Translating .Net Libraries To Parrot

Jonathan Worthington

YAPC::EU::2006

The Problem

Love virtual machines did he,
Shared libraries made his day.
But libraries for VM B,
Wouldn't work on VM A.

Motivation

- Virtual machines are good.
    - Abstract away the operating system and hardware, easing deployment
    - May provide higher level constructs than real hardware, so easier to compile to
    - Safety and security benefits
    - Inter-operability between languages

Motivation

- Shared libraries are good.
  - More generally, code re-use in general is good
- For libraries compiled to native (machine) code, calling into them is easy…
  - Common calling conventions…
  - …and a jump instruction.

Motivation

- What about libraries written in languages that run atop of a VM?
- Fine if they both compile down to (or libraries are available for) both VMs.
- If not there's a problem!
- Different VMs have different instruction sets, provide different levels of support for HLL constructs, etc.

Possible Solution #1

- Modify the compiler for the HLL to emit code for another VM.
  - ✓ Can lead to high quality output code.
  - ✗ Need source of HLL compiler and the library – maybe not available!
  - ✗ If there are libraries in multiple HLLs, we have multiple compilers to modify.
  - ✗ Need to worry about HLL semantics.

Possible Solution #2

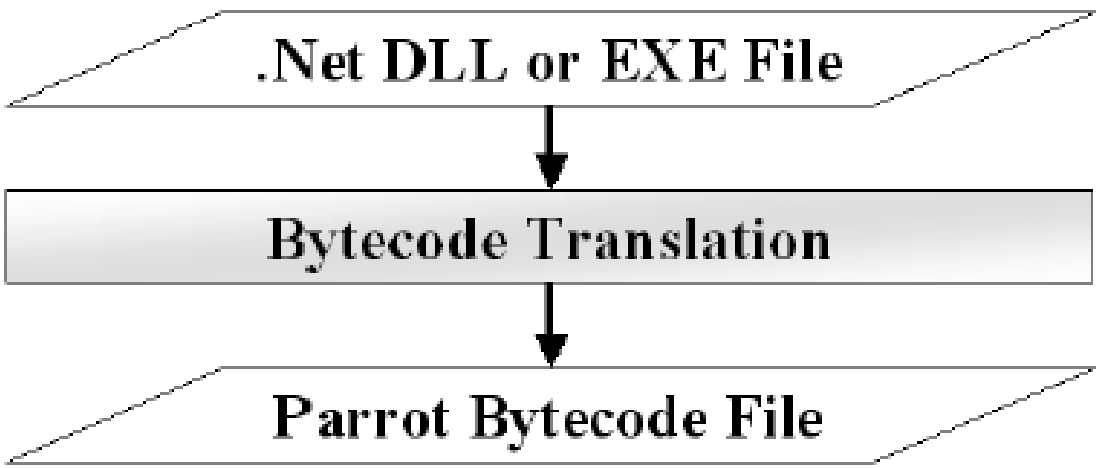
- Embed one VM inside another.
  - ✓ A quick way to something that basically works.
  - ✓ No issues matching semantics.
  - ✗ Making calls into the other VM transparent means duplicating state.
  - ✗ Have memory footprint of both VMs
  - ✗ Performance issues over boundary

Possible Solution #3

- Translate bytecode for VM A to bytecode for VM B.
  - ✓ Independent of the HLL
  - ✓ Translating a small(ish) number of well defined instructions
  - ✓ VM B's "native" code => performance
  - ✗ A lot of initial implementation effort to get something usable.

The Chosen Solution

- Bytecode translation appeared to be the best compromise, so I went with that.
- Chose to translate .Net bytecode to run on the Parrot VM.



Planning

So a translator he conceived;
Designed so it would be,
Declarative and pluggable,
To manage complexity.

Why It's Hard

- Parrot is a register machine, while .Net is a stack machine.
- A .Net library isn't just a sequence of instructions, but metadata too.
  - Set of tables listing classes, fields, methods, signatures, etc.
- Some .Net instructions/constructs have no direct Parrot equivalent.

Other Issues

- Code to translate an instruction will often be pretty similar. Repetitive code is bad.
- Multiple solutions to mapping stack code to register code; want to have simple one at first, the implement and benchmark advanced ones later.
- Want reasonably high performance from the translator.

Metadata Translator

- Partly written in C (reading the .Net assembly), partly in PIR (code generation).
- C-PIR interface through PMCs (Parrot types implemented in C).
- Can generate class and method stubs with the metadata translator; instruction translator fills in the method bodies with the translated code.

Declarative Instruction Translation

- Create a declarative "mini-language" to specify how to translate instructions.

```
[add]
code = 58
class = op
pop = 2
push = 1
instruction = ${DEST0} = ${STACK0} + ${STACK1}
typeinfo = typeinfo_bin_num_op(${STYPES}, ${DTYPES})
```

.Net instruction name and number.

Type of instruction (branch, load, …)

Number of items it takes from/puts onto the stack

The Parrot instruction to generate.

Type transform

Pluggable Stack To Register Mapping

- Need to turn stack code into register code.
- Ideally, want a translation like this:

```
ldc.i4 30
ldc.i4 12
add
stloc.1
```

```
add local0, 30, 12
```

Pluggable Stack To Register Mapping

- Want to do something easy first.
  - Use a Parrot array PMC to emulate the stack => slow, but simple.
  - Pop stuff off the stack into registers to do operations on them.

```
ldc.i4 30
ldc.i4 12
add
stloc.1
```

```
push s, 30
push s, 12
$I0 = pop s
$I1 = pop s
$I2 = add $I0, $I1
push s, $I2
```

Pluggable Stack To Register Mapping

- Later, want to implement something more complex.
- So make stack to register mapping pluggable.
  - Define set of hooks (pre_branch, post_branch, pre_op, post_op, etc.)
  - Stack to register mapping module implements these.

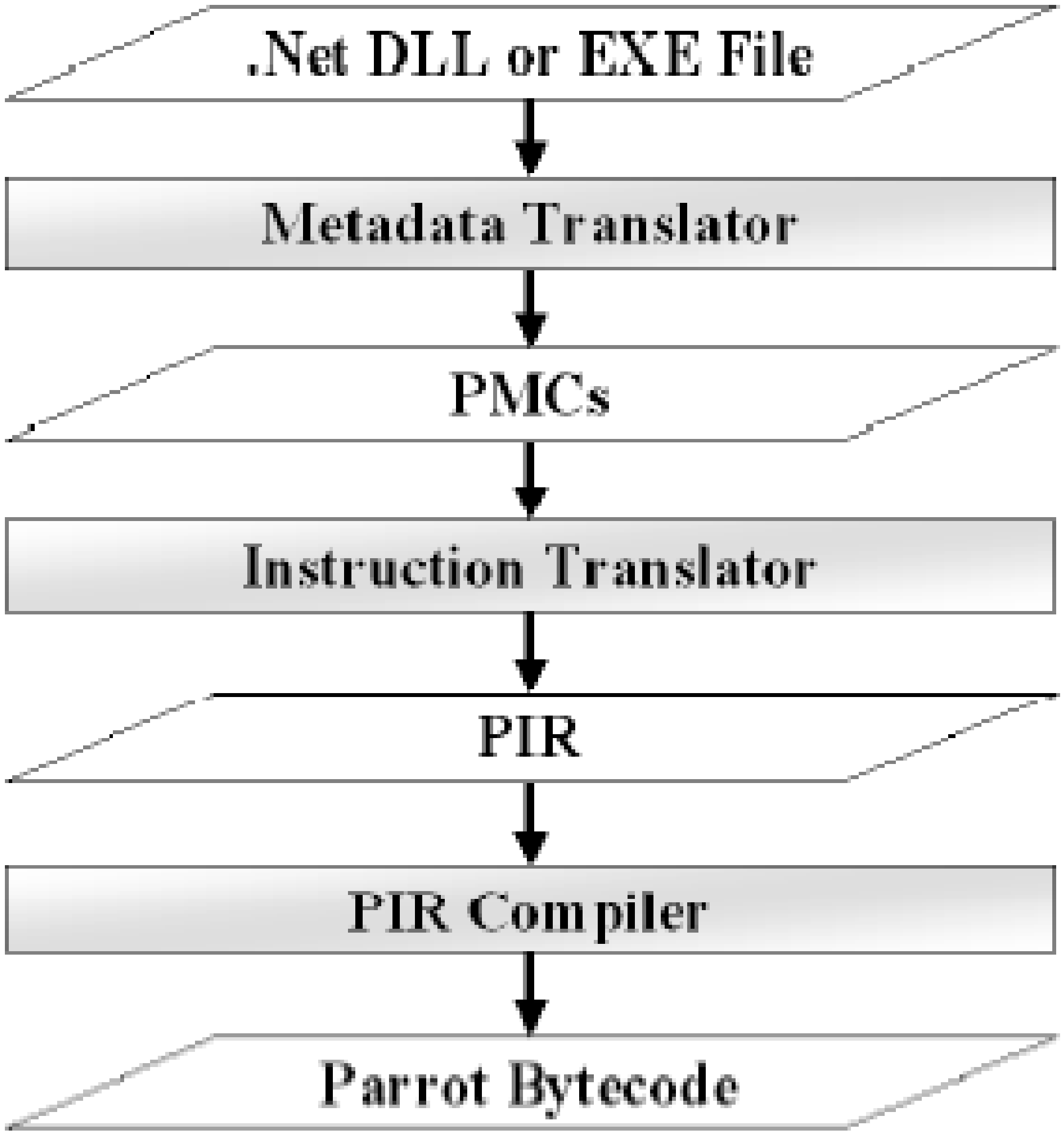
Stack Type State Tracking

- When data is placed on the stack, we always know its type (integer, float, object reference, etc).
- But “add” instruction (for example) could be operating on integers or floats => need to map stack locations to correct Parrot register types.
- Track the types of values on the stack using simple data flow analysis.

Building The Translator

- The translator generator (written in Perl) takes…
  - A file of instruction translation declarations.
  - A stack to register mapper (also written in Perl, generating PIR code)
- Outputs a translator in Parrot Intermediate Representation (PIR).

Overall Design



Implementation

For weeks he toiled day and night,
Fuelled by chocolate and caffeine,
And wove his dreams into code:
A translator like none e'er seen!

Early Days (Oct – Nov)

- The metadata translator was partially implemented first (since the instruction translated depended on it).
- Generated class and method stubs.
- Method stubs did parameter fetching and local variable declaration.
- Stress tested with large DLLs from the .Net class library.

Basic Instructions (Nov to Dec)

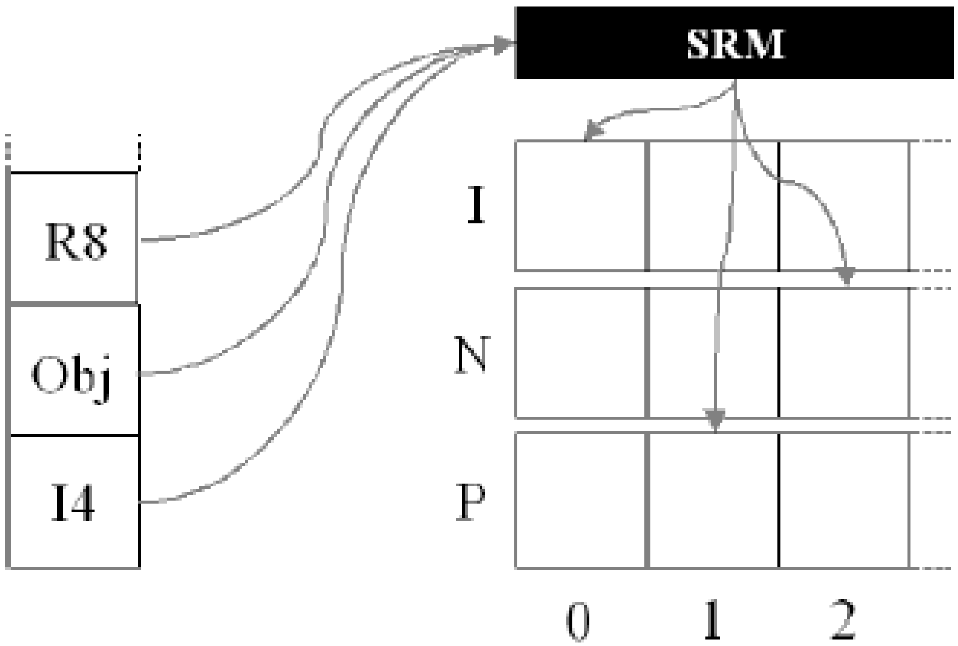
- Instruction translator implemented as described earlier.
- Wrote translation rules for arithmetic and logical operations, load and store of local variables and parameters and branch instructions.
- Regression testing all of these from the start.

Then It Got Harder…

- Work in 2005 had been about building translation infrastructure and getting some basic translation going.
- Work in 2006 involved translating more complex instructions and constructs.
- Many of them described in detail in The Dissertation (on the conference CD); won't look at them here.

A More Advanced SRM

- Wanted to generate better register machine code.
- Idea (from paper!): map each stack location to a register.



A More Advanced SRM

- Means that we don't need to emulate the stack – much better performance.
- Real register code, so the optimizer has a chance.
- But still lots of needless data copying…

```
ldc.i4 30
ldc.i4 12
add
stloc.1
```

→

```
$I0 = 30
$I1 = 12
$I2 = add $I0, $I1
local1 = $I2
```

A More Advanced SRM

- Idea: do loads of constants, local variables and parameters lazily.
- Instead of emitting a register copy, store the name of the source register.
- Emit that directly into instruction that uses it.

```
ldc.i4 30
ldc.i4 12
add
stloc.1
```

```
$I2 = add 30, 12
local1 = $I2
```

Evaluation

It passed all the regression tests,
Such beautiful code it made.
Class libraries were thrown at it,
And class upon class it slayed.

What Can Be Translated?

- 197 out of 213 instructions (over 92%)
- Local variables, arithmetic and logical operations, comparison and branching instructions
- Calling methods, parameter passing
- Arrays
- Managed pointers
- Exceptions (try, catch, finally blocks)

What Can Be Translated?

- Object Oriented Features
  - Classes, abstract classes and interfaces
  - Inheritance
  - Static/instance fields and methods
  - Instantiation, constructors
- And various other odds and ends!
- Regression tests for each of these.

A More Realistic Test

- Supply libraries from the Mono implementation of the .Net class library to the translator
- See how many classes it can translate from each of the libraries
- Results: 4548 out of 5881 classes were translated (about 77%) ☺
- (Not accounting for dependencies ☹)

A More Realistic Test

- What stops us translating 100% of the .Net class library?

Reason	Count	Percentage
Unimplemented instruction	710	53%
Unimplemented built-in method	260	20%
Unimplemented construct	193	14%
Translator fault	171	13%

- A big missing feature is reflection.
- Also need to hand-code 100s of methods built into the .Net VM – a long job.

Comparing Stack To Register Mappers

- The Optimising Register SRM gave the best performing output in a Mandelbrot benchmark…

SRM	t_1	t_2	t_3	t_4	t_5	$t_{average}$
Stack	315.4	316.1	316.6	316.4	315.2	315.9
Register	21.30	21.25	21.31	21.28	21.28	21.28
OptRegister	12.02	12.03	12.00	12.02	12.02	12.02

- Emulating the stack is a serious slow down!

Comparing Stack To Register Mappers

- More surprisingly, the Optimising Register SRM also gave the best translation times for the .Net class library.

SRM	t_1	t_2	t_3	t_4	t_5	$t_{average}$
Stack	267.5	267.4	267.1	267.3	267.1	267.3
Register	228.9	229.4	229.9	228.8	228.6	229.1
OptRegister	220.0	220.0	219.9	219.8	220.0	219.9

- Result is due to compilation of generated PIR to Parrot bytecode dominating the translation time!

Conclusions

Love virtual machines does he,

Shared libraries make his day.

And libraries for VM B,

Now work on VM A.

Bytecode Translation Works!

- As originally predicted, it’s a lot of effort to get a working translator
- However, generated code can be pretty good
- Got most of the instructions and constructs being translated
- Able to translate a lot of the class library; hand-coded bits a sticking point

The Future

- Hoping to get the translator usable for production, but about the same amount of work required again to do so.
- Come and join the fun – lots of low hanging fruit still.
- Code in the Parrot repository, along with a To Do list.
- Or drop me an email, or I'm on #parrot

Any questions?