Bytecode Translation: From The .Net CLR To Parrot

Jonathan Worthington

OSCON 2007

The Problem

Love virtual machines did he,

Shared libraries made his day.

But libraries for VM B,

Wouldn't work on VM A.

Virtual Machines Are Good

- Abstract away the operating system and hardware, easing development and deployment
- Provide higher level constructs than real hardware, so easier to compile to
- Help to enable inter-operability between languages
- Safety and security benefits

Shared Libraries Are Good

- More generally, code re-use in general is good
- For libraries compiled to native (machine) code, calling into them is relatively easy…
  - Define a common calling conventions to pass the arguments
  - Do a jump instruction to library code

The Problem

- What about libraries written in languages that run on 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

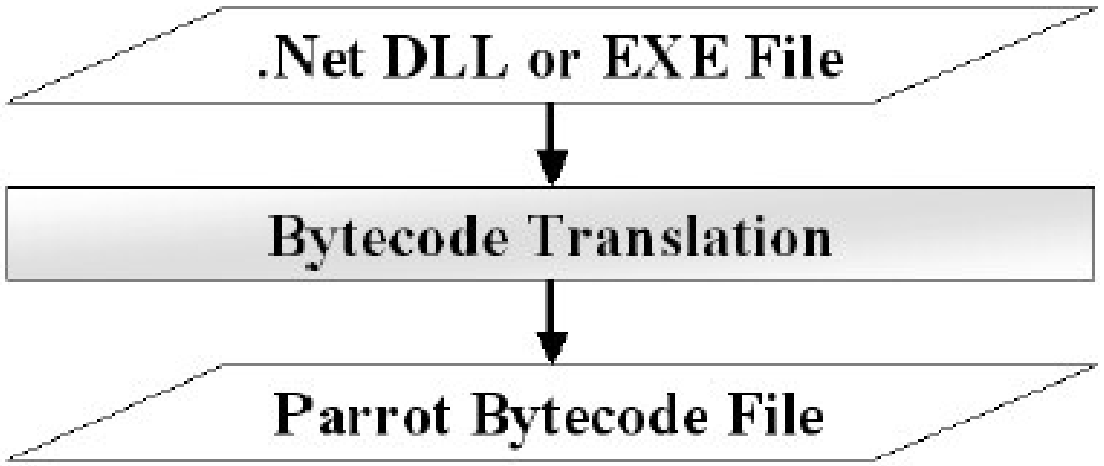
- Embed one VM inside another.
  - ✓ A quick way to something that basically works.
  - ✓ No issues matching semantics.
  - ✗ Making calls into the other VM transparent means duplicating state.
  - ✗ Have memory footprint of both VMs
  - ✗ Performance issues over boundary

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 appears to make the best trade-offs, so I chose to investigate that approach in detail.
- Chose to translate .Net bytecode to run on the Parrot VM.



Architecture

So a translator he conceived;

Designed so it would be,

Declarative and pluggable,

To manage complexity.

The .Net Common Language Runtime

- Stack based
- Polymorphic instructions
- Designed to be JIT-compiled
- An open standard
- A range of HLL features: arrays, value types, classes, fields, methods, single inheritance, interfaces, exceptions, more…

Translating .Net Libraries To Parrot

Parrot

- Aimed primarily at dynamic languages
- Register based, four types of register
- Instructions non-polymorphic
- Designed to be fast to interpret as well as JIT
- One in-progress implementation
- Aims to support wide range of HLL behaviour but retain interoperability

Translating .Net Libraries To Parrot

Issues with the translation

- Parrot is a register machine, while .Net is a stack machine.
- .Net stores in declarative metadata a lot of what Parrot does procedurally
  - Set of tables listing classes, fields, methods, signatures, etc.
- Some .Net instructions/constructs have no direct Parrot equivalent.

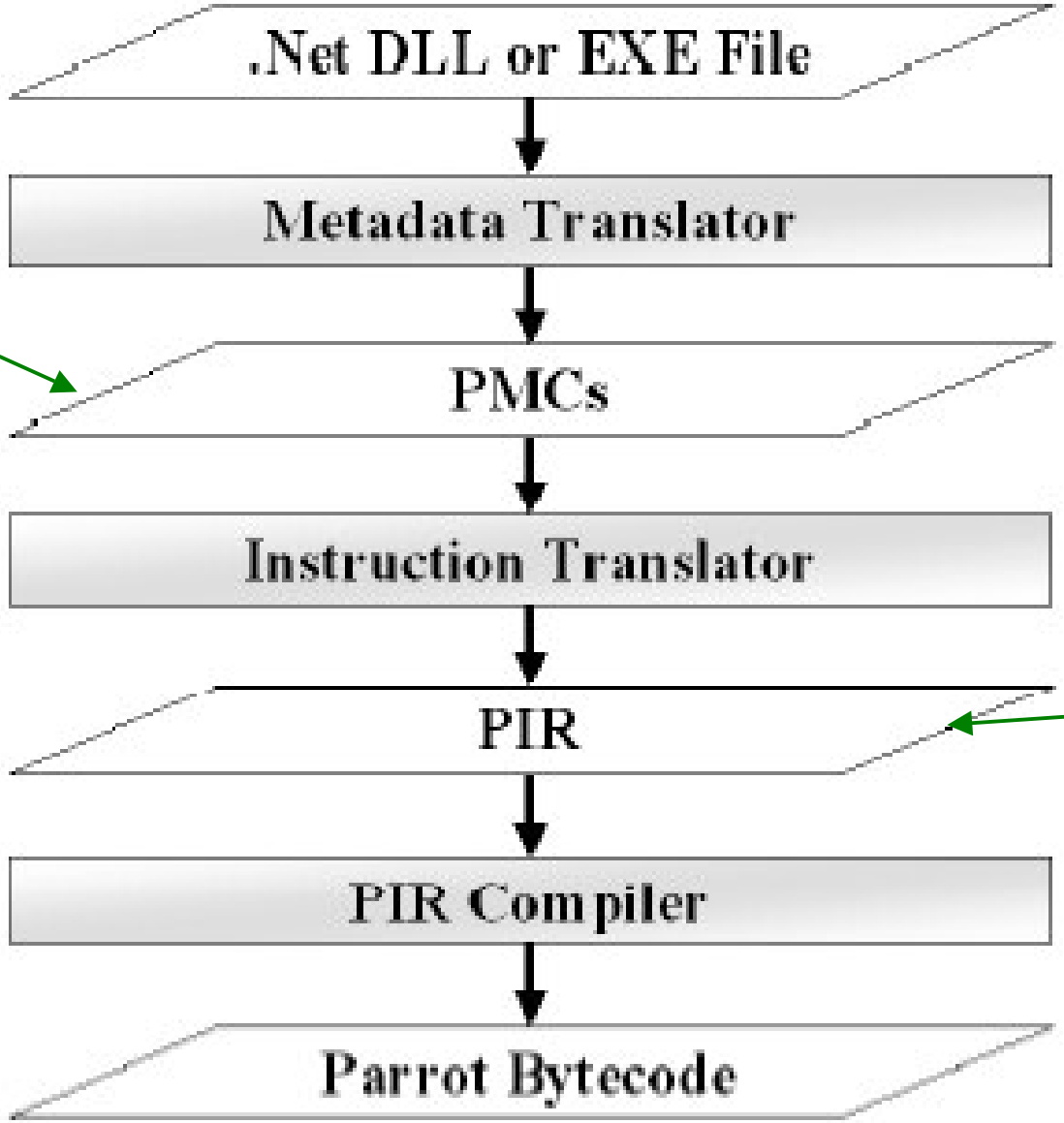
Some side-issues

- 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.

Translating .Net Libraries To Parrot

Architecture

A PMC is a C-based data structure accessible to VM-level code



Parrot's Intermediate Language; gets compiled down to bytecode

The Metadata Translator

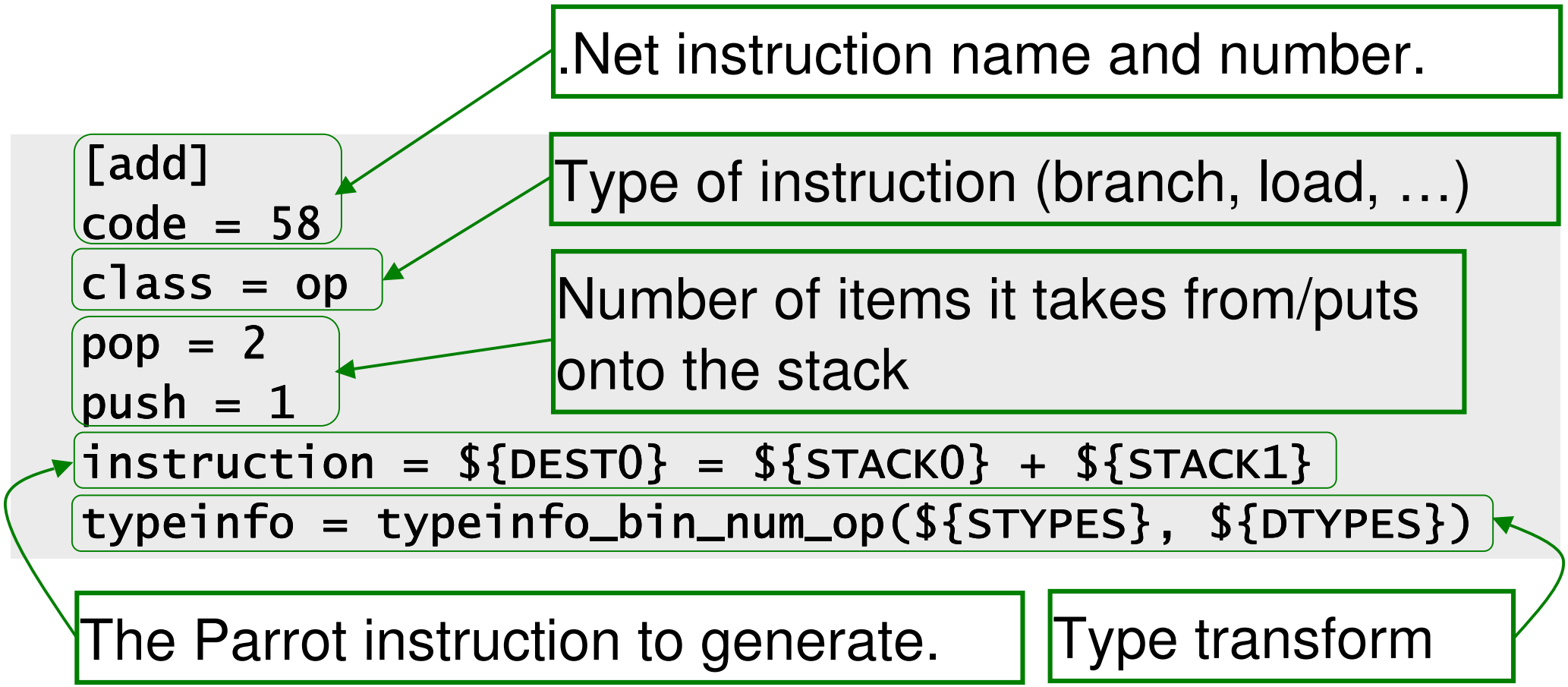
- Partly written in C (reading the .Net assembly), partly in PIR (code generation).
- C->PIR interface through PMCs (Parrot types implemented in C).
- Can generate class and method stubs with the metadata translator; instruction translator fills in the method bodies with the translated code.

The Instruction Translator

- Over 200 .Net instructions
- Much is common in translating instructions => don't want to maintain duplicate code or make the same mistakes again and again
- Stack to register mapping algorithm somewhat cross-cuts the process, and we want to be able to drop alternatives in

Declarative Instruction Translation

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



Pluggable Stack To Register Mapping

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

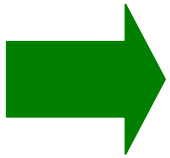
```
ldc.i4 30
ldc.i4 12
add
stloc.1
```

```
add local1, 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.

Generating The Instruction Translator

- A Perl script takes…
  - The declarative instruction translations file
  - A stack to register mapping module (written in Perl, generates PIR)
- Produces a PIR source file implementing the instruction translator.
- Generating code that generates code!

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!

Start With The Metadata Translator

- 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.

Building Translation Architecture

- The next step was to implement the translator and a trivial stack to register mapping algorithm.
- Initial instructions were implemented quickly and easily…
  - Arithmetic and logical operations, load and store of local variables and parameters, branch instructions

Load And Stores

- Metadata translator declares locals to have the names "local0", "local1", …
- Use ${LOADREG} to indicate the value is already in a register => hint to SRM

```
[ldloc.0]
code = 06
class = load
pop = 0
push = 1
pir = ${LOADREG} = "local0"
typeinfo = ${LOADTYPE} = ${LTYPES}[0]
```

Branches

- Emit a label before each translated instruction of the form LABn, where n is the position in the bytecode.

```
[br]
code = 38
class = branch
arguments = int32
pir = <<PIR
${ITEMP0} = ${NEXTPC} + ${ARG0}
${STEMP0} = ${ITEMP0}
${INS} = concat "goto LAB"
${INS} = concat ${STEMP0}
${INS} = concat "\n"
PIR
```

Implementing More Complex Features

- Will look today at…
  - Checked arithmetic
  - Managed pointers
  - Better stack to register mapping
- Other things in the dissertation on my site (http://www.jnthn.net/).

Checked Arithmetic

- Does arithmetic and throws an exception in the event of an overflow.
- Parrot does not have any instructions to do this.
- However, it supports dynamic instruction libraries…
  - Can extend the instruction set dynamically by language.

Checked Arithmetic

- Write the opcode in a .ops file; Parrot build tools do the rest.

```
inline op net_add_ovf(out INT, in INT, in INT) :base_core {
    if (CHECK_ADD_OVERFLOW($2, $3))
    {
        opcode_t *ret = expr NEXT();
        opcode_t *dest = dotnet_OverflowException(interp, ret);
        goto ADDRESS(dest);
    }
    else
    {
        $1 = $2 + $3;
        goto NEXT();
    }
}
```

Checked Arithmetic

- Need to emit code to load the dynamic instruction library
- Then the translation rule just uses the dynamic instruction

```
[add.ovf]
code = D6
class = op
pop = 2
push = 1
instruction = net_add_ovf ${DEST0}, ${STACK0}, ${STACK1}
typeinfo = typeinfo_bin_num_op(${STYPES}, ${DTYPES})
```

Managed Pointers

- Allow you to take a pointer into the stack, to an element of an array or a field of an object
- Can modify the data through the pointer.
- In Parrot terms, corresponds to taking pointers to registers and into PMC data – both are unsupported and dangerous to the VM's memory safety!

Managed Pointers

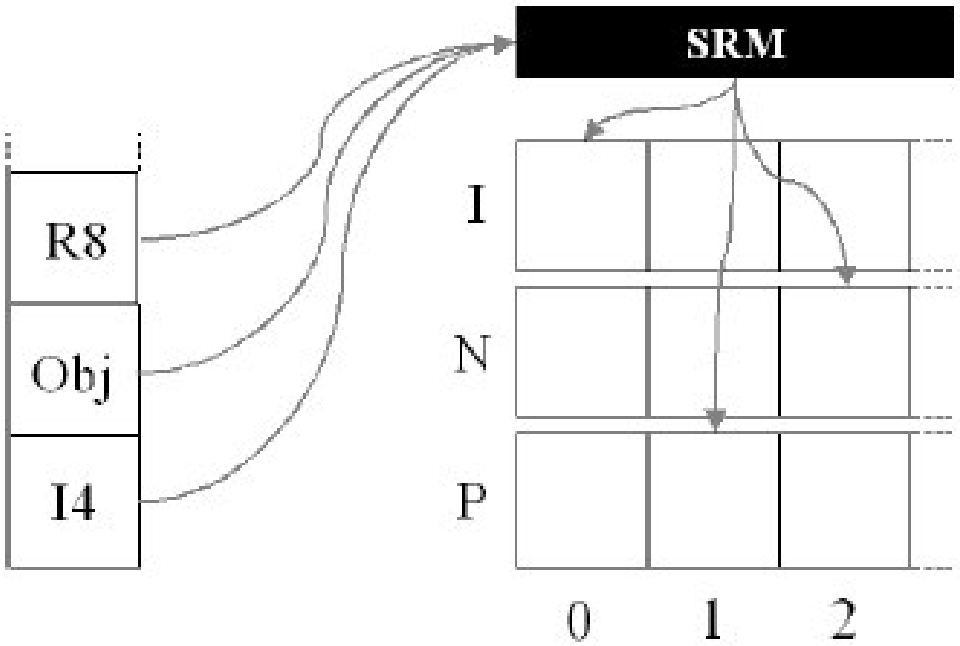
- Implemented a custom PMC to represent a managed pointer.
  - For pointers into arrays and object fields, stored reference to array or object PMC and the array index or field name => encapsulation not broken; will work with any array.
  - Tell garbage collector about array or object we're referencing => safe.

Managed Pointers

- Managed pointers to registers – store a pointer to a Parrot context (holding register frame) and the register type and number.
- Safety problem – what if register frame goes away?
  - Needed to add call-back mechanism to Parrot so pointer could be invalidated when register frame was.

A More Advanced SRM

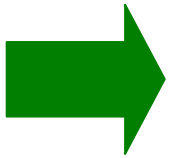
- Wanted to generate better register machine code.
- A better way: map each stack location to a register.



A More Advanced SRM

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

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

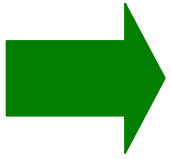


```
$I0 = 30
$I1 = 12
$I0 = add $I0, $I1
local1 = $I0
```

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
```



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

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 with an array is a serious slow down (due to lots of vtable method calls)

Comparing Stack To Register Mappers

- Perhaps 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!

Comparison with a .Net VM

- Mandelbrot again, so not real world code => don't read much into this.

VM	t_1	t_2	t_3	t_4	t_5	$t_{average}$
Mono	2.172	2.140	2.141	2.125	2.156	2.147
Parrot	12.02	12.03	12.00	12.02	12.02	12.02

- Note the .Net VM has to load the entire .Net class library => Parrot not doing so yet, so unrealistic start-up time.
- Parrot disadvantaged - .Net is using JIT but Parrot JIT crashed on the code!

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

Code Less, But Smarter

- Generating the translator was a good thing!
  - Input: 3000 line instruction translations file, few hundred lines per SRM mapper, 1000-ish line script to generate the translator.
  - Output: up to 22,000 lines of PIR that you'd really not want to maintain by hand – and it runs fast!

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: jnthn@jnthn.net

Any questions?