A Tale of Two Projects

It is the best of jitting, it is the worst of jitting…

Collaborators

- Jan Vitek
- Oli Fluckiger
- Jan Jecmen
- Paley Li
- Roman Tsegelskyi
- Alena Sochurkova
- Petr Maj

Design Goals

- Performance
  - The JIT should outperform both AST and BC interpreter
- Compatibility
  - Full R language must be supported
  - At least in theory, in practice we are happy with BC interpreter compatibility
- Easy Maintenance
  - Source code should be easy to understand and simple to maintain
  - Counterexample: LuaJIT

The Importance of having a JIT

- Costs of BC Interpreter
  - Hard to predict indirect jump for each instruction in program
  - Operands stack vs registers

- JIT mitigates these
  - Zero cost of moving to next instructions
  - Uses platform registers directly
  - Better optimization for low-level parts

Low Level Virtual Machine (LLVM)

- Backend for clang compiler
  - Used by many other languages
- State of the art compiler suite
  - Hundreds of optimizations (including some vectorization)
  - Dozens of targets
- Designed as AOT compiler
  - Slow compilation time
  - Fast & Optimized output
- But provides a JIT layer

McJIT – LLVM JIT Layer

- Developed by Laurie Hendren at McGill
  - used for Matlab
- Program must be translated to LLVM IR
- McJIT then turns LLVM functions into pointers to native functions
  - Handles the dynamic loading and native code generation
- Newer LLVM versions uses ORC JIT instead
  - Layered approach, true JIT

LLVM IR

- Everything is Typed
  - Values, functions, registers, instructions
- Very low-level
  - Assembly-like nature
- Registers based VM
  - Unlimited number of registers
  - Single Static Assignment

RJIT

The pros & cons of using LLVM as backend for R

Getting a JIT Quickly

- Translating R semantics directly to LLVM IR too complicated

- Main idea:
  - Convert R bytecode instructions into functions and call them from within the JIT

```
> x = 2 + 3
```

A simple expression in R’s REPL

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

R Bytecode

```
OP(LDCONST, 1):
  R_Visible = TRUE;
  value = VECTOR_ELT(constants, GETOP());
  MARK_NOT_MUTABLE(value);
  BCNPUSH(value);
  NEXT();


OP(ADD, 1):
  FastBinary(R_ADD, PLUSOP, R_AddSym);
  NEXT();


OP(SETVAR, 1):
  int sidx = GETOP();
  SEXP loc;
  SEXP symbol = VECTOR_ELT(constants, sidx);
  loc = GET_BINDING_CELL_CACHE(symbol, rho, vcache, sidx);
  ...
  value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```c
void instruction_LDCONST_OP(InterpreterContext * c, int arg1) {
  R_Visible = TRUE;
  c->value = VECTOR_ELT(c->constants, arg1);
  MARK_NOT_MUTABLE(c->value);
  BCNPUSH(c->value);
  NEXT();
}
```

```c
OP(ADD, 1):
  FastBinary(R_ADD, PLUSOP, R_AddSym);
  NEXT();


OP(SETVAR, 1):
  int sidx = GETOP();
  SEXP loc;
  SEXP symbol = VECTOR_ELT(constants, sidx);
  loc = GET_BINDING_CELL_CACHE(symbol, rho, vcache, sidx);
  ...
  value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
void instruction_LDCONST_OP(InterpreterContext * c, int arg1) {
  R_Visible = TRUE;
  c->value = VECTOR_ELT(c->constants, arg1);
  MARK_NOT_MUTABLE(c->value);
  BCNPUSH(c->value);
  NEXT();
}
```

```
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```
OP(SETVAR, 1):
  int sidx = GETOP();
  SEXP loc;
  SEXP symbol = VECTOR_ELT(constants, sidx);
  loc = GET_BINDING_CELL_CACHE(symbol, rho, vcache, sidx);
  ...
  value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
void instruction_LDCONST_OP(InterpreterContext * c, int arg1) {
  R_Visible = TRUE;
  c->value = VECTOR_ELT(c->constants, arg1);
  MARK_NOT_MUTABLE(c->value);
  BCNPUSH(c->value);
  NEXT();
}
```

```
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  SEXP symbol = VECTOR_ELT(c->constants, arg1);
  loc = GET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);
  ...
  SEXP value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
}
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```c
void instruction_LDCONST_OP(InterpreterContext * c, int arg1) {
  R_Visible = TRUE;
  c->value = VECTOR_ELT(c->constants, arg1);
  MARK_NOT_MUTABLE(c->value);
  BCNPUSH(c->value);
  NEXT();
}
```

```c
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```c
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  SEXP symbol = VECTOR_ELT(c->constants, arg1);
  loc = GET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);
  ...
  SEXP value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
}
```

```c
typedef struct {
  SEXP rho;
  Rboolean useCache;
  SEXP value;
  SEXP constants;
  R_bcstack_t * oldntop;
  R_binding_cache_t vcache;
  Rboolean smallcache;
} InterpreterContext;
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
call void LDCONST_OP(2)
call void LDCONST_OP(3)
call void ADD_OP()
call void SETVAR_OP()
```

LLVM IR

```
void instruction_LDCONST_OP(InterpreterContext * c, int arg1) {
  R_Visible = TRUE;
  c->value = VECTOR_ELT(c->constants, arg1);
  MARK_NOT_MUTABLE(c->value);
  BCNPUSH(c->value);
  NEXT();
}
```

```
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  mbol = VECTOR_ELT(c->constants, arg1);
  ET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);

  lue = GETSTACK(-1);
  NT_NAMED(value);
  DING_VALUE(loc, value))
  ...
  NEXT();
}
```

- So far the effort was minimal
  - Refactor BC insns into functions
  - Interpreter’s local variables go to the context
  - LLVM IR is just a sequence of calls
  - Constant pool is roughly the same
  - Control flow is a bit more involved

- So far the effort was minimal
  - Refactor BC insns into functions
  - Interpreter’s local variables go to the context
  - LLVM IR is just a sequence of calls
  - Constant pool is roughly the same
  - Control flow is a

```
if (a) {
    b;
} else {
    c;
}
```

```
 call void GETVAR_OP a
  %1 = call i1 ConvertToLogicalNoNA()
  br %1 true false
true:
  call void GETVAR_OP b
  br next
false:
  call void GETVAR_OP c
  br next
next:
  %3 = call SEXP bcPop()
  ret SEXP %3
```

Removing the Stack

- So far the effort was minimal
  - Refactor BC insns into functions
  - Interpreter's local variables go to the context
  - LLVM IR is just a sequence of calls
  - Constant pool is roughly the same
  - Control flow is a bit more involved
- We can do better
  - Use LLVM registers instead of the stack
  - Rewrite functions to take & return SEXPs

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
call void LDCONST_
call void LDCONST_
call void ADD_OP(
call void SETVAR_
```

```c
void instruction_LDCONST_OP(InterpreterContext * c, int arg1) {
  R_Visible = TRUE;
  c->value = VECTOR_ELT(c->constants, arg1);
  MARK_NOT_MUTABLE(c->value);
  BCNPUSH(c->value);
  NEXT();
}
```

```c
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```c
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  SEXP symbol = VECTOR_ELT(c->constants, arg1);
  loc = GET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);
  ...
  SEXP value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
}
```

```
> x = 2 + 3
```

```
void instruction_LDCONST_OP(InterpreterContext * c, int arg1);
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  SEXP symbol = VECTOR_ELT(c->constants, arg1);
  loc = GET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);
  ...
  SEXP value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
}
```

```
call void LDCONST_
call void LDCONST_
call void ADD_OP(
call void SETVAR_
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
call void LDCONST_
call void LDCONST_
call void ADD_OP(
call void SETVAR_
```

```
void instruction_LDCONST_OP(InterpreterContext * c, int arg1);
```

```
SEXP constant(SEXP consts, int index) {
  return VECTOR_ELT(consts, index);
}
```

```
void ADD_OP(InterpreterContext * c, int arg1) {
  FastBinary2(R_ADD, PLUSOP, R_AddSym, arg1);
  NEXT();
}
```

```
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  SEXP symbol = VECTOR_ELT(c->constants, arg1);
  loc = GET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);
  ...
  SEXP value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
}
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
call void LDCONST_
call void LDCONST_
call void ADD_OP()
call void SETVAR_
```

```c
void instruction_LDCONST_OP(InterpreterContext * c, int arg1);
```

```c
SEXP constant(SEXP consts, int index) {
  return VECTOR_ELT(consts, index);
}
```

```c
void ADD_OP(InterpreterContext * c, int arg1);
```

```c
SEXP genericAdd(SEXP lhs, SEXP rhs, SEXP rho, SEXP consts, int
call) {
  return cmp_arith2(
    VECTOR_ELT(consts, call),
    PLUSOP,
    R_AddSym,
    lhs,
    rhs,
    rho);
}
```

```c
void SETVAR_OP(InterpreterContext * c, int arg1) {
  SEXP loc;
  SEXP symbol = VECTOR_ELT(c->constants, arg1);
  loc = GET_BINDING_CELL_CACHE(symbol, c->rho, vcache, sidx);
  ...
  SEXP value = GETSTACK(-1);
  INCREMENT_NAMED(value);
  SET_BINDING_VALUE(loc, value))
  ...
  NEXT();
}
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
call void LDCONST_
call void LDCONST_
call void ADD_OP()
call void SETVAR_
```

```c
void instruction_LDCONST_OP(InterpreterContext * c, int arg1);
```

```c
SEXP constant(SEXP consts, int index) {
  return VECTOR_ELT(consts, index);
}
```

```c
void ADD_OP(InterpreterContext * c, int arg1);
```

```c
SEXP genericAdd(SEXP lhs, SEXP rhs, SEXP rho, SEXP consts, int
call) {
  return cmp_arith2(
    VECTOR_ELT(consts, call),
    PLUSOP,
    R_AddSym,
    lhs,
    rhs,
    rho);
}
```

```c
void SETVAR_OP(InterpreterContext * c, int arg1);
```

```c
void genericSetVar(SEXP value, SEXP rho, SEXP consts, int
symbol) {
  SEXP sym = VECTOR_ELT(consts, symbol);
  assert(sym != R_DotsSymbol && sym != R_UnboundValue);
  SEXP loc = GET_BINDING_CELL(sym, rho);
  INCREMENT_NAMED(value);
  if (! SET_BINDING_VALUE(loc, value)) {
    …
  }
}
```

```
> x = 2 + 3
```

```
LDCONST.OP 2
LDCONST.OP 3
ADD.OP
SETVAR.OP x
```

```
call void LDCONST_
call void LDCONST_
call void ADD_OP()
call void SETVAR_
```

```
%1 = call SEXP constant(2)
%2 = call SEXP constant(3)
%3 = call SEXP genericAdd(%1,%2)
call void genericSetVar(x, %3)
```

```
void instruction_LDCONST_OP(InterpreterContext * c, int arg1);
```

```
SEXP constant(SEXP consts, int index) {
  return VECTOR_ELT(consts, index);
}
```

```
void ADD_OP(InterpreterContext * c, int arg1);
```

```
SEXP genericAdd(SEXP lhs, SEXP rhs, SEXP rho, SEXP consts, int call) {
  return cmp_arith2(
    VECTOR_ELT(consts, call),
    PLUSOP,
    R_AddSym,
    lhs,
    rhs,
    rho);
}
```

```
void SETVAR_OP(InterpreterContext * c, int arg1);
```

```
void genericSetVar(SEXP value, SEXP rho, SEXP consts, int symbol) {
  SEXP sym = VECTOR_ELT(consts, symbol);
  otsSymbol && sym != R_UnboundValue);
  NDING_CELL(sym, rho);
  alue);
  _VALUE(loc, value)) {
}
```

GC is a Headache

- Unprotected SEXP in LLVM register
  - Never found by GC
- Statepoints to rescue
  - Precise locations of values stored in registers and on stack
  - Solves the issue
- But is a pain

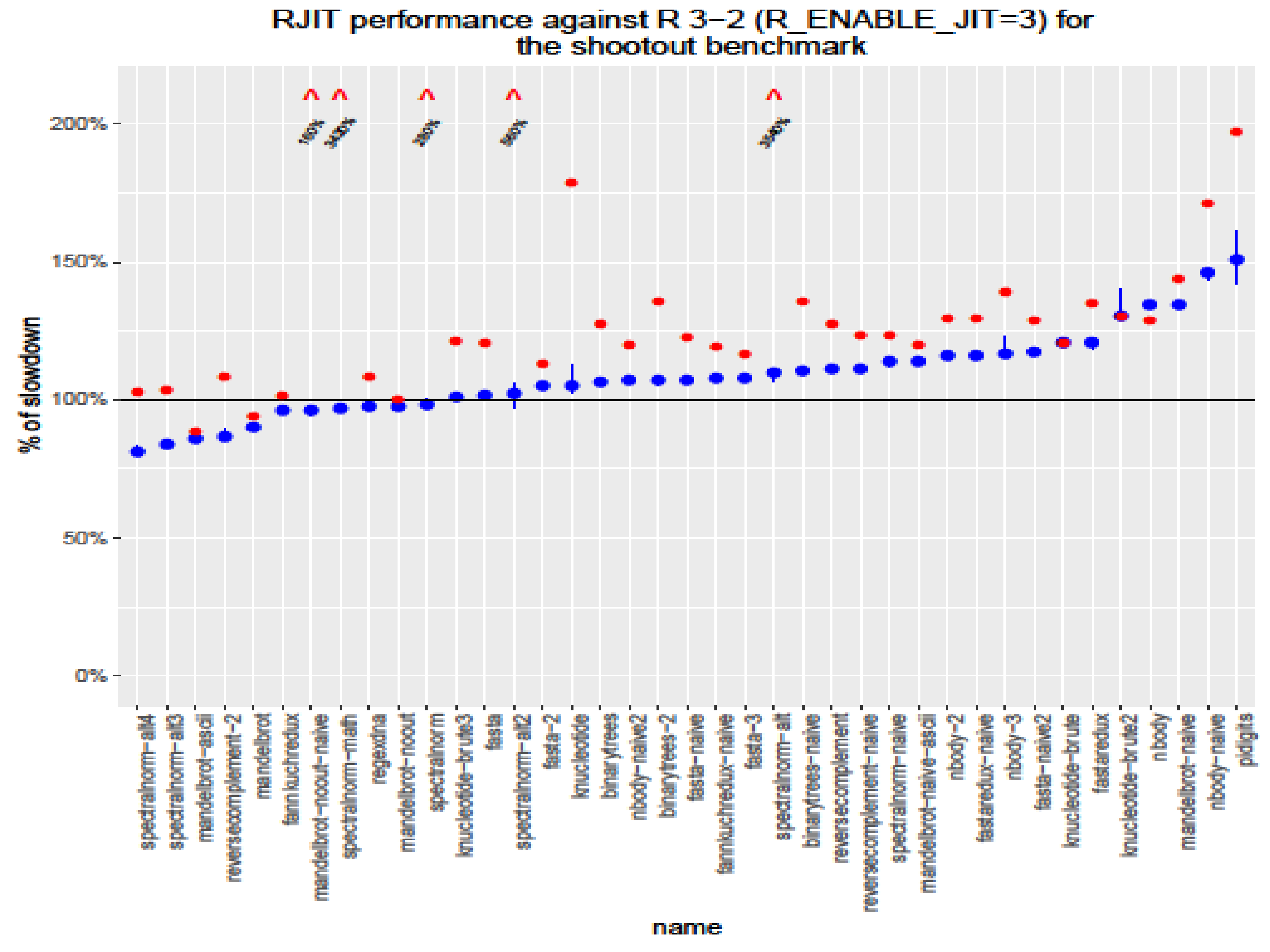
```
define %struct.SEXPREC addrspace(1)* @rfunction(%struct.SEXPREC addrspace(1)* %body, %struct.SEXPREC addrspace(1)* %rho, i32 %useCache) #0 gc
"statepoint-example" {
start:
  %safepoint_token = call i32 (i64, i32, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*)*, i32, i32, ...)
@llvm.experimental.gc.statepoint.p0f_p1struct.SEXPRECp1struct.SEXPRECp1struct.SEXPRECf(i64 4, i32 0, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC
addrspace(1)*, %struct.SEXPREC addrspace(1)*)* @genericGetVar, i32 2, i32 0, %struct.SEXPREC addrspace(1)* inttoptr (i64 29444840 to %struct.SEXPREC
addrspace(1)*), %struct.SEXPREC addrspace(1)* %rho, i32 0, i32 0, %struct.SEXPREC addrspace(1)* %rho) #2
  %rho.relocated = call coldcc i8 addrspace(1)* @llvm.experimental.gc.relocate.p1i8(i32 %safepoint_token, i32 9, i32 9) ; (%rho, %rho)
  %rho.relocated.casted = bitcast i8 addrspace(1)* %rho.relocated to %struct.SEXPREC addrspace(1)*
  %0 = call %struct.SEXPREC addrspace(1)* @llvm.experimental.gc.result.p1struct.SEXPREC(i32 %safepoint_token)
  %safepoint_token.1 = call i32 (i64, i32, i32 (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*)*, i32, i32, ...)
@llvm.experimental.gc.statepoint.p0f_i32p1struct.SEXPRECp1struct.SEXPRECf(i64 4, i32 0, i32 (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC
addrspace(1)*)* @convertToLogicalNoNA, i32 2, i32 0, %struct.SEXPREC addrspace(1)* %0, %struct.SEXPREC addrspace(1)* inttoptr (i64 29444840 to
%struct.SEXPREC addrspace(1)*), i32 0, i32 0, %struct.SEXPREC addrspace(1)* %0, %struct.SEXPREC addrspace(1)* %rho.relocated.casted) #2
  %1 = call coldcc i8 addrspace(1)* @llvm.experimental.gc.relocate.p1i8(i32 %safepoint_token.1, i32 9, i32 9) ; (%0, %0)
  %rho.relocated6 = call coldcc i8 addrspace(1)* @llvm.experimental.gc.relocate.p1i8(i32 %safepoint_token.1, i32 10, i32 10) ; (%rho.relocated.casted,
%rho.relocated.casted)
  %2 = call i32 @llvm.experimental.gc.result.i32(i32 %safepoint_token.1)
  %condition = icmp eq i32 %2, 1
  br i1 %condition, label %ifTrue, label %ifFalse

ifTrue:                                           ; preds = %start
  %3 = bitcast i8 addrspace(1)* %rho.relocated6 to %struct.SEXPREC addrspace(1)*
  %safepoint_token.2 = call i32 (i64, i32, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*)*, i32, i32,
...)
llvm.experimental.gc.statepoint.p0f_p1struct.SEXPRECp1struct.SEXPRECp1struct.SEXPRECf(i64 4, i32 0, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC
addrspace(1)*, %struct.SEXPREC addrspace(1)*)* @genericGetVar, i32 2, i32 0, %struct.SEXPREC addrspace(1)* inttoptr (i64 44860416 to %struct.SEXPREC
addrspace(1)*), %struct.SEXPREC addrspace(1)* %3, i32 0, i32 0, %struct.SEXPREC addrspace(1)* %3) #2
  %rho.relocated8 = call coldcc i8 addrspace(1)* @llvm.experimental.gc.relocate.p1i8(i32 %safepoint_token.2, i32 9, i32 9) ; (%3, %3)
  %4 = call %struct.SEXPREC addrspace(1)* @llvm.experimental.gc.result.p1struct.SEXPREC(i32 %safepoint_token.2)
  br label %next

ifFalse:                                          ; preds = %start
  %5 = bitcast i8 addrspace(1)* %rho.relocated6 to %struct.SEXPREC addrspace(1)*
  %safepoint_token.3 = call i32 (i64, i32, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*)*, i32, i32,
...) @llvm.experimental.gc.statepoint.p0f_p1struct.SEXPRECp1struct.SEXPRECp1struct.SEXPRECf(i64 4, i32 0, %struct.SEXPREC addrspace(1)*
(%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*)* @genericGetVar, i32 2, i32 0, %struct.SEXPREC addrspace(1)* inttoptr (i64 32777984 to
%struct.SEXPREC addrspace(1)*), %struct.SEXPREC addrspace(1)* %5, i32 0, i32 0, %struct.SEXPREC addrspace(1)* %5) #2
  %rho.relocated10 = call coldcc i8 addrspace(1)* @llvm.experimental.gc.relocate.p1i8(i32 %safepoint_token.3, i32 9, i32 9) ; (%5, %5)
  %6 = call %struct.SEXPREC addrspace(1)* @llvm.experimental.gc.result.p1struct.SEXPREC(i32 %safepoint_token.3)
  br label %next

next:                                             ; preds = %ifFalse, %ifTrue
  %7 = phi %struct.SEXPREC addrspace(1)* [ %4, %ifTrue ], [ %6, %ifFalse ]
  ret %struct.SEXPREC addrspace(1)* %7
}
```

Pushing Further

- Getting rid of stack helps
- No interpreter loop

- But every BC instruction is a call
  - Simple bytecodes can be translated to LLVM directly

- Specialized faster versions can be added
- Inline caching & native calls for builtins
  - Speculative?


RJIT performance against R 3−2 (R_ENABLE_JIT=3) for the shootout benchmark
% of slowdown
200%
150%
100%
50%
0%
name
spectralnorm−alt4
spectralnorm−alt3
mandelbrot−ascii
reversecomplement−2
mandelbrot
fannkuchredux
mandelbrot−noout−naive
spectralnorm−math
regexdna
mandelbrot−noout
spectralnorm
knucleotide−brute3
fasta
spectralnorm−alt2
fasta−2
knucleotide
binarytrees
nbody−naive2
binarytrees−2
fasta−naive
fannkuchredux−naive
fasta−3
spectralnorm−alt
binarytrees−naive
reversecomplement
reversecomplement−naive
spectralnorm−naive
mandelbrot−naive−ascii
nbody−2
fastaredux−naive
nbody−3
fasta−naive2
knucleotide−brute
fastaredux
knucleotide−brute2
nbody
mandelbrot−naive
nbody−naive
pidigits

- Local transformations only get you so far…

In the end we need to optimize

- Local transformations only get you so far…

- LLVM has great optimizers
  - Turns out these are good for C/C++

- R is way too high-level for LLVM to do much
  - Everything is a SEXP
  - Arguments passed in evironments
  - Most functionality done by runtime functions, opaque to LLVM

High level optimizations in LLVM

- We started breaking GNU-R instructions into smaller reusable components
- But the more involved the compilation was the more we realized LLVM IR is not good at representing high-level concepts
- Do high level optimizations before translating to LLVM IR

RIR

Yet Another R Bytecode

Why Another Bytecode?

- R Bytecode is optimized for fast execution
  - Having few instructions mitigates the interpreter switch overhead
  - Having generic instructions mitigates static optimizer

- JIT does not care how many instructions you have

- Optimizer works better if instructions are predictable

```
> f(a, b, c, d)
```

```
> f(a, b, c, d)
```

```
GETFUN.OP 1 // f
MAKEPROM.OP 4 // a
MAKEPROM.OP 5 // b
MAKEPROM.OP 6 // c
MAKEPROM.OP 7 // d
CALL.OP 2
RETURN.OP
```

Loads the function, pushes on stack, pushes empty args on stack

Depending on what function is loaded at runtime:

Makes a promise (default)
Evaluates (builtins)
Does nothing (specials)

Does MAKEPROM evaluate?
Which arguments function takes?
Non-local promise code

```
> f(a, b, c, d)
```

```
GETFUN.OP 1 // f
MAKEPROM.OP 4 // a
MAKEPROM.OP 5 // b
MAKEPROM.OP 6 // c
MAKEPROM.OP 7 // d
CALL.OP 2
RETURN.OP
```

Loads function

Calls function, makes promises, or evaluates

Promises kept locally with the code

```
    ldfun_   3 # f
    call_ [ 0 1 2 3]
    ret_
@0
    ldvar_   4 # a
    ret_
@1
    ldvar_   5 # b
    ret_
@2
    ldvar_   6 # c
    ret_
@3
    ldvar_   7 # d
    ret_
```

Different calls for different needs
(call_, static_call_stack_, ...) (*)

Speculative

- Most optimizations are unsound in R
- But most of the time, they are OK
- Speculate they are ok
  - Revert to unoptimized code if they are not (*)

```
> sum(a)
```

```
guard_fun_  sum == 0x154c410
ldvar_   4 # a
static_call_stack_  1  0x154c410
ret_
```

Optimization Framework

- Abstract Interpretation

- Easily extendable classes for different analyses & optimizations

- Worst case is a **big** issue
  - In the worst case every variable read may trigger a promise which may invalidate **all** local state
  - Speculation to the rescue

```
> a = 1; b = 2; a + b;
```

```
guard_fun_  = == 0x153add0
push_  16 # [1] 1
set_shared_
stvar_  4 # a
push_  17 # [1] 2
set_shared_
stvar_  5 # b
guard_fun_  + == 0x1540800
~~ local
ldvar_  4 # a
~~ local
ldvar_  5 # b
~~ TOS : const,
pop_
~~ TOS : const,
pop_
push_  18 # [1] 3
ret_
```

Load guaranteed to succeed in local env

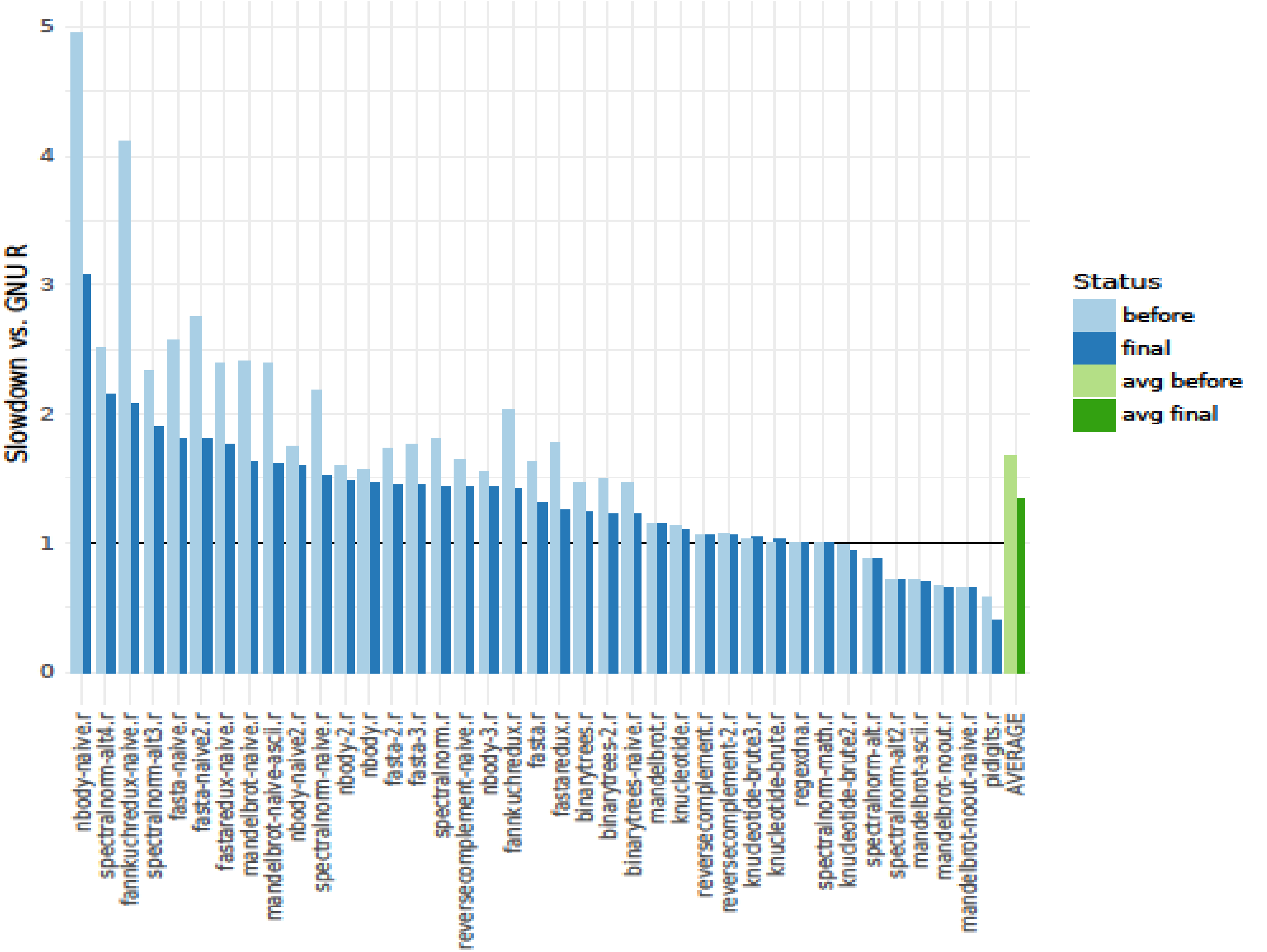
TOS is constant before pop

```
> a = 1; b = 2; a + b;
```

```
guard_fun_  = == 0x153add0
push_  16 # [1] 1
set_shared_
stvar_  4 # a
push_  17 # [1] 2
set_shared_
stvar_  5 # b
guard_fun_  + == 0x1540800
push_  18 # [1] 3
ret_
```

Performance Matters

- RIR currently does not have JIT
  - The plan is to use LLVM after sufficient amount of high level opts is done
- Improvements on baseline
- Adding more specialized instructions
- More performant interpreter loop
- Optimizations



Slowdown vs. GNU R
0
1
2
3
4
5
Status
before
final
avg before
avg final
nbody-naive.r
spectralnorm-alt4.r
fannkuchredux-naive.r
spectralnorm-alt3.r
fasta-naive.r
fasta-naive2.r
fastaredux-naive.r
mandelbrot-naive.r
mandelbrot-naive-ascii.r
nbody-naive2.r
spectralnorm-naive.r
nbody-2.r
nbody.r
fasta-2.r
fasta-3.r
spectralnorm.r
reversecomplement-naive.r
nbody-3.r
fannkuchredux.r
fasta.r
fastaredux.r
binarytrees.r
binarytrees-2.r
binarytrees-naive.r
mandelbrot.r
knucleotide.r
reversecomplement.r
reversecomplement-2.r
knucleotide-brute3.r
knucleotide-brute.r
regexdna.r
spectralnorm-math.r
knucleotide-brute2.r
spectralnorm-alt.r
spectralnorm-alt2.r
mandelbrot-ascii.r
mandelbrot-noout.r
mandelbrot-noout-naive.r
pidigits.r
AVERAGE

Future

- Improvements to the baseline
- More optimizations
  - Control Flow Analysis
  - Removing promises
  - Inferring types
  - Tracking functions
  - Escape Analysis
  - …
- Better speculation
- Adding a JIT

Thank You

https://github.com/reactorlabs/rjit

https://github.com/reactorlabs/rir