DSC2014

Optimizing R VM: Interpreter-level Specialization and Vectorization

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

[1] University of Illinois at Urbana-Champaign [2] Huawei America Lab

Our Taxonomy - Different R Programming Styles

Type I: Looping Over Data

```
b <- rep(0, 500*500);
dim(b) <- c(500, 500)
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

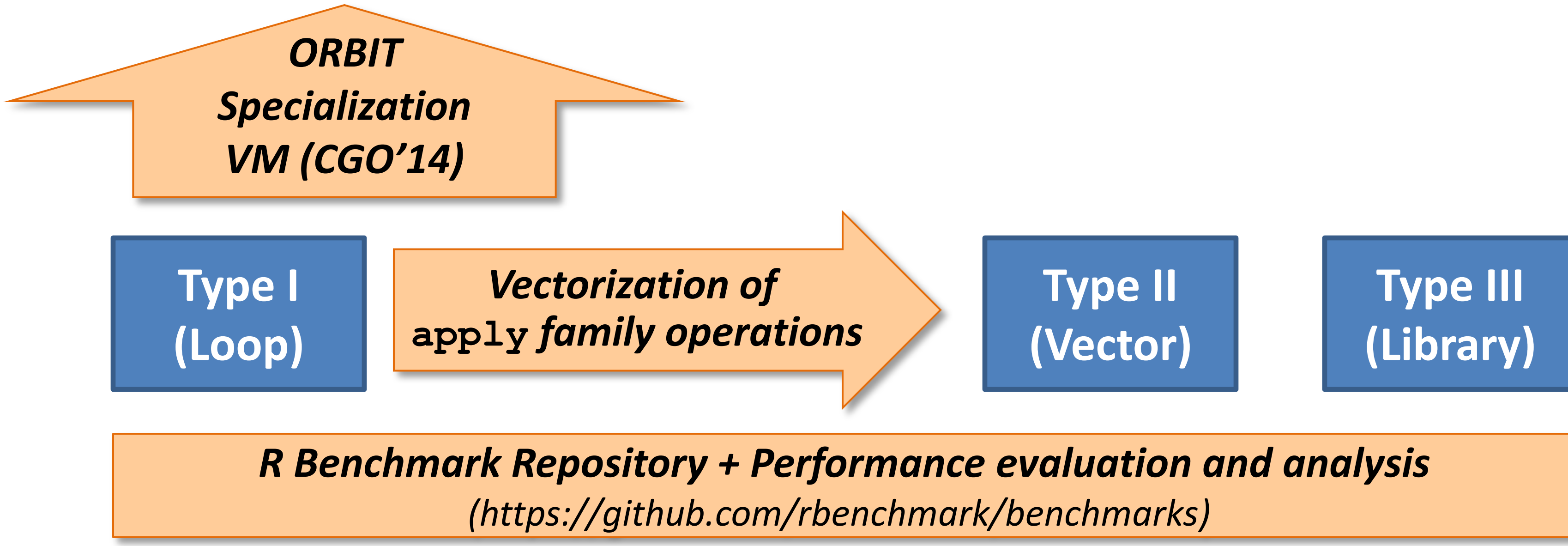
Type III: Native Library Glue

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

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

Our Project - ORBIT

- Approaches



- ***Pure Interpreter***
  - Portable, Simple. Interesting research problem
- ***Compiler plus Runtime***
  - Simplify the compiler analysis. Have to use runtime info due to the dynamics

Specialization

Source

```
a + 1
```

Byte-code

```
GETVAR_OP, 1
LDCONST_OP, 2
ADD_OP
```

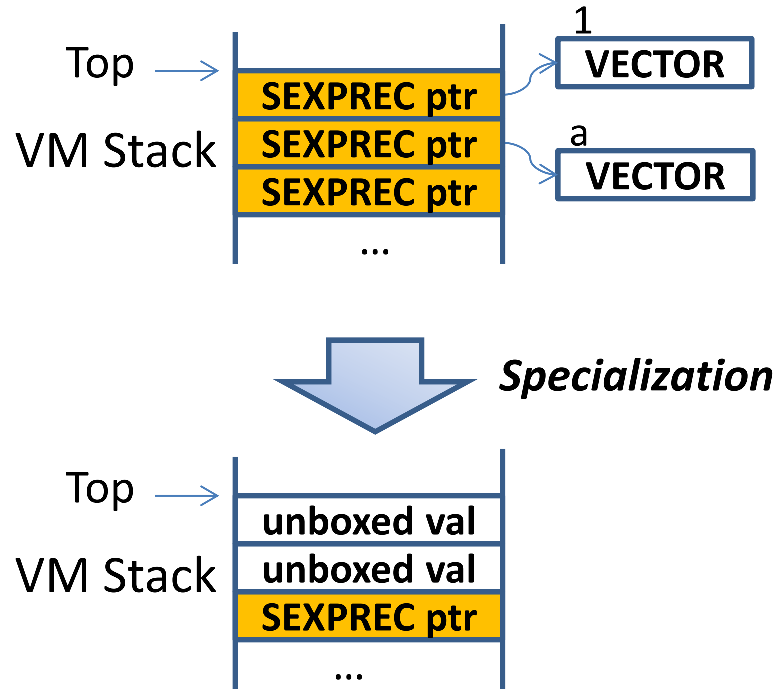
Operation Side

```
int typex = ...
int typey = ...
if(typex == REALSXP) {
  if(typey == REALSXP)
     ...
  else if (...)
     ...
}
else if (typex == INTSXP && ... )
  if(typey == REALSXP)
     ...
  else if (...)
     ...
}
Arith2(...) //Handle complex case
```

Specialization

REALADD_OP	*REALVECADD_OP*	*ADD_OP*
INTADD_OP	*INTVECADD_OP*	
SCALADD_OP	*VECADD_OP*	

Data Object Side



More Specialization are Required in the Object Side

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

Node object

SEXPREC
sxpinfo_struct sxpinfo
SEXPREC* attrib
SEXPREC* pre_node SEXPREC* next_node
SEXPREC* CAR SEXPREC* CDR SEXPREC* TAG

Vector object

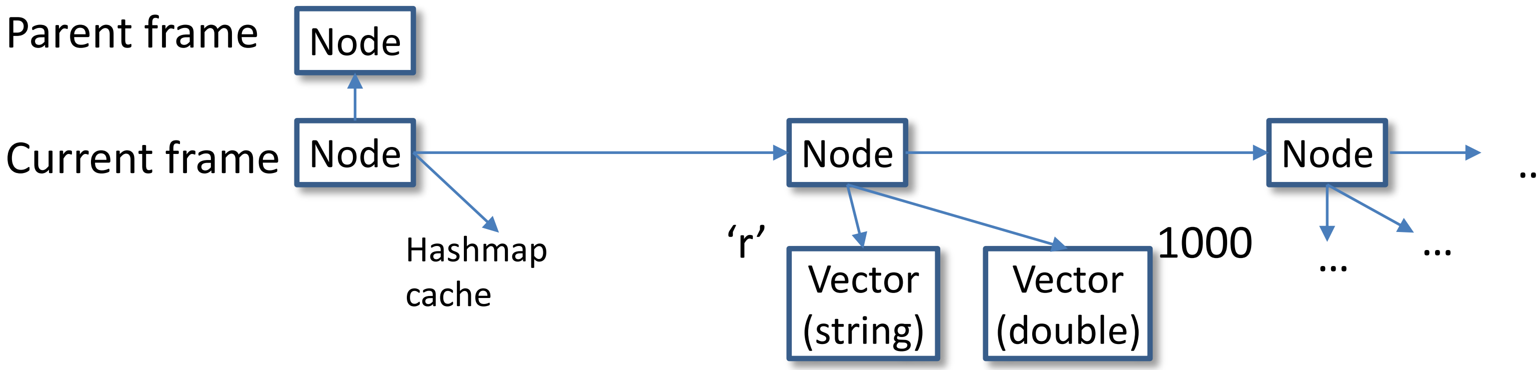
VECTOR_SEXPREC
sxpinfo_struct sxpinfo
SEXPREC* attrib
SEXPREC* pre_node SEXPREC* next_node
R_len_t length R_len_t truelength
Vector raw data

  - All runtime and user type objects are expressed with the two types

Generic Object Representation – Two Examples

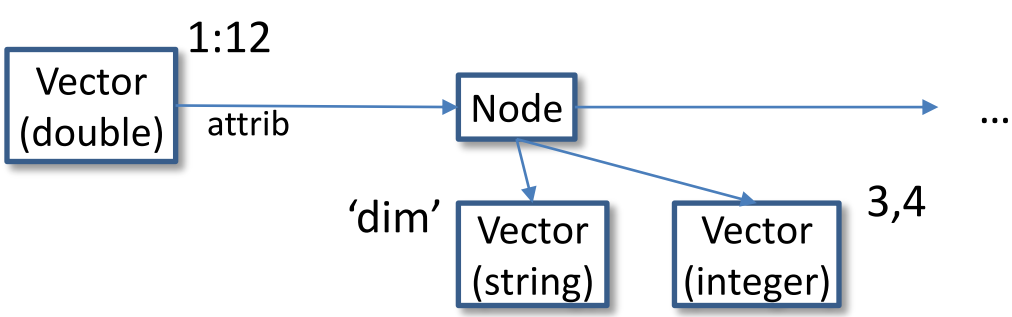
- Local Frames (linked list)

```
r <- 1000
```



- Matrix (vector + linked list)

```
matrix(1:12, 3, 4)
```



Data Object Specialization – Implemented in ORBIT

- Approaches
  - 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
- Results

```
b <- rep(0, 500*500);
dim(b) <- c(500, 500)
for (j in 1:500) {
  for (k in 1:500) {
    jk<-j - k;
    b[k,j] <- abs(jk) + 1
  }
}
```

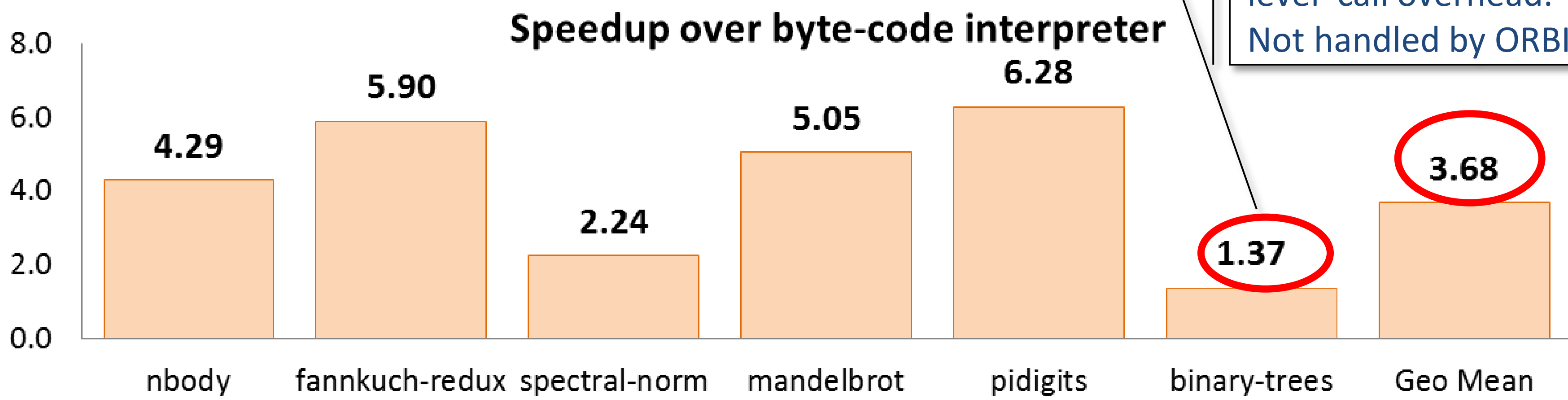
(1) ATT bench: creation of Toeplitz matrix

GNU R VM Memory System Metrics

	Byte-code Interpreter	ORBIT
GC Time (ms)	32.0	14.8
Node objs allocated	3,753,112	750,104
Vector scalar objs allocated	3,004,534	2,251,526
Vector non-scalar allocated	3,032	23

Performance of ORBIT – Shootout Benchmark

Dominated by user level call overhead. Not handled by ORBIT



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%**

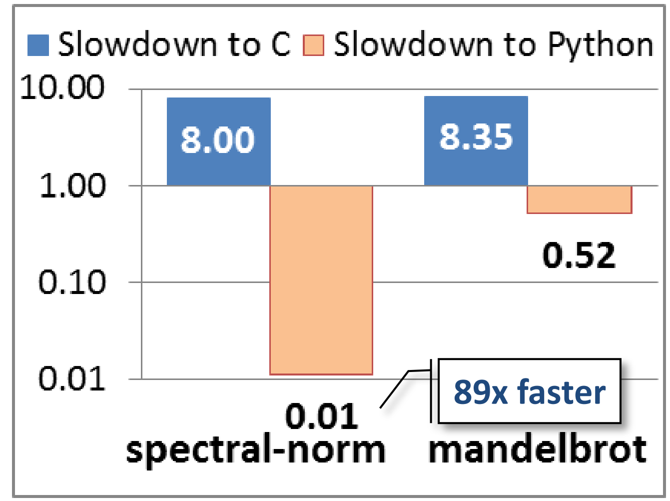
Data Object Specialization – Ideas

- Approach
  - Introduce new data representation besides the nodes and vector
  - Use them to express runtime objects, and some R data types
- Some candidates

Object	Current Representation	Possible Specialization
Local frames	Linked list, search by name	Stack, search by index, and a Map for the dynamic part
Argument list	Linked list	Slots in the stack
Hashmap	Constructed using Node object and Vector objects	A dedicated HashMap data structure
Attributes of a object	Linked list	using a hashmap,
Matrix, high dim arrays	Vector plus attributes lists	Dedicated objects based on Vector

Vectorization Background

- Observations: the performance of type II code is good
  - Two shootout benchmark examples
    - R: Using Type II coding style
    - C/Python: from shootout website
  - R is within 10x slowdown to C
  - R is faster, or much faster than Python



Type II with standard input size

- But
  - It’s relatively hard to write type II code
- ORBIT’s optimization

  - Vectorize one specific category application

apply Family of Operations

- A family of built-in functions in R

Name	Description
apply	Apply Functions Over Array Margins
by	Apply a Function to a Data Frame Split by Factors
eapply	Apply a Function Over Values in an Environment
lapply	Apply a Function over a List or Vector
mapply	Apply a Function to Multiple List or Vector Arguments
rapply	Recursively Apply a Function to a List
tapply	Apply a Function Over a Ragged Array

- Their behaviors – Similar to the ***Map*** function
  - Use ***lapply*** as the example
  - if $L = \{s_1, s_2, \ldots, s_n\}$, f is a function $r \leftarrow f(s)$, then
  - $\{f(s_1), f(s_2), \ldots, f(s_n)\} \leftarrow lapply(L, f)$

Performance Issues of ***apply*** Operations

- Interpreted as Type I style – Loop over data

pseudo code of ***lapply***

```
lapply(L, f) {
    len <- length(L)
    Lout <- alloc_veclist(len)
    for(i in 1:len) {
        item <- L[[i]]
        Lout[[i]] <- f(item)
    }
    return(Lout)
}
```

Implemented in C code to improve the performance

- Problems remaining
  - Interpretation overhead
    - Pick element one by one, and **invoke f()** many times.
  - Data representation overhead
    - *L* and *Lout* are represented as R list objects. Composed by R Node objects

A Motivating Example

- ***apply*** style *V.S.* Vector programming

```
# a<- rnorm(100000)
b <- lapply(a, function(x){x+1})
```

time = 2.013 s

```
# a<- rnorm(1000000)
b <- a + 1
```

time = 0.016 s

- Vectorization of apply based applications?

Linear Regression

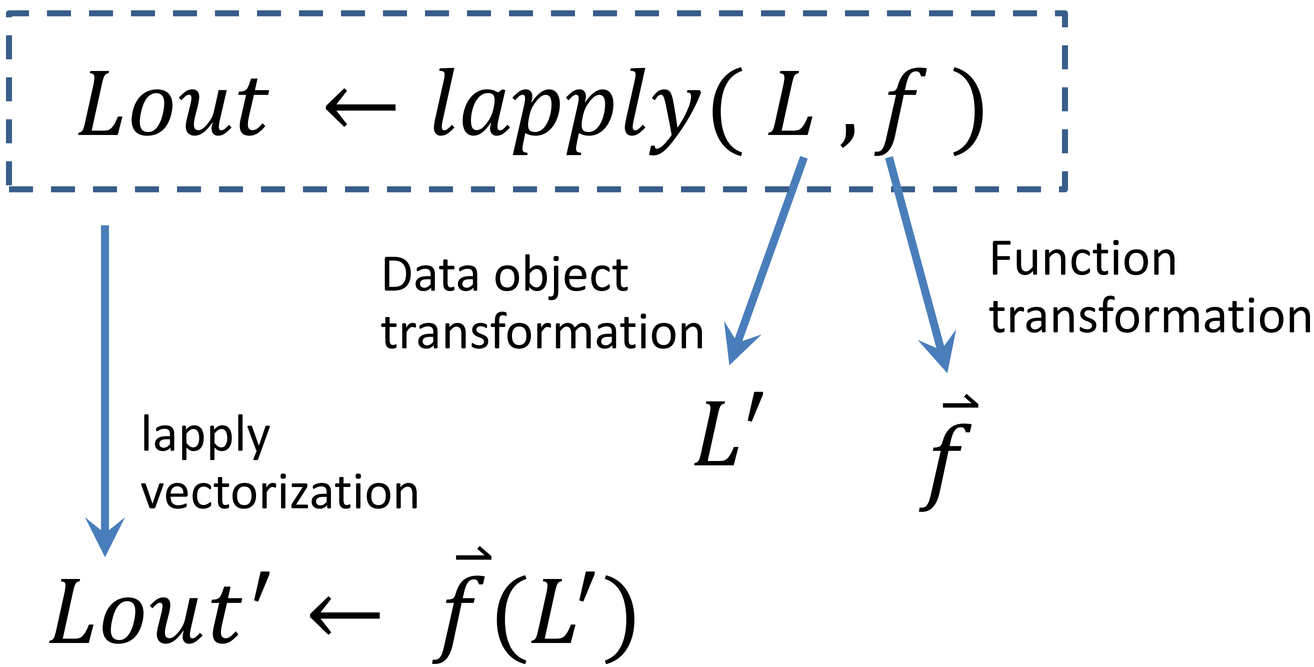
```
grad.func <- function(yx) {
   y <- yx[1]
   x <- c(1, yx[2])
   error <- sum(x *theta) - y
   delta <- error * x
}
```

```
delta <- lapply(sample.list,
                   gradfunc)
```

→ ***Vector version?***

Vectorization – High Level Idea

- Transform Type I interpretation to Type II/Type III execution



- L': The corresponding vector representation of L
- $\vec{f}$: The vector version of f , that can take a vector object as input

Some Preliminary Results of Vectorization

- Up to 27x, in average 9x speedup

Name	Original (s)	Vectorized (s)	Speedup
LR	25.227	1.576	16.01
LR-n	35.712	4.241	8.42
K-Means	15.646	2.776	5.63
K-Means-n	22.387	3.369	6.64
Pi	23.134	11.320	2.04
NN	24.690	0.893	27.65
kNN	26.477	1.687	15.69
Geo Mean			8.91

No data reuse, the overhead of data reshape cannot be amortized

- This Vectorization is orthogonal to the current R parallel frameworks

Conclusion

- Our Work – ORBIT VM
  - Extension to GNU R, Pure interpreter based JIT Engine
  - Specialization
    - Operation specialization + Object representation specialization
    - Some results were published in CGO 2014
  - Vectorization
    - Focusing on applications based on apply class operations
    - Transform Type I execution into Type II and Type III

- The benchmarks
  - https://github.com/rbenchmark/benchmarks
  - Benchmark collections
  - Benchmarking tools
    - A driver + several harness to control different research R VMs

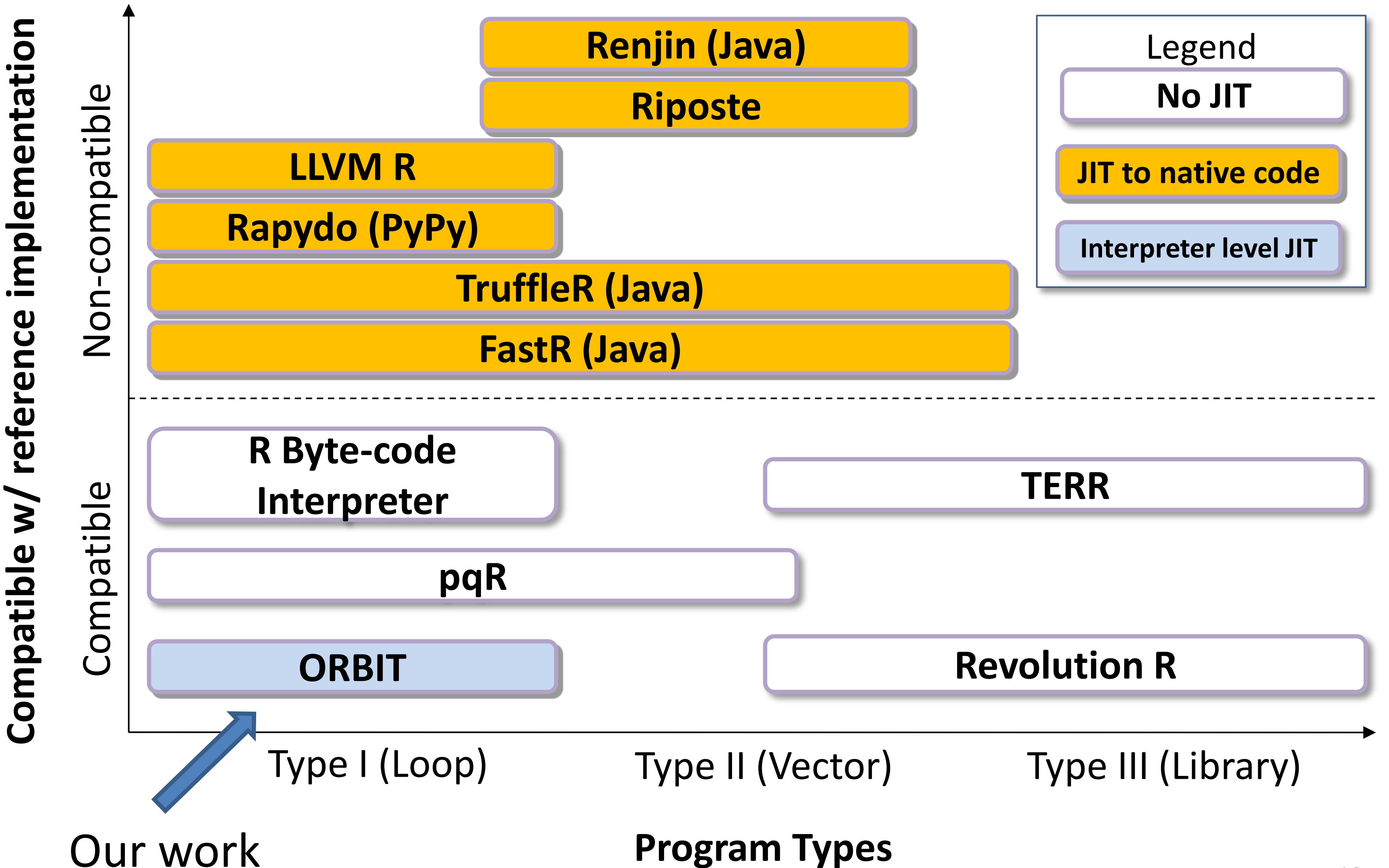
Thank You!

Contact Info:

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

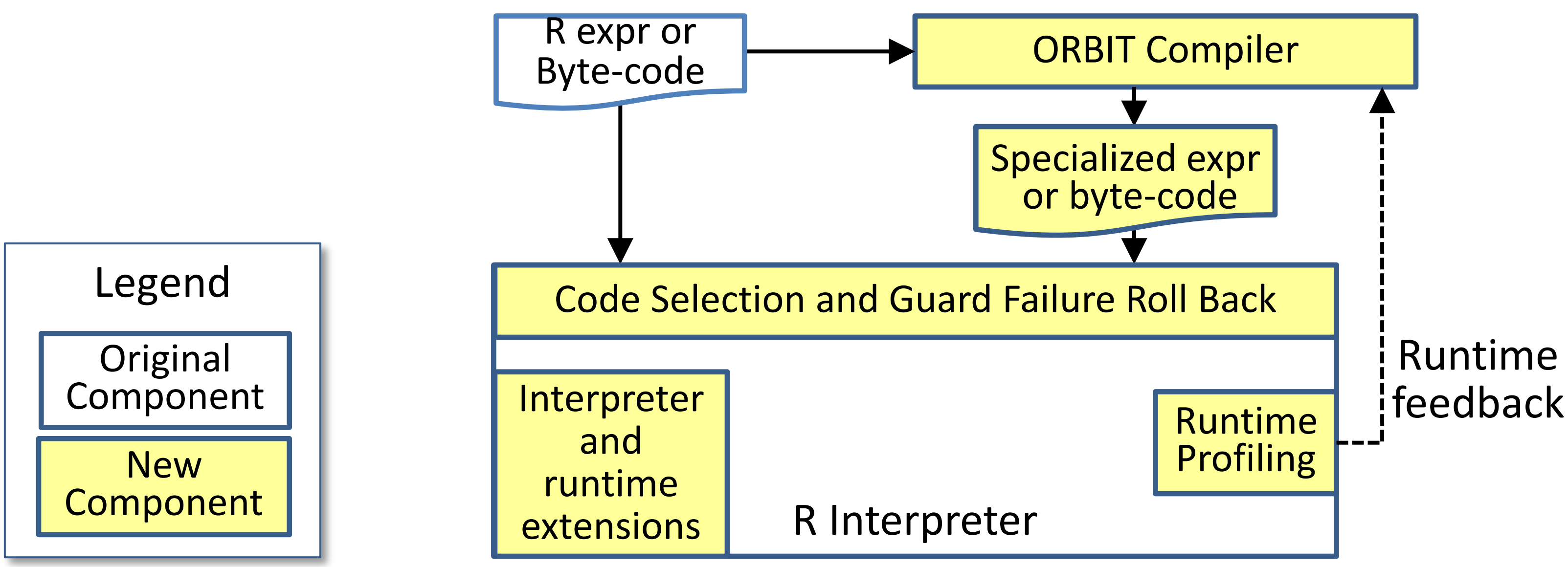
Backup

Related Work



ORBIT Project Overview

- Focus on Type I code's performance improvement
  - Specialization: operation and data object representation
  - Vectorization: translate Type I code into Type II code
- Pure Interpreter Approach
  - Portable, simple, and easy to be compatible with GNU R
- Compiler plus runtime
  - Use runtime information to guide compiler optimization



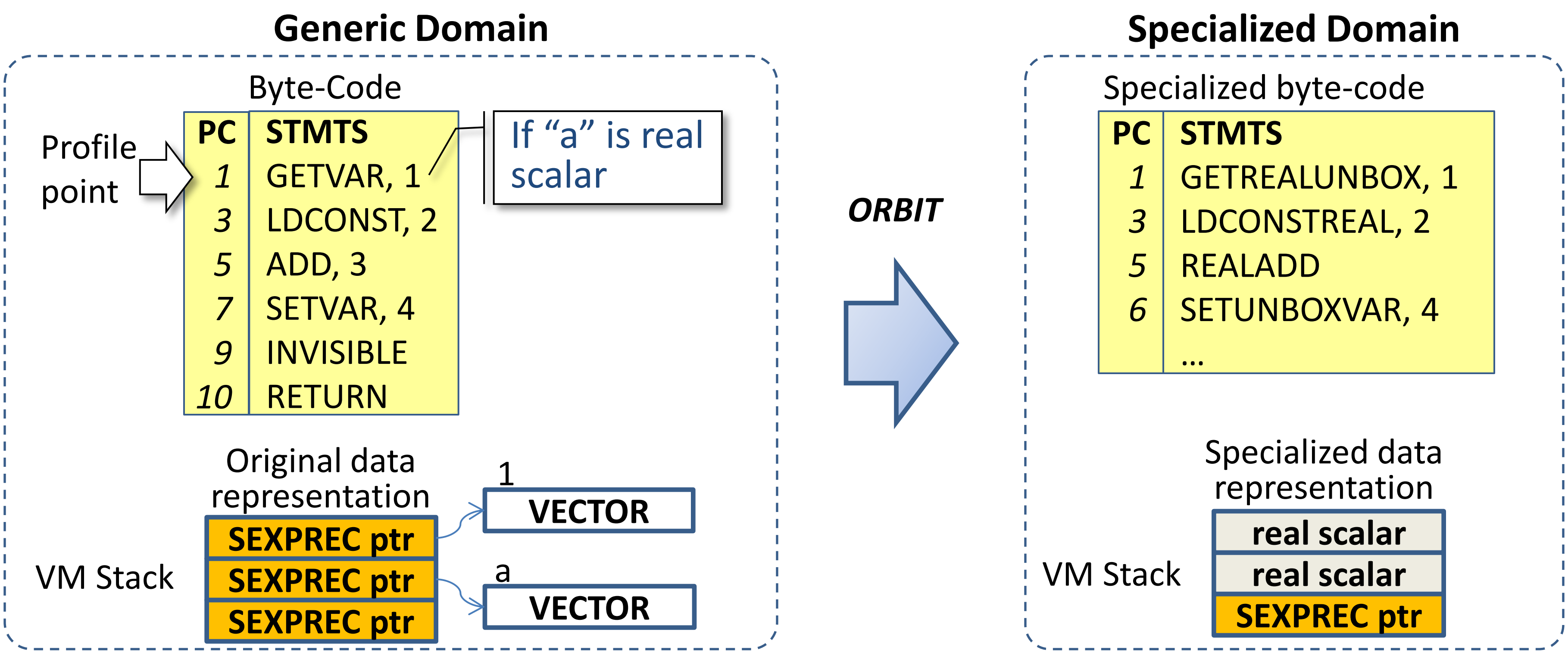
An Example of ORBIT Specialization

Source

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

Byte-code Symbol table

Idx	Value
1	"a"
2	1
3	a+1
4	b



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