UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization

Haichuan Wang[1], Peng Wu[2], David Padua[1]

[1] University of Illinois at Urbana-Champaign [2] Huawei U.S. Research

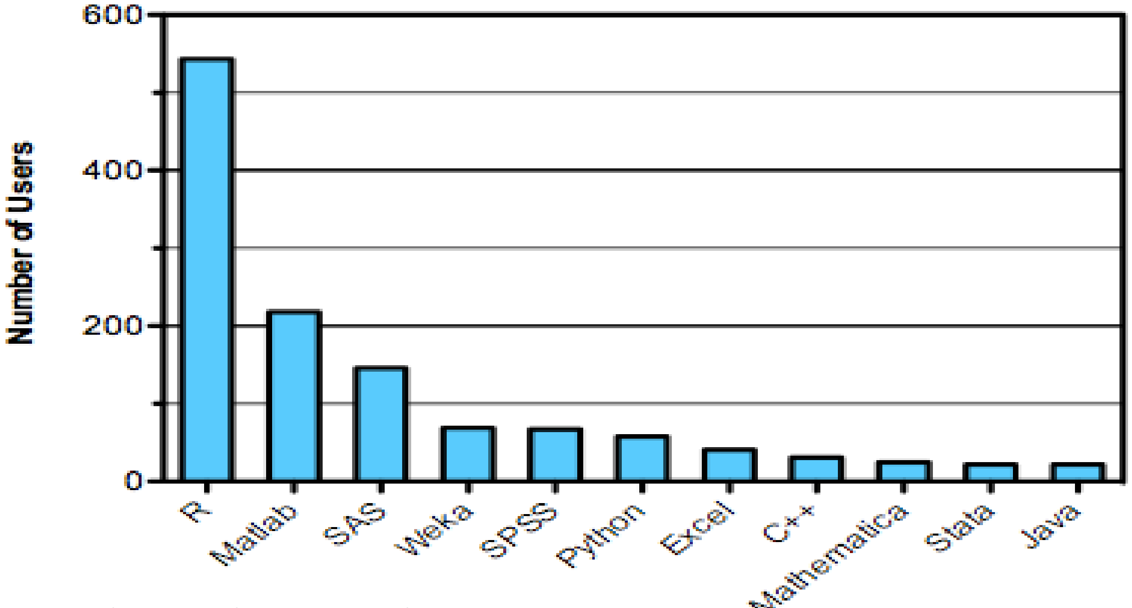
Outline

- R Background
- GNU R VM and Performance Analysis
- Our Solution – ORBIT (***O***ptimized ***R*** ***B***yte-code ***I***nterpre***T***er)
- Performance Evaluation
- Conclusion

R Background

- R language
  - Dynamic Scripting Language, used in statistics domain
  - Origin from S language of Bell Lab
- R GNU Virtual Machine
  - The reference R implementation, maintained by about 20 people
- The language for data analytics in the age of Big Data

Tool Used By Competitors in Data Analytics Competitions at Kaggle.com



source: http://r4stats.com/articles/popularity/

Different R Programming Styles

Type I: Looping Over Data

```
for (j in 1:500) {
  for (k in 1:500) {
    jk<-j - k;
    b[k,j] <- abs(jk) + 1
  }
}
```

(1) ATT bench: creation of Toeplitz matrix

Type II: Vector Programming

```
males_over_40 <- function(age, gender) {
  age >= 40 & gender == 1
}
```

(2) Riposte bench: a and g are large vectors

Type III: Native Library Glue

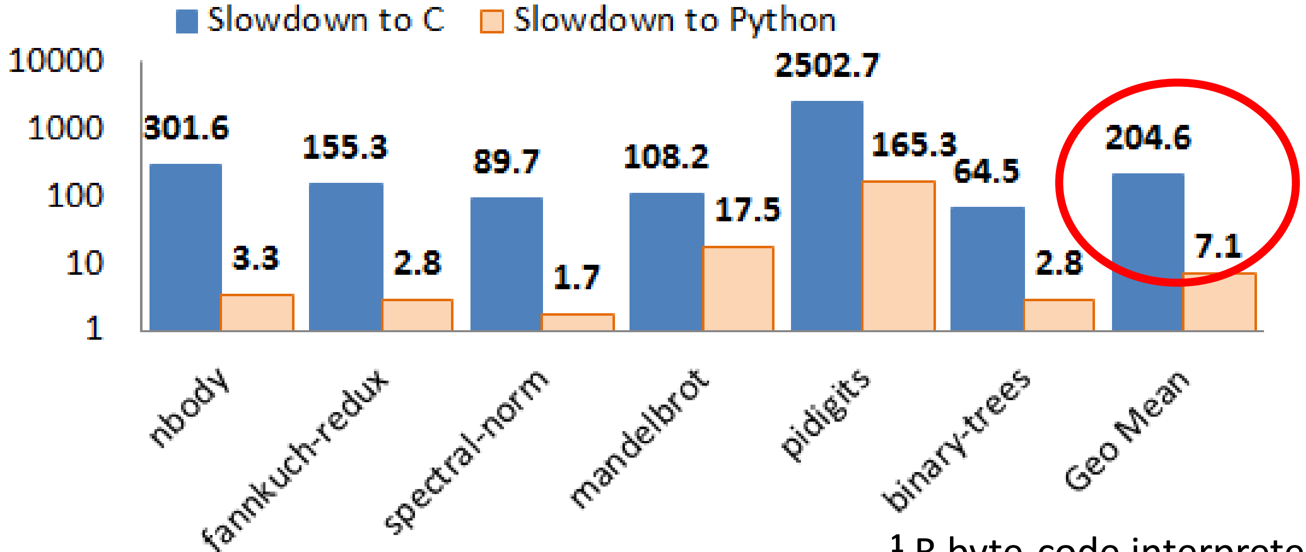
```
a <- rnorm(2000000);
b <- fft(a)
```

(3) ATT bench: FFT over 2 Million random values

Performance Issues with Type I (Loop) R Programs

- Speed

Slowdown of R[1] on the Shootout benchmarks relative to C and CPython



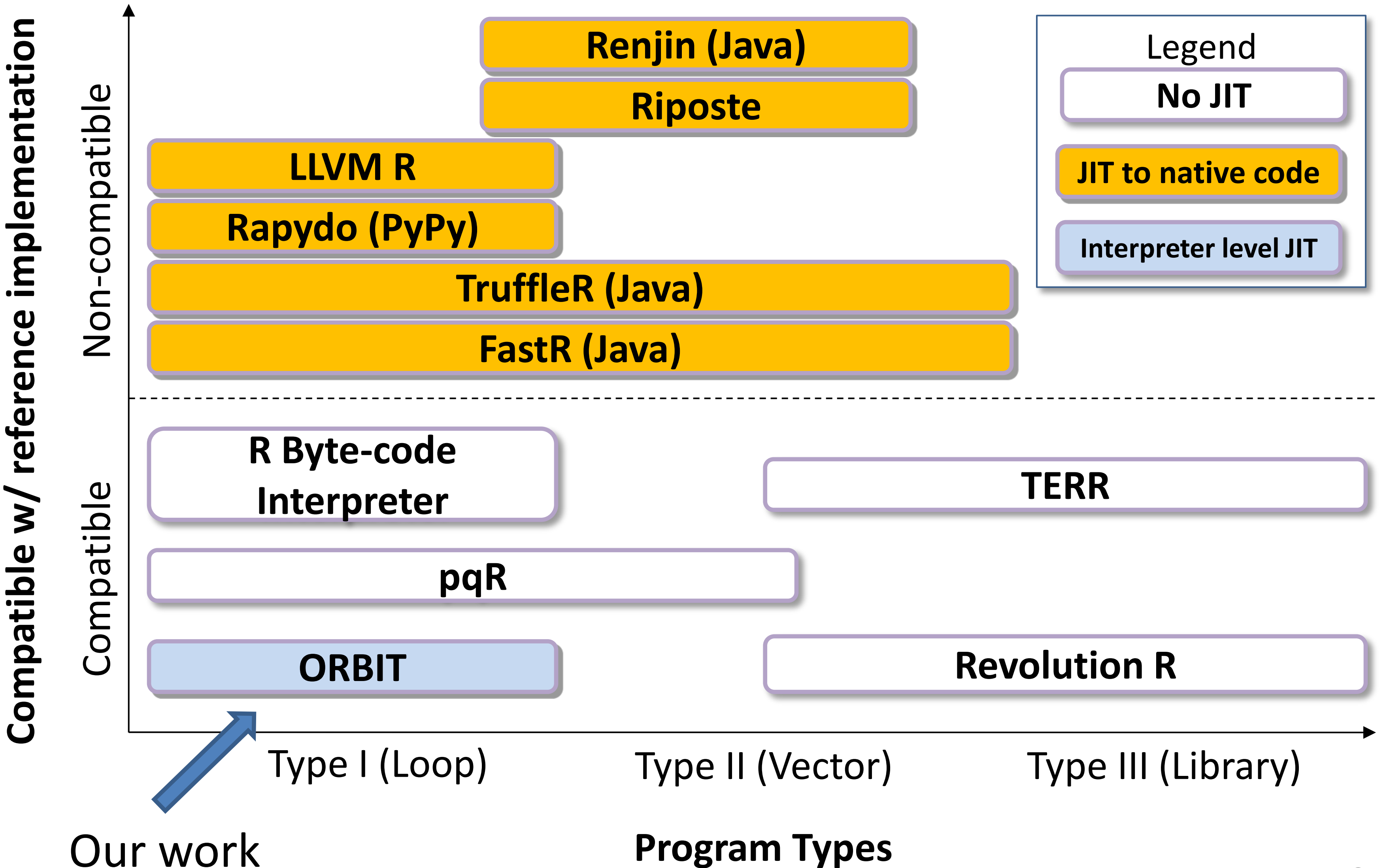
[1] R byte-code interpreter

- Memory Consumption/Allocation

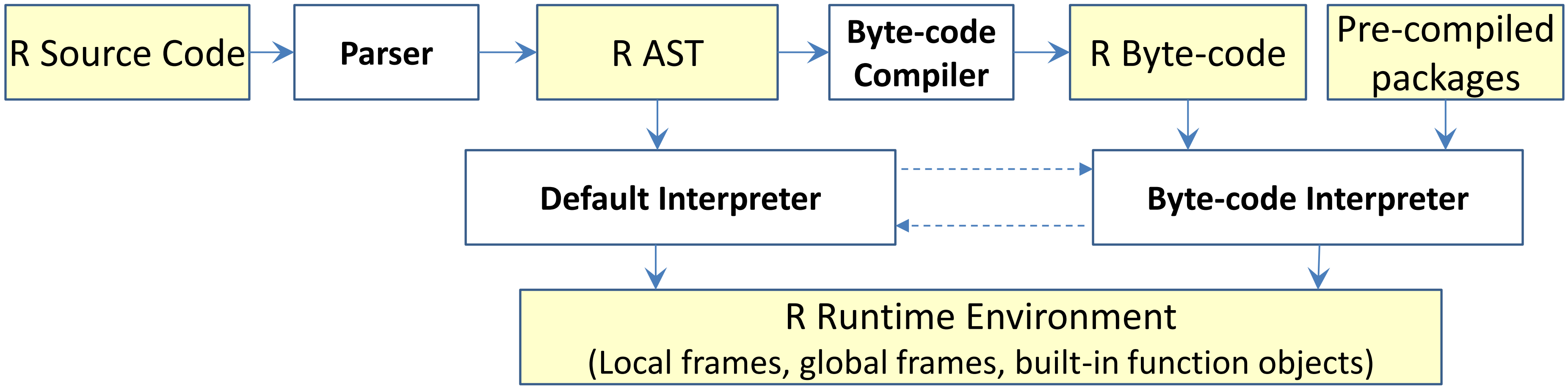
```
r <- 0;
for( i in 1:1000000) { #1M
  r <- r + i;
}
print(r);
```

	R byte-code Interpreter
Machine Instructions	**327 M**
SEXPREC Object Allocated	**20**
VECTOR Scalar Allocated	**1 M**
VECTOR Non-scalar Allocated	**2**

Related Work



The GNU R VM



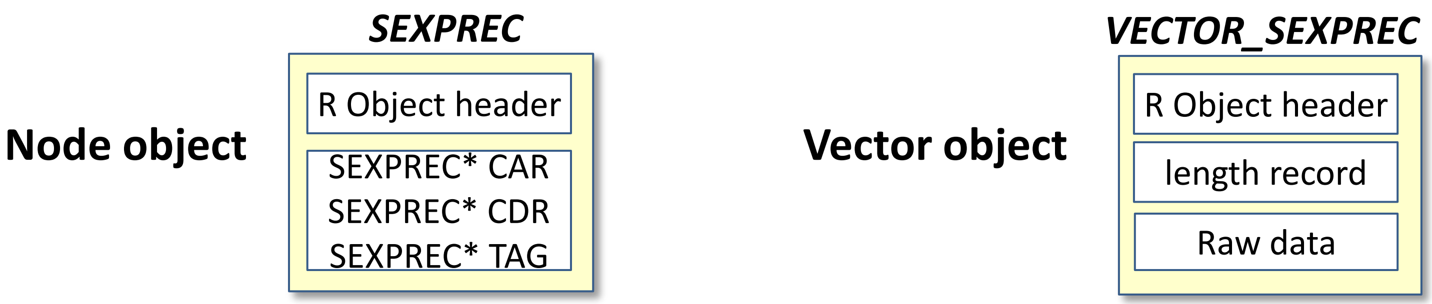
- Default Interpreter
  - AST interpreter
- Byte-code Interpreter
  - Stack VM based interpreter
- Both interpreters
  - Share the same R runtime environment
  - Use the same object model

Problems Analysis – Slow Speed

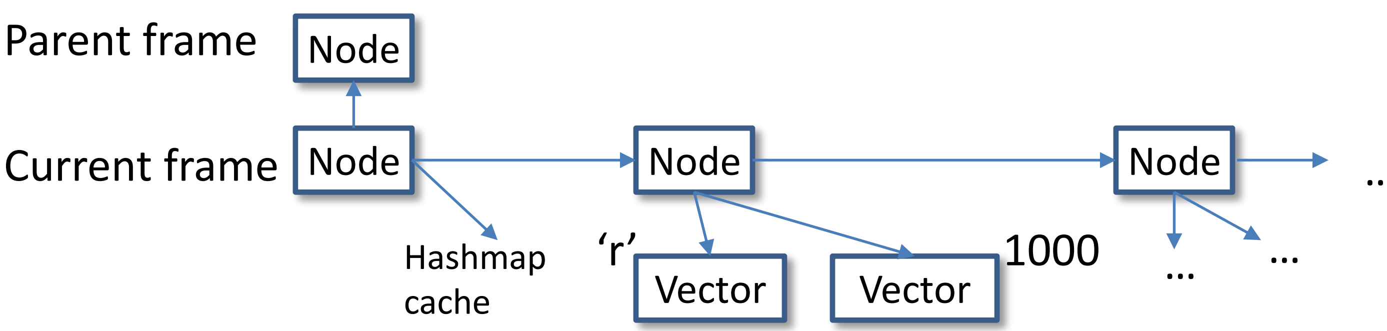
- Reasons
  - Common problems of Dynamic scripting languages
    - …
  - R specific semantics
    - …
  - Overhead from R’s generic object representation
    - Instructions for allocation and garbage collection

Problem Analysis - Memory Consumption/Allocation

- Generic Object Representation
  - Two basic meta object types for all



  - All runtime and user type objects are expressed with the two types
    - E.g. local frame context: linked list by node objects



    - E.g. matrix: vector object (data) + linked list(attributes) + vector objects ('dim', dim sizes)

Optimizations in GNU R

- Improving Speed
  - Translate into byte-code
  - Byte-code interpreter: direct threading code dispatch
  - Classic compiler optimizations to the byte-code
  - Copy-on-write
  - …

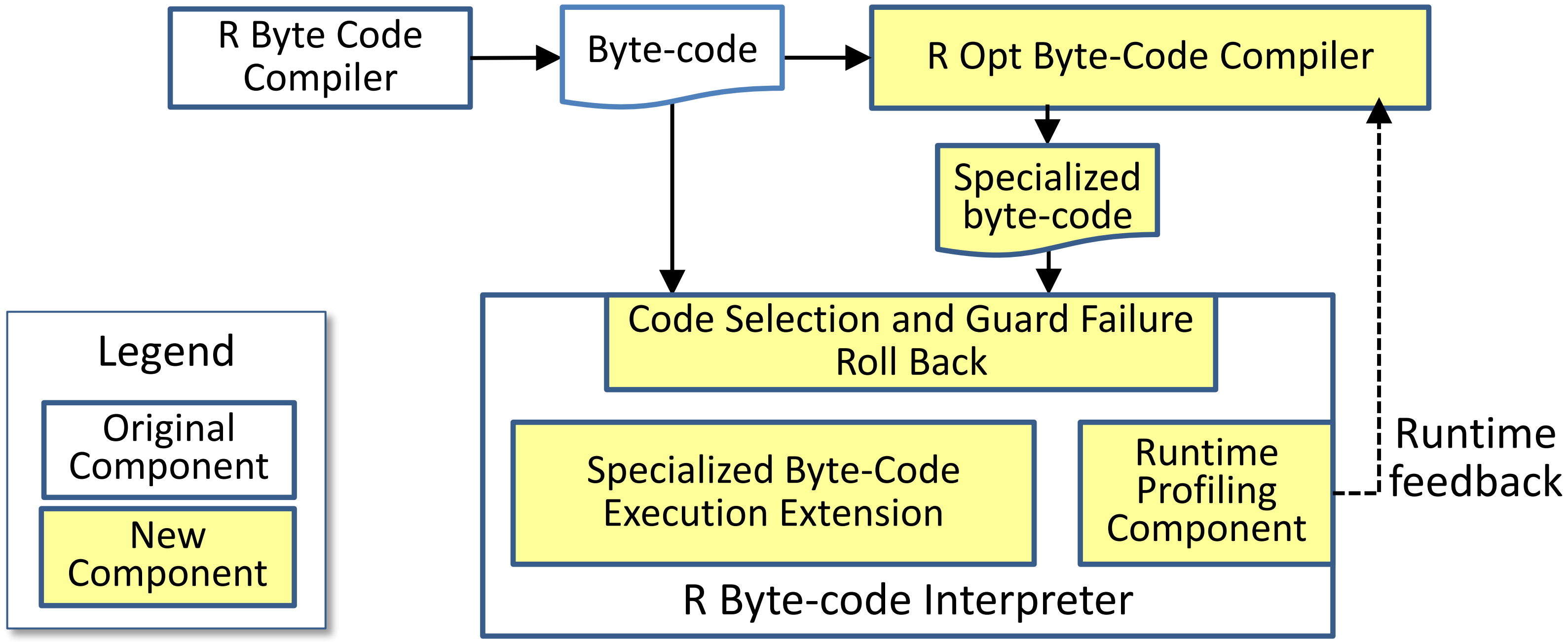
The generic byte-code instruction set does not change !

- Optimizing Memory system
  - Memory Allocator
    - Pre-allocate pages of SEXPREC
    - Pre-allocates different sizes of small VECTOR_SEXPREC
  - Garbage Collator
    - Stop-world, multi-generation based collector

The representation of generic objects does not change!

ORBIT – *O*ptimized *R* *B*yte-code *I*nterpre*T*er

- Focus on Specialization
  - Generic byte-code → type specialized byte-code
  - Generic data representation → specialized data representation
- Rely on runtime feedback
  - Aggressive: profile once → speculative typing
- Pure interpreter approach, no native code generation
- Be compatible with the GNU R implementation



An Example of ORBIT Specialization

Source

```
foo <- function(a) {
  b <- a + 1
}
```

Byte-code Symbol table

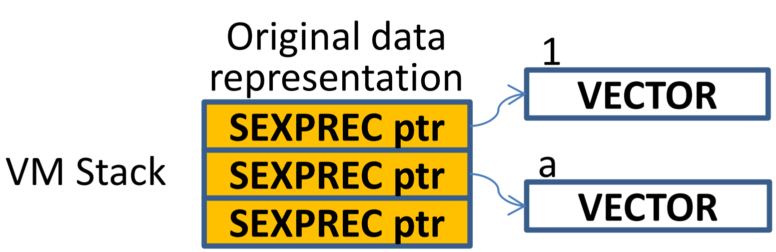
Idx	Value
1	“a”
2	1
3	a+1
4	b

Generic Domain

Byte-Code

Profile point → PC 1 (If “a” is real scalar)

PC	STMTS
1	GETVAR, 1
3	LDCONST, 2
5	ADD, 3
7	SETVAR, 4
9	INVISIBLE
10	RETURN

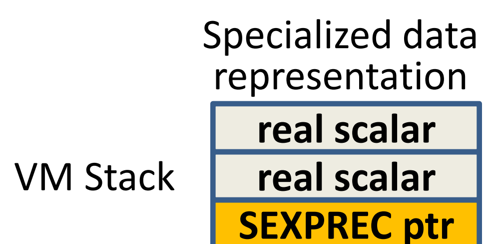


ORBIT →

Specialized Domain

Specialized byte-code

PC	STMTS
1	GETREALUNBOX, 1
3	LDCONSTREAL, 2
5	REALADD
6	SETUNBOXVAR, 4
	…



ORBIT Approach Highlight

- Type profiling + Fast type inference
  - Profiling once -> trigger optimization
  - Simple type system, use profiling type to help typing
- Specialized data representation
  - Use raw (unboxed) objects to replace generic objects
  - Mixed Stack to store boxed and unboxed objects
  - With a type stack to track unboxed objects in the stack
  - Unbox value cache: a software cache for faster local frame object access
- Specialized byte-code and runtime function routines
  - Type specialized instructions for common operations
  - Simplify calling conventions according to R's semantics
- Guards to handle incorrect type speculation
  - Type change → Guard failure → Restore the generic code and object
  - Combine the new type with the original profiling type → Retry optimization later

For Loop Performance Metrics

```
r <- 0;
for( i in 1:1000000) {
    r <- r + i;
}
print(r);
```

	R byte-code	ORBIT
Machine Instructions	327 M	98 M
SEXPREC Object Allocated	20	17
VECTOR Scalar Allocated	1 M	9
VECTOR Non-scalar Allocated	2	0

- Memory allocated removed
  - The long 1:1000000 object
  - New “r” value used in each iteration

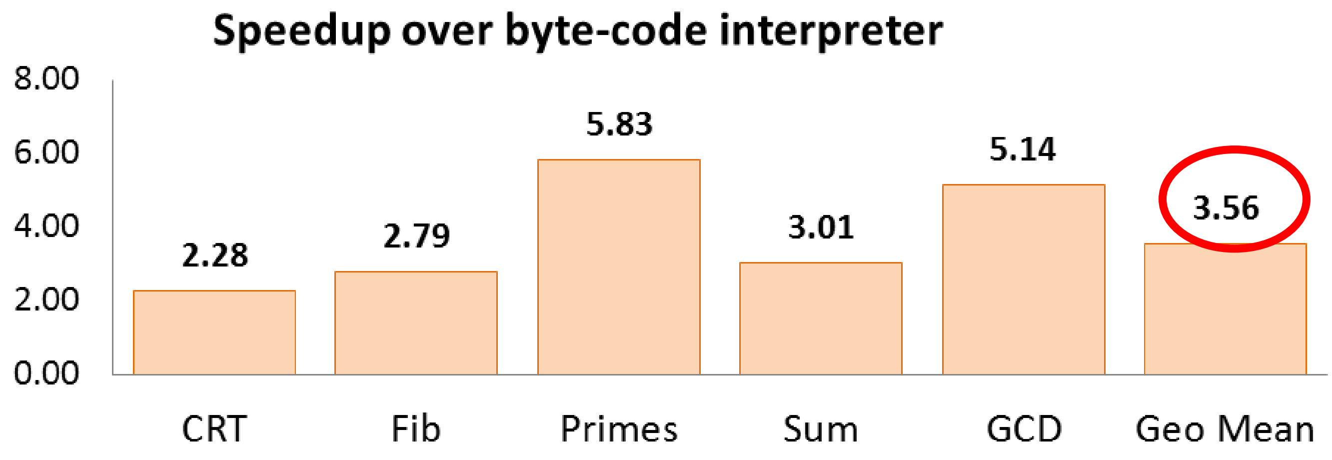
Performance Evaluation

- Benchmarks – Type I code
  - Scalar benchmark suite

CRT	Chinese Remainder Theorem
Fib	fibonacci number, iterative method
Sum	For loop based accumulation
GCD	Greatest Common Divisor for 100M pairs of random numbers
Primes	Find prime numbers

  - Shootout benchmark suite
    - nbody, fannkuch-redux, spectral-norm, mandelbrot, pidigits
- Environment
  - CPU: Xeon(R) CPU E31245 @3.30GHz (Turbo boost disabled)
  - Linux: Fedora 16 (3.1.0-0.rc10.git0.1.fc16.x86_64)
  - R VMs:
    - Byte-code interprter: R-2.14.1 with byte-code compiling enabled
    - ORBIT: R-2.14.1 with ORBIT extensions

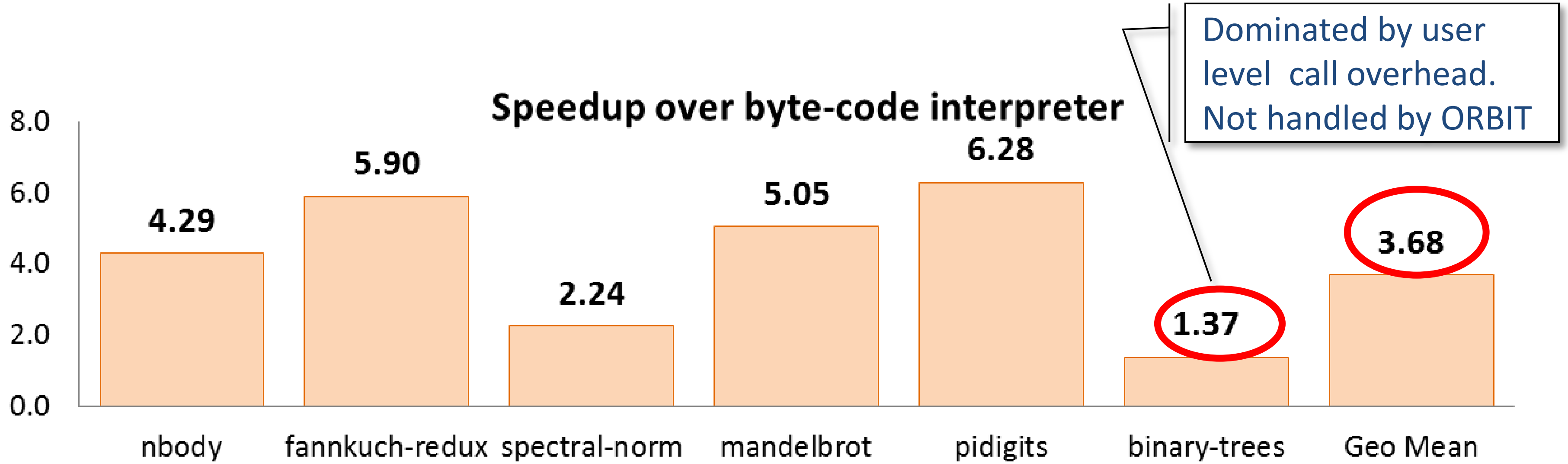
Performance of ORBIT – Scalar Benchmark



Percentage of Memory Allocation Reduced

Benchmark	SEXPREC	VECTOR scalar	VECTOR non-scalar
CRT	76.06%	82.83%	97.58%
Fib	99.16%	99.99%	100%
Primes	98.21%	94.70%	50.00%
Sum	15.00%	99.99%	100%
GCD	99.99%	99.99%	25.00%
Mean	77.68%	95.50%	74.52%

Performance of ORBIT – Shootout Benchmark



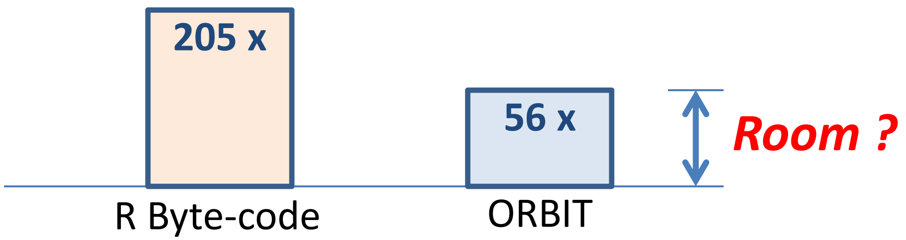
Percentage of Memory Allocation Reduced

Benchmark	SEXPREC	VECTOR scalar	VECTOR non-scalar
nbody	**85.47%**	**86.82%**	**69.02%**
fannkuch-redux	**99.99%**	**99.30%**	**71.98%**
spectral-norm	**43.05%**	**91.46%**	**99.46%**
mandelbrot	**99.95%**	**99.99%**	**99.99%**
pidigits	**96.89%**	**98.37%**	**95.13%**
Binary-trees	**36.32%**	**67.14%**	**0.00%**
Mean	**76.95%**	**90.51%**	**72.60%**

Conclusion

- Our Work
  - Revealed Generic Object Representation is a key source of low performance
  - Focused on specialization
    - Operation specialization + Object representation specialization
  - Implemented a JIT engine, pure interpreter based
  - Reduced instruction path length and memory allocations
- Next Step

Shootout – Slowdown to C implementation



- Need Better Benchmarks for R
  - An community effort: https://github.com/rbenchmark/benchmarks

Thank You!

Contact Info:

Haichuan Wang (hwang154@illinois.edu)
Peng Wu (pengwu@acm.org)
David Padua (padua@illinois.edu)