy use of static method calls, relying on the CLR JIT for inlining and optimization. The implementation of dynamic sites uses a complex runtime system and extensive runtime code generation [Turner and Chiles, 2009]. Recent work on IronPython has moved to mixed-mode execution, with an interpreter for DLR syntax trees as the main mode of execution and the compiled dynamic sites for hotspots, as the extensive code generation needed by the dynamic sites was found to be too heavyweight [Hugunin, 2009].

Currently Microsoft has compilers that target the DLR for Python (the IronPython compilers cited above), Ruby [Lam, 2009], and JavaScript [Dhamija, 2007], and the set of the DLR features is biased towards these languages. In particular, there is no native support for multiple return values from functions, as these three languages implement multiple return values using tuples or arrays. Tail call optimization is also not present in the DLR, despite the CLR supporting for it. The semantics of Lua require tail call optimization, and efficient implementation of Lua function calls requires support for returning multiple values. Another issue is the implementation of lexical scoping and varargs in the DLR, which revert to building stack frames in the heap instead of using the CLR stack.

IronScheme [Pritchard, 2009] is a Scheme R^6RS [Sperber et al., 2007] compiler for the CLR in active development that uses the DLR, but it uses a modified version of the DLR that implements the extra functionality needed to compile the Scheme language. The author estimates that the modified DLR

used by the IronScheme compiler uses only 20% of code the original DLR, and wants to investigate the viability of writing his own code generator instead of using the DLR [Vastbinder, 2008].

Phalanger [Benda et al., 2006] is a PHP compiler for the CLR. The unit of execution in PHP is a script, and Phalanger compiles a script to a CLR class, with the body of the script as a `Main` method of this class and other functions declared in the script as static methods of the class. Each function is compiled to two methods, one that takes the function's arguments as formal parameters, so uses the CLR stack, and another method that takes arguments in an array and delegates to the other method, for function calls where the target is unknown. Phalanger represents all values as a pointer of type `object`, using a combination of boxing for PHP types that have corresponding CLR primitive types (such as numbers), and subclasses of a `PhpObject` class for other PHP types such as classes and interfaces.

Cuni et al. [2009] use a generalization of polymorphic inline caches called *flexswitches* to implement a JIT compiler for a toy dynamic language targeting the CLR, so there are two layers of JIT compilation, the flexswitch-based JIT compiler and the CLR JIT compiler. While a polymorphic inline cache optimizes just the specific operation at the site of the cache, and does not affect other operations, a flexswitch can call back to the flexswitch-based JIT compiler to generate specialized code for code reachable from the flexswitch. This approach can generate code that is very efficient, but the extra level of compilation and recompilation adds considerable overhead, with most of the total running time of the microbenchmarks in the paper being compilation overhead.