Reflections on a decade of MoarVM

A runtime for the Raku programming language

Jonathan Worthington
Edument

MoarVM is...

A virtual machine

Built for the Raku programming language (née Perl 6)

Developed by an open source community

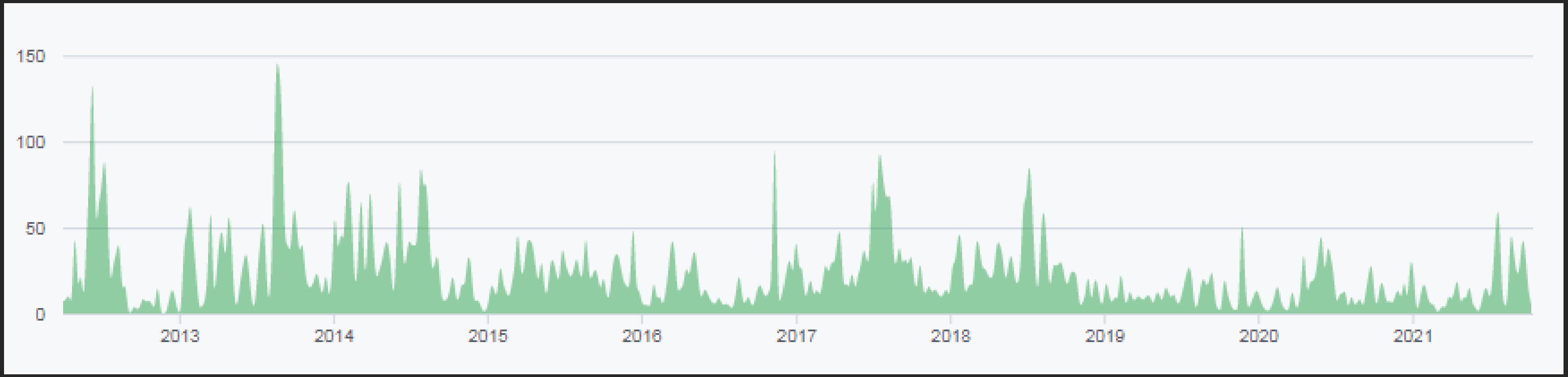
I am...

Co-founder and architect of MoarVM
Architect of the Rakudo compiler
Raku MOP & concurrency co-designer

Working at Edument, primarily on developer tooling projects
(previously was mostly teaching)

Time flies

Nearly 10 years of development!



I've learned a lot about VMs.
Still got a lot left to learn.

The origins

of the MoarVM project

The early days of MoarVM

as a simple bytecode interpreter

How MoarVM advanced

to incorporate many of the VM "tricks of the trade"

The growing pains

that we experienced as MoarVM advanced

A new generalized dispatch mechanism

that's enabling us to do more with less

The origins

of the MoarVM project

How I got involved

Ran a small web development company in my teens, used Perl a lot

At university, really enjoyed the courses on compilers and languages

Wanted to explore that area and give something back to the Perl community

Perl 6

Yes, I've already heard *all* the jokes

Yes, it was eventually released

Diverged from Perl 5 in <u>many</u> ways

Perl 6

Yes, I've already heard *all* the jokes

Yes, it was eventually released

Diverged from Perl 5 in many ways

It darn well needed to!

Perl 6

Yes, I've already heard *all* the jokes

Yes, it was eventually released

Diverged from Perl 5 in many ways

It darn well needed to!

but

Is this thing even Perl?

Raku

Eventually renamed to Raku

I'll refer to the language as Raku throughout this talk

What makes Raku interesting* to implement?

* As in "may you live in interesting times"

Dynamic language, but...

There are types, and they *must* be enforced runtime at latest

```
my class IPGNode {
    has Function $.function is required;
    has ValueStateGraph::LambdaNode $.lambda is rw;
    has IPGNode @.calls;
    method add-callee(IPGNode $node --> Nil) {
        @!calls.push($node);
    }
}
```

Naive implementation?

Loads of runtime spent doing type checks!

Operators are multis

Multiple dispatch very widely used, including for nearly ever operator

```
say $m * $x + $c;
```

=

```
say infix:<+>(infix:<*>($m, $x), $c);
```

Not much ad-hoc polymorphism...
...but demands that multiple dispatch is fast!

Arbitrary precision

The `Int` type is arbitrary precision (also native `int` which is not)

4.2 is a `Rat` (rational number), not floating point

*time at *time

EVAL (compile time at runtime)

but also

BEGIN (runtime at compile time)

Meta-programming

Meta-classes not just for introspection

Called by the compiler to construct types, and at runtime to find methods, do type checks, etc.

Can subclass built-in metaclasses or define completely new ones

Grammars

Raku has a new "regex" syntax...that scales up to decidedly irregular things

```
grammar JSON::Tiny::Grammar {
    token TOP       { \s* <value> \s*          }
    rule object     { '{' ~ '}' <pairlist>     }
    rule pairlist   { <pair> * % \,            }
    rule pair       { <string> ':' <value>     }
    rule array      { '[' ~ ']' <arraylist>    }
    rule arraylist  {  <value> * % [ \, ]      }
    ...
}
```

Raku eats itself

The Raku Language syntax is defined (and parsed) using...a Raku grammar!

Can mix into the grammar to tweak the language syntax

```
multi postfix:<!>(Int $n) {
    [*] 1..$n
}
```

Which means...

The compiler is just another Raku program running atop of the VM

The Raku standard library is written in Raku, with a means to call VM primitives

Sounds idyllic…

...but, well...

...it's challenging...

...to even start to compete with a "classic" dynamic language implementation (let alone a modern one) when you're writing your...

Basic operators
Object model
Compiler

...in something you're still trying to run fast!

But back to me

A younger, naiver, me had *no idea* about the challenges ahead

Started contributing to the Rakudo compiler

But was also curious about the Parrot virtual machine

Parrot

"One bytecode to rule them all"

Aimed to be a VM for all dynlangs

Parrot didn't make it, but the idea survived, and was (independently) later realized in GraalVM

Parrot frustrations
(only really clear to me in hindsight)

We're an independent project from Perl 6 and don't want to put all our eggs in that basket...

Parrot frustrations

(only really clear to me in hindsight)

We're an independent project from Perl 6 and don't want to put all our eggs in that basket...

We're Parrot's main customer, it doesn't even run our language well yet (slow, struggling with threading...)

What happened?

What happened?

Youthful arrogance happened!

"What if I implemented a VM focused entirely on the Raku language?"

The early days of MoarVM

as a simple bytecode interpreter

Why the name?

I'd previously been working on the Raku Meta-Object Protocol

We'd build a runtime to host that

"Metamodel On A Runtime VM"

Actually, uhhh….

We just liked silly memes

https://seekingalpha.com/article/4061225-moar-cheezburgers-plz

The rough plan

Start out as a simple interpreter

Try to make different mistakes to Parrot

Add the trickier things (type specialization, JIT, etc.) later

Raku Architecture

(Prior to MoarVM)

Rakudo

Compiler (NQP) | MOP (NQP) | Library (Raku)

NQP

Bootstrapped subset of Raku (thus written in NQP itself)

Parrot VM

Raku Architecture

(Prior to MoarVM)

Com

Write a bytecode generator for MoarVM, then get NQP to compile itself for MoarVM

NQP

Bootstrapped subset of Raku (thus written in NQP itself)

Parrot VM

A language for MoarVM

Needed to pick a systems language

C was the least imperfect choice

I knew it, but more importantly, so did many folks in the community around the language - more so than other options

The interpreter

Chose register-based over stack-based (in common with Parrot)

Computed `goto` where available, fallback to a giant `switch` statement

Why register-based?

No stack pointer to maintain

Registers have types (native int/num/str or object), so easier GC marking

SSA form would have a straightforward relationship with the original bytecode

Register VM downsides

Probably bigger bytecode

Invocation records have to zero out object registers to not confuse the GC, and this becomes rather costly
(and mitigating it gets back to the same complexity as having stack maps)

GC

2 generations

Nursery is per-thread semispace copying

Old generation shared and non-moving

Parallel (but not concurrent) GC

The GC has invariants

Must know the locations of all collectable objects (for copying)

Assignments into collectables require write barriers (for generational)

Need a lot of discipline to uphold them

Needs discipline

=

Will be done wrong

Needs discipline

=

Will be done wrong

Again and again and again and again and again and again and again and again and again and again and again and again and again and again and again and again and again and again and again

Coping with C and a GC

Run with tiny nursery (makes broken invariants far more likely to cause failures)

Compile with "not in fromspace" assertions on every register access and assignment

Allocate new fromspace every time, so bad reads will trigger ASAN/valgrind

GCC plugin doing static analysis

Object system

Type = Meta-object + Representation

Meta-object	Representation
Implemented in HLL	Implemented in the VM
Dispatch semantics	Memory layout
Type membership	Involved with GC
Introspection	Serialization/deserialization

Object system

Type = Meta-object + Representation

Meta-object

Implemented in HLL

Dispatch semantics

Type membership

Introspection

> With the exception of one provided by the VM to "bootstrap" the rest (supports fields and methods, but no subtyping)

Involved with GC

Serialization/deserialization

Object system

Type = Meta-object + Representation

M

Dynamic array
Fixed array
Record type
CStruct (FFI)
Continuation
...

Introspection

Representation

Implemented in the VM

Memory layout

Involved with GC

Serialization/deserialization

In a simple bytecode interpreter world....

Interpreting bytecode is slow
Making calls is slow
but
Things written in C are fast
therefore
Find ways to do hot path things in C

Loads of complex ops and APIs

Meta-objects could publish a flat method lookup table, used for quick lookups of methods

Loads of complex ops and APIs

A tree-based multi dispatch lookup cache (nominal types only) to speed up multiple dispatches

Loads of complex ops and APIs

Raku has first-class l-values, but assignment is hot, so the assignment process was written in C

C a la CPS

Many of these complex operations sometimes needed to call into bytecode

But nested runloops are bad
(they cause a continuation barrier)

Thus have to write them CPS-style

How MoarVM advanced

to incorporate many of the VM "tricks of the trade"

Scarce resources

Early bet: type specialization, inlining, etc. would offer greater speedups than compilation to machine code

Compiled to machine code

!=

It'll run fast

Specializer ops

Interpreter opcodes that are disallowed in input bytecode, but may be produced internally

Can do things that are only safe because analyses proved them so

Getting started

Keep call counts of functions, and once a limit is reached, try to produce a specialization

Keyed on callsite shape (arity, named argument names) and the types of any object arguments

Analysis

Form CFG from bytecode

Turn it into SSA

Facts (known type, known value) kept per SSA variable

Optimizations

Delete arity checks
Delete proven type checks
Turn method lookups to constants
Dead branch elimination
Dead instruction elimination
Lower attribute access to pointer ops
Specialize some complex ops

Then crash and burn...

Raku has mixins

Types of objects can change at a distance

The type a specialization was keyed on could change → opts break stuff

Deoptimization

Every function call is a potential deoptimization point

Keep a table mapping optimized to unoptimized return addresses

On a mixin, walk stack and rewrite them to point to unoptimized code

Deoptimization

But this changes everything!

Every functi… …ial deop…

Keep a table mapping optimized to unoptimized return addresses

On a mixin, walk stack and rewrite them to point to unoptimized code

Deoptimization

Every function c… ial deoptim…

Can do it for any program point

Keep a table mapping optimized to unoptimized return addresses

On a mixin, walk stack and rewrite them to point to unoptimized code

Statistics will do

Add logging of types of...

Non-local variable lookups

Attribute lookups

Return values of function calls

If a stable type is observed most of the time, insert a guard

➔ can assume that type beyond it

Specialization linking

Applies when we call one specialized function from another

May already "know" that we have the input types for a specialization

Directly call the specialization
➔ eliminates some guards

Inlining

For small callees where we know the specialization, can inline it

MoarVM does multi-level inlining
(Related headache: this means for deopt we need multi-level uninlining too!)

More chances to eliminate guards

OSR

Some programs have long-running hot loops (micro-benchmarks!)

I cheated: compiler emits `osrpoint` ops at the end of a loop that trigger the production of a specialization

Really just deopt in reverse

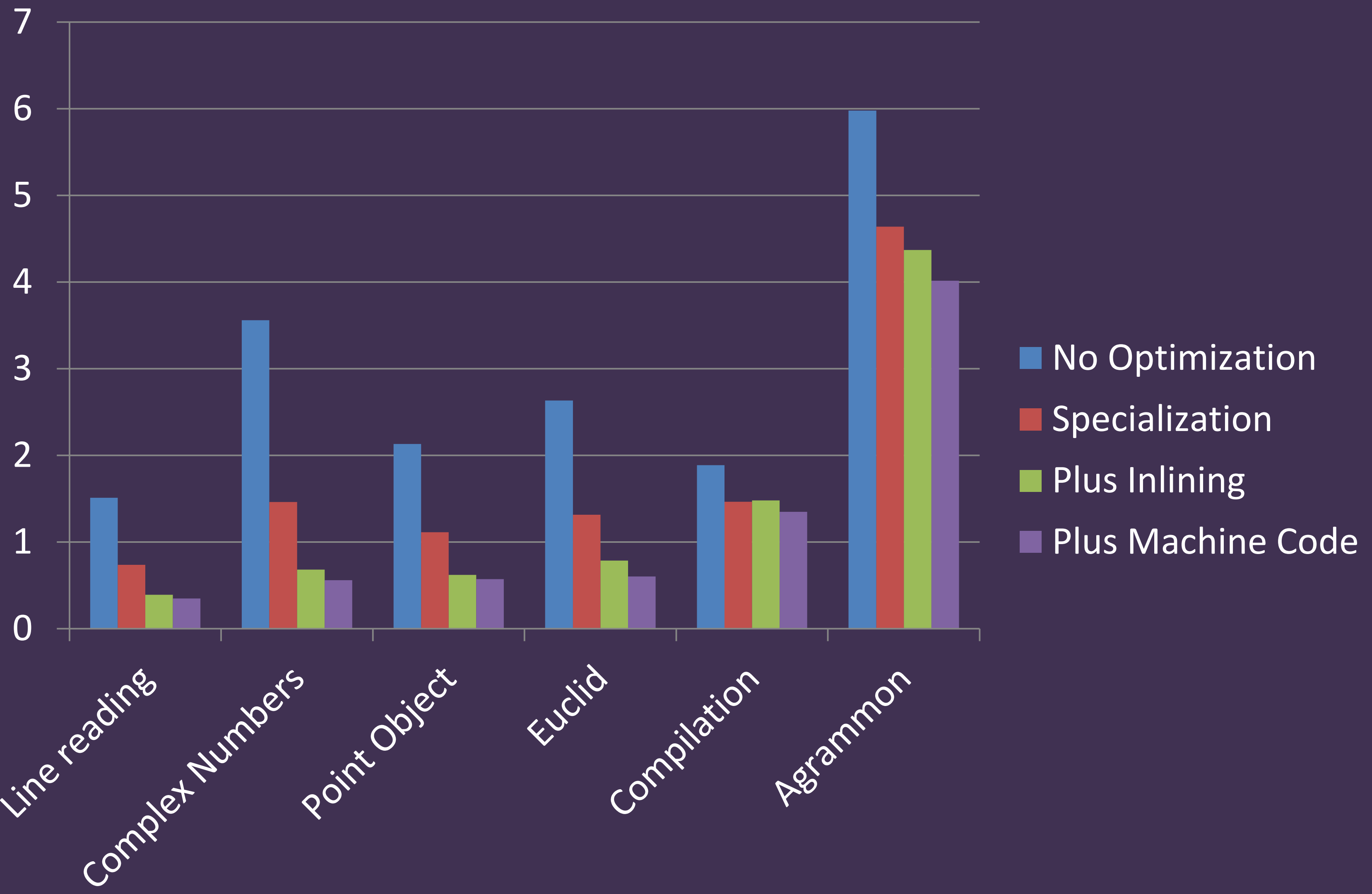
Machine code

Somebody did eventually implement compilation to machine code (x64)

A significant win for tight math involving native types

Smaller win elsewhere
➔ seems to validate our strategy

What helps most?



The growing pains

that we experienced as MoarVM advanced

Complexity? Bugs!

All too easy to be fast and wrong

When optimization or deoptimization bugs happen, need ways to debug them

Also want to be proactive (find them before language users do)

Triggering bugs

A special NODELAY mode, which optimizes all code, not just hot code

Exercises the optimizer a lot

Also, bad type statistics mean terrible optimization choices, so it exercises deoptimization a lot too!

Hunting bugs

Lots of logging

Dump SSA before and after optimization
Analyses/transforms can add comments
Dump deoptimizations

Specialization bisection

Environment variable to limit number of specializations produced ➔ can quickly find which specialization breaks the program

Optimization takes time

Initially, interrupted interpreting code to produce specializations

Poor use of multi-core hardware

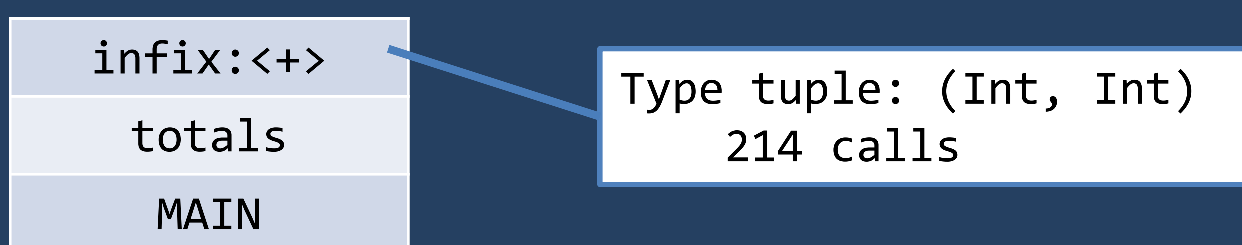
Also fun: data parallel code tended to have every thread trying to produce the same set of specializations

Specializer thread

Interpreter threads running unspecialized code log calls, returns, and types into a buffer...

...and, once it's full, send it to the specializer thread.

The specializer thread replays these, simulating the stack, and builds up statistics, which are used to plan specializations



The Good

Very much a measurable improvement

Puts another core to work

The Bad

New source of non-determinism

BLOCKING mode to recover bisection

The ugly

Some programs exhibit significant performance differences from run to run

[Poly|mega]morphism

Improvements in micro-benchmarks don't map directly to real programs

Initially, set an upper limit on number of specializations, to cope with the "rare megamorphic cases"

Turns out they ain't so rare...

Stable type

```
class Array {
    multi method ASSIGN-POS(Int $index, Any $value) {
        ...
    }
    ...
}
```

Dozens of different types in any non-tiny program

Mono, poly, mega

Observed type specialization
(From an exact observed type tuple)

Derived type specialization
(Only the stable types in the tuple)

Certain specialization
(From an observed callsite but any types)

Broken assumptions

Remember this slide?

In a simple bytecode interpreter world...

Interpreting bytecode is slow
Making calls is slow
but
Things written in C are fast
therefore
Find ways to do hot path things in C

Broken assumptions

Remember this slide?

In ~~a simple bytecode interpreter~~ our new world...

Interpreting bytecode is slow
Making calls is slow
but
Things written in C are fast
therefore
Find ways to do hot path things in C

Broken assumptions

Remember this slide?

In ~~a simple bytecode interpreter~~ our new world....

~~Interpreting bytecode is slow~~ We compile to machine code

Making calls is slow

but

Things written in C are fast

therefore

Find ways to do hot path things in C

Broken assumptions

Remember this slide?

In ~~a simple bytecode interpreter~~ our new world....

~~Interpreting bytecode is slow~~ We compile to machine code
~~Making calls is slow~~ And perform inlining
but
Things written in C are fast
therefore
Find ways to do hot path things in C

Broken assumptions

Remember this slide?

In ~~a simple bytecode interpreter~~ our new world....

~~Interpreting bytecode is slow~~ We compile to machine code
~~Making calls is slow~~ And perform inlining
but
Things written in C are ~~fast~~ opaque to the optimizer
therefore
Find ways to do hot path things in C

Broken assumptions

Remember this slide?

In ~~a simple bytecode interpreter~~ our new world....

~~Interpreting bytecode is slow~~ We compile to machine code
~~Making calls is slow~~ And perform inlining
but
Things written in C are ~~fast~~ opaque to the optimizer
therefore
~~Find ways to do~~ Stop doing hot path things in C

Example: assignment

Type checks done from C?

Can't eliminate them ☹

Assignment triggers a call?

Can't specialization link or inline it ☹

Write into container done in C?

Escape analyzer can't see it ☹

~~Speedup~~ Speed hump

Complex operations to avoid interpreter and call overhead are either...

Opaque to the optimizer

(And so opportunities to optimize are lost)

Abstractly interpreted in the optimizer

(Causing duplication, complexity, and thus bugs)

Performance cliffs

Too many language semantics to bake special cases for them all into the VM

Optimizer then tends to make the performance cliffs even higher

A new generalized dispatch mechanism

that's enabling us to do more with less

It's all about dispatch

If something is a dispatch...

It's all about dispatch

If something is a dispatch...

Any operation where the
code we decide to run is
determined by the types
or values of the arguments

It's all about dispatch

If something is a dispatch...

...and the VM doesn't know it's one...

...it's going to be slow...

...and the optimizer won't help much

A solved problem?

Take the types or values of a set of arguments, transform the arguments, invoke some code with them...

...sounds very much like the JVM's `invokedynamic`?

Take it all the way

The VM *and* the compilers targeting it are under our control

So we didn't just *add* a new dispatch mechanism to MoarVM

We were also able to *remove* almost a dozen ad-hoc dispatch-y things

One opcode

```
result = dispatch 'name', callsite, ...
```

One opcode

```
result = dispatch 'name', callsite, ...
```

The name of a dispatcher,
looked up in a registry

One opcode

```
result = dispatch 'name', callsite, ...
```

The argument shape (count of positional arguments, names of named arguments)

One opcode

```
result = dispatch 'name', callsite, ...
```

The registers holding values
for each of the arguments

Dispatch terminals

Every dispatch bottoms out in one of:

boot-constant (a literal value)
boot-value (a read argument, read field, etc.)
boot-code-constant (constant bytecode handle)
boot-code (a read bytecode handle)
boot-syscall (a VM-provided primitive)
boot-foreign-code (a call using the FFI)

Dispatch terminals

Every dispatch bottoms out in one of:

boot

boot-v

boot-code … de handle)

boot-code (a … ad bytecode handle)

boot-syscall (a VM-provided primitive)

boot-foreign-code (a call using the FFI)

`dispatcher-register` call to register a userspace-defined dispatcher

Userspace dispatchers

Invoked with an argument capture
(The Raku term for an argument tuple, except it can have positional and named arguments)

Can add guards and transform capture

Must finish by delegating to another dispatcher (user-defined or terminal)

Userspace dispatchers

Invoked v… re
(The Raku t… pt it
can hav…)

raku-meth-call →
raku-meth-call-resolved →
raku-multi →
raku-multi-core →
raku-invoke →
boot-code-constant

Can add guar… form capture

Must finish by delegating to another dispatcher (user-defined or terminal)

Dispatch program

Set of ops derived from the guards, capture transformations, and terminal

Delegations and captures are erased, guards are de-duplicated

Program installed at the callsite

(Polymorphic sites may have many programs)

Optimization

Specializer translates dispatch programs in hot code into specializer ops

No guard if property already proven
(inserted ones may later be dropped too)

Implementation is very regular
(no knowledge of method cache, multi cache, etc.)

Not quite enough

What's described so far is mostly a remix of ideas found elsewhere

However, in Raku, dispatch can be a process over time...

```
class Operator {
    method emit($left, $right) {
        ...
    }
}
class Comparison is Operator {
    has Bool $.negated;
    method emit($left, $right) {
        $!negated
            ?? self.negated(callsame())
            !! callsame()
    }
}
```

```
class Operator {
    method emit($left, $right) {
        ...
    }
}
class Co
    has Bool
    method emit($left, $right)
        $!negated
            ?? self.negated(callsame())
            !! callsame()
    }
}
```

Call the next candidate

Next wrapper, or next most general multi, or next method in the MRO

Call the next candidate

```
class Operato
    method emit($left, $right)
        ...
    }
}
class Co
    has Bool
    method emit($left, $right)
        $!negated
            ?? self.negated(callsame())
            !! callsame()
    }
}
```

```
multi fac(Int $n where $n <= 1) {
    1
}
multi fac(Int $n) {
    $n * fac($n - 1)
}
```

Invoke this candidate, if its where clauses fail, try calling the next most general one

```
multi fac(Int $n where $n <= 1) {
    1
}
multi fac(Int $n) {
    $n * fac($n - 1)
}
```

The problem

Dispatches may need to be continued (and we don't always know up front)

We'll need to recover the original args

Where we go next may be late-bound, and `lastcall`/`nextcallee` even make the whole thing stateful!

Resumable dispatchers

A user-space dispatcher can:

Provide a resume callback

For if the dispatch it started should be resumed

Specify resume initialization arguments

Derived from the initial capture; carries either no or very low dispatch-time cost

The resume callback

Can do all the dispatch callback can, and:

Recover the resume init args

As a capture, although it's erased in the dispatch program

Read/write one object reference of state

Most often used to hold a linked list of candidates, for example, of all matching methods in the MRO

Record phase

```
class GP {
    method m($x) { 'gp-' ~ $x }
}

class P is GP {
    method m($x) { 'p-' ~ callsame() }
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

Record phase

```
class GP {
    method m($x) { 'gp-' ~ $x }
}

class P is GP {
    method m($x) { 'p-' ~ callsame() }
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

Method call dispatcher indicates (C, 'm', 42) are the resume init args

Record phase

```
class GP {
    method m($x) { 'gp-
}

class P is GP {
    method m($x) { '
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

Triggers resume callback, that:

1. Obtains resume init args
2. Guards on them
3. Builds linked list of MRO
4. Stores 3rd node onward as dispatch state
5. Invokes 2nd node

Rec

Triggers resume callback, that:

1. Obtains dispatch state, D
2. Guards on D.meth
3. Updates dispatch state to D.next
4. Obtains resume init args
5. Invokes D.meth with the args

```
class GP {
    method m($x) {
}

class P is GP {
    method m($x) { 'p-' ~ callsame() }
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

Run phase

```
class GP {
    method m($x) { 'gp-' ~ $x }
}

class P is GP {
    method m($x) { 'p-' ~ callsame() }
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

Guards on invocant type, resume init args are just data, so no cost

Run phase

```
class GP {
    method m($x) { 'gp-
}

class P is GP {
    method m($x) { '
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

1. Guards that the next resumption is the expected one
2. Guards on callsite shape
3. Reads init args, guard on invocant type and method name
4. Runs the target method

Re

1. Guards that the next resumption is the expected one
2. Guards on callsite shape
3. Guards on dispatch state
4. Obtains init arguments
5. Runs the target method

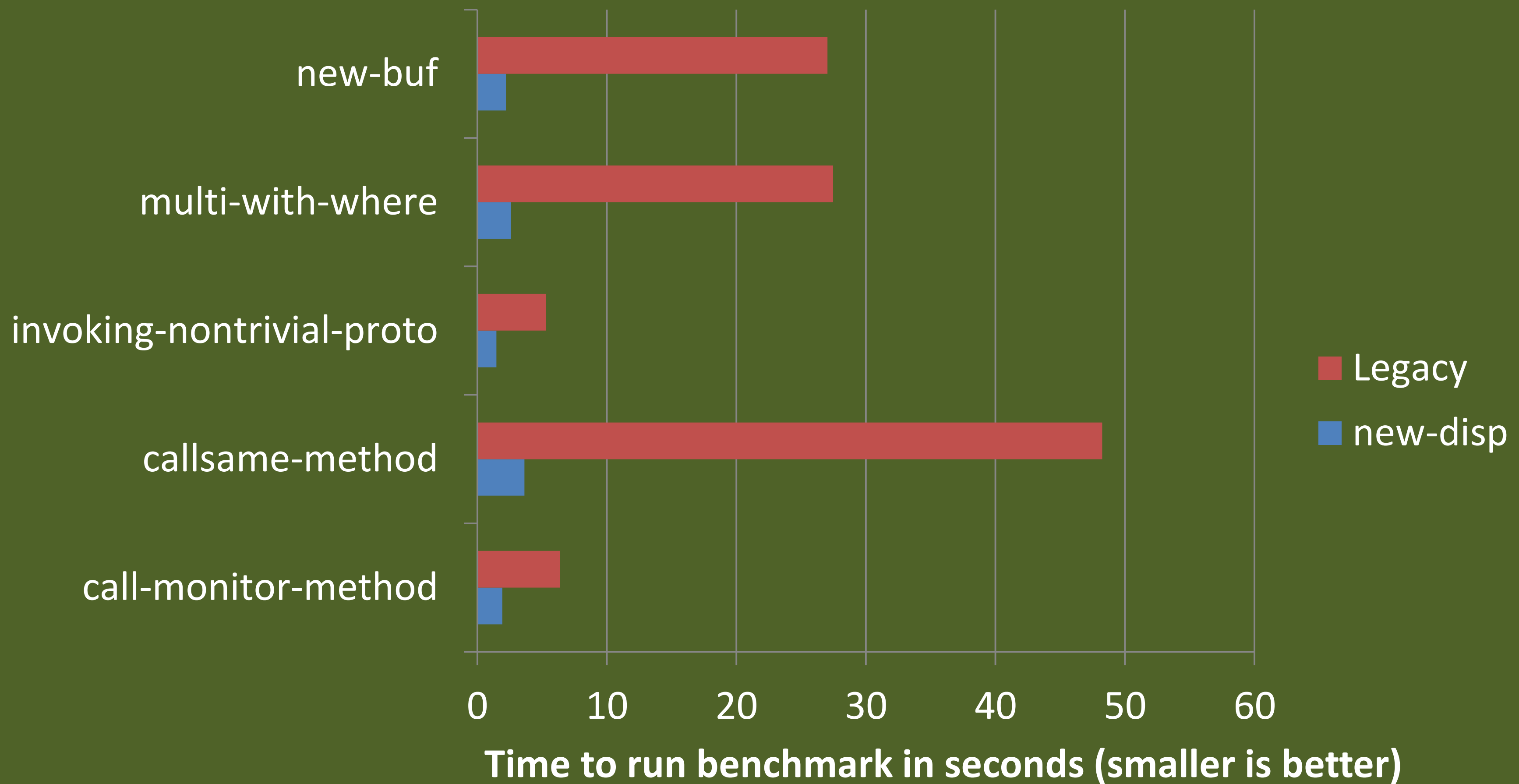
```
class GP {
    method m($x) {
}

class P is GP {
    method m($x) { 'p-' ~ callsame() }
}

class C is P {
    method m($x) { 'c-' ~ callsame() }
}

say C.m(42); # c-p-gp-42
```

A big improvement



No silver bullet

Moved from caching at "destinations" to caching at the callsite

Better for the monomorphic majority, let us deal with resumable dispatches

<u>Much</u> worse for megamorphic sites

No silver bullet

Moved from caching at "destinations" to caching at the callsite

Better for the monomorphic majority, let us d… atches

Can easily do 20x worse!

Much worse for megamorphic sites

Doing better (a WIP)

Expose callsite size to dispatchers

Once it reaches a certain size, switch strategy (for example, method hash)

Try latest dispatch program first (to immediately hit megamorphic strategy)

Also: longer warm-up

Setup work at every callsite now

Dispatchers are themselves optimized and JIT-compiled - but only after they have warmed up too

Can we somehow safely cache the work or "prime" the callsites at compile time?

In closing…

MoarVM has been more about trying to apply existing ideas than creating new approaches to VM design

(Although the resumable dispatch handling is something I didn't see elsewhere yet.)

Demonstrated that one can transition a "traditional" dynlang implementation to a modern one in an incremental manner?

Maybe. But many caveats.

(Clean compiler/VM separation. FFI rather than C extension API. Small user base when we started.)

Thank you!

Questions?

@ jonathan@edument.cz

W jnthn.net

jnthnwrthngtn

jnthn