# Design of the Portable.NET Interpreter

Rhys Weatherley, Southern Storm Software, Pty Ltd, Australia*
Gopal V, Independent DotGNU Contributor, India†

January 6, 2003

## Abstract

Portable.NET[1] is an implementation of the Common Language Infrastructure (CLI). Its primary design goal is portability to as many platforms as possible, which it acheives through the use of interpretation rather than Just-In-Time compilation.

The bytecode format of the CLI presents some challenges to efficient interpreter implementation. Rather than directly interpret, we translate the bytecode into a simpler abstract machine; the Converted Virtual Machine (CVM). This machine is then interpreted using a high-performance engine.

Traditionality, abstract machines have used the same bytecode representation "on the wire" as for execution. Our work shows that there are definite performance advantages to using different bytecode representations internally and externally.

## 1 Introduction

The Common Language Infrastructure (CLI) is a set of specifications that describe a bytecode-based development and runtime environment [2]. Portable.NET is an implementation of the CLI, whose primary design goal is portability to as many platforms as possible.

Portable.NET consists of three major components to support the CLI: a bytecode-based Common Language Runtime (CLR), a C# compiler, and a C# base class library. This article will concentrate on the runtime engine.

The runtime engine achieves portability primarily through the use of interpretation rather than Just-In-Time compilation. However, the bytecode format of the CLI presents some challenges to efficient interpreter implementation. This article discusses how we have overcome these challenges to build a high-performance interpreter for Common Intermediate Language (CIL) programs.

We compare the performance of Portable.NET against Mono [3] to demonstrate how our approach differs from direct polymorphic interpretation and full Just-In-Time compilation.

*rweather@southern-storm.com.au
†gopalv82@symonds.net

```
case CEE_LDC_I4_0:
  sp->type = VAL_I32;
  sp->data.i = 0;
  ++sp;
  ++ip;
  break;

case CEE_LDC_I8:
  ++ip;
  sp->type = VAL_I64;
  sp->data.l = read64(ip);
  ip += 8;
  ++sp;
  break;
```

```
case CEE_ADD:
  ++ip;
  --sp;
  if(sp->type == VAL_I32) {
    sp[-1].data.i += sp->data.i;
  } else if(sp->type == VAL_I64) {
    sp[-1].data.l += sp->data.l;
  } else if(sp[-2].type == VAL_DOUBLE) {
    sp[-1].data.f += sp->data.f;
  } else {
    ...
  }
  break;
```

Figure 1: Polymorphic interpretation

# 2 Polymorphic CIL

The CIL instruction set contains instructions to perform arithmetic, logical operations, branching, method calls, object accesses, and pointer manipulation.

A unique feature of CIL, compared to other similar abstract machines, is that its instructions are polymorphic. The `add` instruction can be used on integers, longs, and floating-point values, for example. Other virtual machines, such as the Java Virtual Machine (JVM)[4], use separate instructions for each type.

The polymorphic nature makes direct interpretation very inefficient, as demonstrated by the fragment from Mono's interpreter shown in Figure 1. The types of all stack values must be tracked explicitly, leading to significant runtime overhead.

# 3 CVM instruction set

The challenge for Portable.NET was finding a way to implement a high-performance engine without writing a full Just-In-Time compiler.

The approach we took was very similar to a JIT: the CIL bytecode is translated into instructions for a simpler abstract machine, dubbed CVM (for "Converted Virtual Machine"). The CVM instructions are then interpreted using a high-performance interpreter, written in C.

As each method is entered, the following process occurs:

1. Look for a cached CVM version of the method, and use it if found.

2. Perform bytecode verification and convert the CIL into CVM.

3. Record the CVM version in the cache.

```
case COP_IADD:
  sp[-2].intval += sp[-1].intval;
  --sp;
  ++pc;
  break;

case COP_LADD:
  *((ILInt64 *)(&(sp[-(WORDSPERLONG * 2])))) +=
      *((ILInt64 *)(&(sp[-WORDSPERLONG])));
  sp -= WORDSPERLONG;
  ++pc;
  break;

case COP_FADD:
  *((ILNativeFloat *)(&(sp[-(WORDSPERFLOAT * 2])))) +=
      *((ILNativeFloat *)(&(sp[-WORDSPERFLOAT])));
  sp -= WORDSPERFLOAT;
  ++pc;
  break;
```

Figure 2: Interpreting converted instructions

4. Jump into the interpreter to execute the CVM code.

Eventually the application's working set of methods ends up in the CVM method cache, and execution proceeds quickly.

Instead of a single `add` instruction, the CVM instruction set has several: `iadd`, `ladd`, `fadd`, etc. The conversion process chooses the most appropriate variant, based on the operand types reported by the bytecode verifier.

Figure 2 shows a simplified form of the CVM interpreter code for the converted instructions. The interpreter executes more efficiently because it can assume that the values on the stack are of the correct type (bytecode verification having already been performed).

Items on the CVM execution stack are a uniform size of one word: 64-bit and larger types straddle multiple words. The CVM conversion process takes care of laying out the stack according to the types of local variables and stack items.

This isn't necessarily a new approach - it is normally known as "threaded interpretation" in the Forth community [5].

The complete list of CVM instructions is given in Figure 4 at the end of this article.

# 4 Ramping Up

Conventional wisdom says that one should write a hand-crafted assembly code loop to get a fast interpreter. However, there are some simple tricks that can be used to speed up an interpreter, even in C code.

1. Register variables.
2. Computed gotos.
3. Direct threading.
4. CPU-specific unrolling.

C compilers aren't terribly good at determining which values are most-used in switch-loop interpreter code. The compiler invariably guesses wrong, favouring temporaries over important variables like the program counter and stack pointer. So it is necessary to "help" the compiler a little.

The gcc compiler can bind variables to explicit registers, as follows:

```
register unsigned char *pc
    __asm__ ("esi");
```

We placed the program counter, the top of stack pointer, and the frame pointer into x86 registers. This produced a significant improvement in performance compared to straight C code, for such a small change.

The next step was to change from `switch` statements to using computed goto's. This is normally referred to as *token threading*. The `break` at the end of each case is replaced with a `goto` statement:

```
goto *main_label_table[*pc];
```

The `main_label_table` contains pointers to each of the cases in the `switch` statement, allowing the interpreter to jump directly to the next case, avoiding the overhead of jumping back to the top of the switch-loop. More information on computed gotos can be found in the gcc documentation [6].

The result of these two changes (explicit registers and token threading) was an interpreter that was so close to a hand-crafted assembly loop that there was little point writing one by hand.

The third step involved a change in representation. The switch loop and token threaded versions select instruction handlers based on CVM bytecode. Instead of storing the single-byte opcodes, we can store the actual addresses of the opcode handlers in the CVM instruction stream. This is known as *direct threading*.

```
goto **((void **)pc);
```

Direct threading increases the size of the CVM code by a factor of 4, because instructions are now pointers to handlers rather than bytecodes. But it avoids the overhead of looking up values in `main_label_table`. On RISC platforms, this can give a signficiant increase in engine performance, but on x86 it isn't too impressive. If memory is an issue, token threading gives better results.

# 5 Unrolling

Direct threading really shines when combined with some native JIT techniques. We implemented a "mini JIT" that converted simple CVM instruction sequences into x86 machine code on the fly. We call this an *unroller* because it essentially unrolls the interpreter loop into straight-line machine code.

The unroller uses simple register allocation techniques on the basic blocks of a

| | Switch | Regs | Token | Direct | Unroll | Mint | Mono |
|---|---|---|---|---|---|---|---|
| Sieve | 499 | 784 | 1342 | 1583 | 6568 | 144 | 10040 |
| Loop | 504 | 779 | 1104 | 1277 | 13013 | 119 | 19517 |
| Logic | 490 | 724 | 1933 | 2378 | 7266 | 204 | 16311 |
| String | 1038 | 1110 | 1158 | 1089 | 1139 | 495 | 1307 |
| Float | 66 | 87 | 117 | 129 | 698 | 11 | 220 |
| Method | 430 | 668 | 1297 | 1471 | 3457 | 159 | 14977 |
| PNetMark | 392 | 552 | 891 | 998 | 3456 | 120 | 4895 |
| % Mono | 8% | 11% | 18% | 20% | 71% | 2% | 100% |

Figure 3: Comparison of different engines using PNetMark

method. Complex instructions, especially those involving method calls, are not unrolled. It isn't possible for unrolling to achieve the same performance as a JIT, but it can get very close.

The primary advantage of the unroller compared to a JIT is that it is vastly simpler to implement. Portable.NET's x86 unroller took about two weeks to write, and we expect that other CPU's would require a similar amount of effort.

Anything that is too complicated to convert is replaced with a jump back into the interpreter core. This allows unrollers to be developed in stages, replacing one instruction at a time and then re-testing. This made development a lot easier than the "all or nothing" approach required for a JIT.

# 6 PInvoke

The "platform invoke" (or PInvoke) feature is a very powerful mechanism that CLI programs can use to call legacy native code.

When Portable.NET encounters a PInvoke method reference, it compiles a small CVM stub which performs any necessary parameter marshaling and then calls the underlying native function. Upon return, the CVM stub de-marshals the return value.

A similar process is used for "internal-call" methods within the runtime engine that implement builtin features for the C# class library.

Using CVM to perform marshaling operations simplifies native function invocations quite considerably. Only a small amount of platform-specific code is needed to perform the native call, for which we use the standard "libffi" library.

# 7 Inlining

Method call overhead is an issue for all interpreter-based abstract machines because method calls are more complicated than the equivalent native code.

CVM addresses this by selectively inlining some of the more commonly used methods in the C# class library. The method call is replaced with a special-purpose opcode during code conversion.

The major groups of inlineable methods within Portable.NET are currently the string, monitor, and 2D array operations.

Inlining common methods can have a dramatic impact on performance. The PNetMark "Float" benchmark improved by a factor of 12 when 2D array operations were inlined. Such operations are normally very expensive in CLR's because a method must be called for every element get or set operation.

# 8 Alternate backends

The construction of the interpreter itself was made as modular as possible. The interface between the metadata handling and the execution was kept seperate using a standard interface: the coder API.

Coders are an interface between the CIL frontend and the execution engine. The design allows for the current CVM backend to be replaced by a fully-fledged JIT without any modification to the other components in the system. The same frontend could be used with the CVM engine, a native JIT, or even a polymorphic interpreter.

# 9 Performance summary

Figure 3 compares the CVM interpreter variants with the Mint polymorphic interpreter and the Mono x86 JIT. All tests were done on an 866 MHz Pentium III, running RedHat Linux 7.1. Version 0.17 of Mono and version 0.5.0 of Portable.NET were used for these comparisons.

As can be seen, the simple techniques described in this article produce very good results, with the unrolled version acheiving 71% overall compared to Mono's JIT.

# 10 Future Work

Portable.NET's interpreter remains a work in progress. More optimizations are possible by introducing new CVM instructions for special cases.

We are also investigating *selective inlining* [7] as an alternative to writing a handcrafted unroller for each CPU. The authors of that paper also reported performance of up to 70% of optimized C code using simple techniques. Their engine, for Objective Caml, has a single line of platform-specific code, to perform a flush of the CPU's instruction cache. Selective inlining doesn't work very well for x86, but it should do well for RISC CPU's like the PowerPC.

In the near future, we will be investigating fully-fledged JIT coders for Portable.NET, as well as front ends for other instruction sets such as the Java Virtual Machine.

# 11 Conclusion

Using a variety of well-known, yet simple, techniques, Portable.NET is able to achieve adequate performance for most application-oriented tasks.

At the same time, the code is highly portable. Ports to new platforms take a matter of days, sometimes hours (e.g. the author ported the code to MacOSX in a single day).

# 12 Acknowledgments

Portable.NET would not have been possible without the generous assistance of volunteers from the DotGNU community [8].

## References


[1] http://www.southern-storm.com.au/portable_net.html.

[2] Common Language Infrastructure (CLI), Partitions I to IV.
ECMA 335, European Computer Manufacturers Association, 2001.

[3] http://www.go-mono.com/.

[4] Tim Lindholm and Frank Yellin.
*The Java Virtual Machine Specification, Second Edition.*
Addison-Wesley, 1999.

[5] M. Anton Ertl.
A Portable Forth Engine.
In *Proc. euroFORTH '93*, pages 253–257, 1993.
http://www.complang.tuwien.ac.at/forth/threaded-code.

[6] http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Labels-as-Values.html.

[7] Ian Puimarta and Fabio Riccardi.
Optimizing direct threaded code by selective inlining.
In *SIGPLAN '98*, pages 291–300. ACM Press, 1998.

[8] http://www.dotgnu.org/.

ansi2str array2ptr array_len beq bfixup bge bge_un bgt bgt_un ble ble_un bload blt blt_un bne box box_ptr br bread bread_elem bread_field break brfalse br_long brnonnull brnull br_peq br_pne brtrue bstore bwrite bwrite_elem bwrite_field bwrite_r call call_ctor call_interface call_native call_native_raw call_native_void call_native_void_raw call_virtual castclass castinterface cctor_once ckarray_load_i4 ckarray_load_i8 ckarray_store_i8 ckfinite ckheight ckheight_n cknull cknull_n delegate2fnptr dfixup dread dread_elem dup dup2 dup_n dup_word_n dwrite dwrite_elem dwrite_r enter_try f2d f2d_aligned f2f f2f_aligned f2i f2i_ovf f2iu f2iu_ovf f2l f2l_ovf f2lu f2lu_ovf fadd fcmpg fcmpl fdiv ffixup fix_i4_i fix_i4_u fmul fneg fread fread_elem frem fsub fwrite fwrite_elem fwrite_r get2d get_static i2b i2b_aligned i2b_ovf i2f i2iu_ovf i2l i2p_lower i2s i2s_aligned i2s_ovf i2ub i2ub_ovf i2ul_ovf i2us i2us_ovf iadd iadd_ovf iadd_ovf_un iand icmp icmp_un idiv idiv_un iload iload_<n> imul imul_ovf imul_ovf_un ineg inot ior iread iread_elem iread_field iread_this irem irem_un ishl ishr ishr_un isinst isinterface istore istore_<n> isub isub_ovf isub_ovf_un iu2b_ovf iu2f iu2i_ovf iu2l iu2s_ovf iu2ub_ovf iu2us_ovf iwrite iwrite_elem iwrite_field iwrite_r ixor jsr l2f l2i l2i_ovf l2ui_ovf l2ul_ovf ladd ladd_ovf ladd_ovf_un land lcmp lcmp_un ldc_i4 ldc_i4_<n> ldc_i4_s ldc_i8 ldc_r4 ldc_r8 ldftn ldinterfftn ldiv ldiv_un ldnull ldstr ldtoken ldvirtftn lmul lmul_ovf lmul_ovf_un lneg lnot local_alloc lor lread_elem lrem lrem_un lshl lshr lshr_un lsub lsub_ovf lsub_ovf_un lu2f lu2i_ovf lu2iu_ovf lu2l_ovf lwrite_elem lxor maddr memcpy memmove memset memzero mk_local_1 mk_local_2 mk_local_3 mk_local_n mkrefany mload monitor_enter monitor_exit mread mstore mwrite mwrite_r new new_value nop pack_varargs padd_i4 padd_i4_r padd_i8 padd_i8_r padd_offset padd_offset_n pcmp pload pload_<n> pop pop2 pop_n pread pread_elem pread_field pread_this prefix pstore pstore_<n> psub psub_i4 psub_i8 pushdown push_thread push_thread_raw pwrite pwrite_elem pwrite_field pwrite_r refanytype refanyval refarray2ansi refarray2utf8 ret_jsr return return_1 return_2 return_n set2d seteq setge setgt setle setlt setne set_num_args sfixup squash sread sread_elem sread_field str2ansi str2utf8 string_concat_2 string_concat_3 string_concat_4 string_eq string_get_char string_ne switch swrite swrite_elem swrite_field swrite_r tail_call throw throw_caller type_from_handle ubread ubread_elem ubread_field unroll_method usread usread_elem usread_field utf82str waddr waddr_native_<n> wide

Figure 4: Complete CVM instruction set