Optimizing R VM: Interpreter-level Specialization and Vectorization

Haichuan Wang¹, Peng Wu², David Padua¹

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

- **ORBIT Specialization VM (CGO’14)**
  
  - **Type I (Loop)**
  
  - **Vectorization of apply family operations**
  
  - **Type II (Vector)**
  
  - **Type III (Library)**
  
- **R Benchmark Repository + Performance evaluation and analysis**
  (https://github.com/rbenchmark/benchmarks)
Optimizing R VM: Interpreter-level Specialization and Vectorization

**Specialization**

Source

\[ a + 1 \]

Byte-code

- GETVAR\_OP, 1
- LDCONST\_OP, 2
- ADD\_OP

**Operation Side**

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

**Data Object Side**

- Top → VM Stack → Specialization

1. VECTOR

```
Top: SEXPREC ptr
    SEXPREC ptr
    SEXPREC ptr
    ...
VM Stack: VECTOR
    a
```

```
Top: unboxed val
    unboxed val
    SEXPREC ptr
    ...
VM Stack: VECTOR
```

**Specialization**

- REALADD\_OP
- REALVECAD\_D_OP
- INTADD\_OP
- INTVECAD\_D_OP
- SCALADD\_OP
- VECADD\_OP
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**

```r
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**

<table>
<thead>
<tr>
<th></th>
<th>Byte-code Interpreter</th>
<th>ORBIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC Time (ms)</td>
<td>32.0</td>
<td>14.8</td>
</tr>
<tr>
<td>Node objs allocated</td>
<td>3,753,112</td>
<td>750,104</td>
</tr>
<tr>
<td>Vector scalar objs allocated</td>
<td>3,004,534</td>
<td>2,251,526</td>
</tr>
<tr>
<td>Vector non-scalar allocated</td>
<td>3,032</td>
<td>23</td>
</tr>
</tbody>
</table>
Optimizing R VM: Interpreter-level Specialization and Vectorization

Performance of ORBIT – Shootout Benchmark

- **Speedup over byte-code interpreter**
  - nbody: 4.29
  - fannkuch-redux: 5.90
  - spectral-norm: 2.24
  - mandelbrot: 5.05
  - pidigits: 6.28
  - binary-trees: 1.37
  - Geo Mean: 3.68

- **Percentage of Memory Allocation Reduced**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SEXPREC</th>
<th>VECTOR scalar</th>
<th>VECTOR non-scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbody</td>
<td>85.47%</td>
<td>86.82%</td>
<td>69.02%</td>
</tr>
<tr>
<td>fannkuch-redux</td>
<td>99.99%</td>
<td>99.30%</td>
<td>71.98%</td>
</tr>
<tr>
<td>spectral-norm</td>
<td>43.05%</td>
<td>91.46%</td>
<td>99.46%</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>99.95%</td>
<td>99.99%</td>
<td>99.99%</td>
</tr>
<tr>
<td>pidigits</td>
<td>96.89%</td>
<td>98.37%</td>
<td>95.13%</td>
</tr>
<tr>
<td>Binary-trees</td>
<td>36.32%</td>
<td>67.14%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mean</td>
<td>76.95%</td>
<td>90.51%</td>
<td>72.60%</td>
</tr>
</tbody>
</table>

Dominated by user level call overhead. Not handled by ORBIT.
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**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>Apply Functions Over Array Margins</td>
</tr>
<tr>
<td>by</td>
<td>Apply a Function to a Data Frame Split by Factors</td>
</tr>
<tr>
<td>eapply</td>
<td>Apply a Function Over Values in an Environment</td>
</tr>
<tr>
<td>lapply</td>
<td>Apply a Function over a List or Vector</td>
</tr>
<tr>
<td>mapply</td>
<td>Apply a Function to Multiple List or Vector Arguments</td>
</tr>
<tr>
<td>rapply</td>
<td>Recursively Apply a Function to a List</td>
</tr>
<tr>
<td>tapply</td>
<td>Apply a Function Over a Ragged Array</td>
</tr>
</tbody>
</table>

- 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

  ```r
  # 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?

  ```r
  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

\[ L_{out} \leftarrow \text{lapply}(L, f) \]

- \( L' \): The corresponding vector representation of \( L \)
- \( \hat{f} \): 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

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

- This Vectorization is orthogonal to the current R parallel frameworks

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

Program Types

- Type I (Loop)
- Type II (Vector)
- Type III (Library)

Compatible

- R Byte-code Interpreter
- pqR
- ORBIT
- Revolution R

Non-compatible

- LLVM R
- Rapydo (PyPy)
- Riposte
- TruffleR (Java)
- FastR (Java)
- Renjin (Java)

Legend

- No JIT
- JIT to native code
- Interpreter level JIT

Our work

Optimizing R VM: Interpreter-level Specialization and Vectorization
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

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

<table>
<thead>
<tr>
<th>Idx</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>a+1</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
</tbody>
</table>

Generic Domain

<table>
<thead>
<tr>
<th>PC</th>
<th>STMTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GETVAR, 1</td>
</tr>
<tr>
<td>3</td>
<td>LDCONST, 2</td>
</tr>
<tr>
<td>5</td>
<td>ADD, 3</td>
</tr>
<tr>
<td>7</td>
<td>SETVAR, 4</td>
</tr>
<tr>
<td>9</td>
<td>INVISIBLE</td>
</tr>
<tr>
<td>10</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

If "a" is real scalar

Specialized Domain

<table>
<thead>
<tr>
<th>PC</th>
<th>STMTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GETREALUNBOX, 1</td>
</tr>
<tr>
<td>3</td>
<td>LDCONSTREAL, 2</td>
</tr>
<tr>
<td>5</td>
<td>REALADD</td>
</tr>
<tr>
<td>6</td>
<td>SETUNBOXVAR, 4</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Original data representation

- SEXPREC ptr
- a

Specialized data representation

- real scalar
- real scalar
- SEXPREC ptr

Profile point

VM Stack

SEXPREC ptr

SEXPREC ptr

SEXPREC ptr

VM Stack
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