Improving R Interpreter's Performance Through Specialization

Haichuan Wang
University of Illinois at Urbana-Champaign
hwang154@illinois.edu

David Padua
University of Illinois at Urbana-Champaign
padua@illinois.edu

Peng Wu
Huawei America Lab
pengwu@acm.org

Abstract

R, a dynamic scripting language designed for statistical computing, has grown in popularity in recent years. However, the low performance of R, due to inefficiencies in the interpretation, limits its usability. Our previous study classifies R programs into three types, including Type I (looping over data), Type II (vector programming), and Type III (glue codes). The most serious performance problems of R are mostly manifested on Type I R codes.

We have proposed and implemented two approaches based on specialization to improve R's performance for Type I R codes. Firstly, ORBIT VM, an extension of the GNU R VM, to perform aggressive object allocation removal and instruction path length reduction in the GNU R VM via profile-driven specialization techniques. Secondly, the VALOR compiler, transforming one specific R Type I code into Type II to reduce the interpretation overhead. These two approaches improved the running speedup from 3x to 7x in different contexts.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Interpreters, Run-time environments

Keywords R, Specialization, Dynamic Scripting Language

1. Introduction

R is considered as the lingua franca for data analysis. However, like other dynamic scripting languages, R is very slow, which makes it difficult for R to process the truly BIG data. Earlier research revealed that GNU R VM, the most widely used R implementation today, can be hundreds of times slower than C [1]. Our previous study [2] classified R programs into three categories, Type I (looping over data), Type II (vector programming), and Type III (glue codes). We found out that the major performance problem only appears in Type I R code. Although real R applications are a mixture of all the three programming styles, due to the very slow performance of Type I codes, there is a significant amount of R scripts spend most of the time in the Type I portion.

There are many research projects trying to improve the performance of R. Figure 1 summarizes all the major projects through JIT compilation or VM-level optimizations. We classify the projects according to their target R programming styles (x-axis) and the compatibility with the GNU R VM (y-axis).

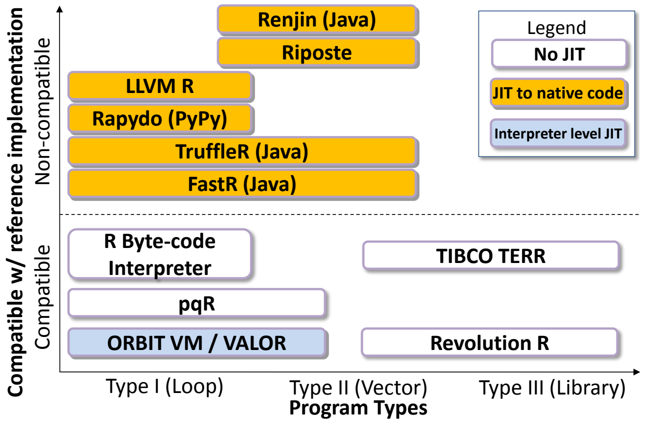


Figure 1: Landscape of R optimization projects.

The compatibility here means the R VM uses the same memory object representation as the GNU-R's. Many related work built brand new R VMs with their own object representations, which create a high barrier to adoption by the R user community. This is because the thousands of packages available from public R repositories, such as CRAN and Bioconductor, are the most valuable assets of R, and many of these packages depend on internals of the GNU R VM.

We aim at improving the performance of Type I codes, while maintaining the full compatibility with the GNU R VM. We do specialization JIT in the GNU-R interpreter level to maintain the original memory object representation. Our work includes two approaches

- **ORBIT**(**O**ptimized **R** **B**yte-code **I**nterpre**T**er). An extension of the GNU R VM, to perform aggressive removal of allocated objects and reduction of instruction path lengths in the GNU R VM via profile-driven specialization techniques. The ORBIT VM is fully compatible with the R language and is purely based on interpreted execution. It is a specialization JIT and runtime focusing on data representation specialization and operation specialization. For our benchmarks of Type I R codes, ORBIT is able to achieve an average of 3.5X speedups over the current release of GNU R VM

- **VALOR**(**V**ectorization of **A**pp**L**y operation for **O**verhead **R**eduction). A lightweight compiler package that reduces the interpretation overhead of R through the vectorization of the widely used `Apply` class of operations in R. Our approach combines data transformation and function vectorization to transform the looping-over-data execution into a code with mostly vector operations, which can significantly speedup the execution of `Apply` operations in R. The evaluation shows that the transformed code can achieve up to 22x speedup (and 7x on average) for a suite of data analysis benchmarks without any native code generation and still using only a single-thread of execution.

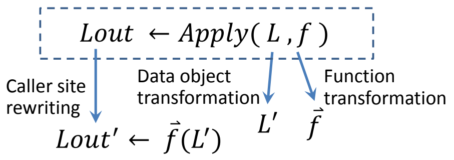
2. ORBIT VM

The inefficiency of R interpreter mainly comes from 1) The type generic interpretation, which always requires heavy dynamic type checking and dispatch, and 2) The heavy generic object representation of R, which allocates huge amount of small memory objects to represents R's data. We offered a new approach that combines JIT compilation and runtime techniques to tackle these problems:

- **Profile-directed specialization.** We instrumented a minimal set of byte-codes to get precise type info efficiently and use the profiled type to do a lightweight type inference to get all required type info.
- **Interpretation of optimized codes.** We translate type generic byte-codes into type specialized byte-codes, and still execute them in the extended byte-code interpreter. There is no native code-gen involved to achieve the lightweight goal.
- **Object allocation removal.** We dynamically translate the type generic object representation to type specialized object representation, and remove most of the small object allocated, which reduces a huge amount memory management overhead.

3. VALOR Compiler

VALOR compiler uses a lightweight approach that reduces the interpretation overhead of R through the vectorization of the widely used `Apply` class of operations. The standard implementation of `Apply` incurs in a large interpretation overhead resulting from iteratively applying the input function to each element of the input data. Our approach combines data transformation and function vectorization to transform the looping-over-data execution (Type I) into a code with mostly vector operations (Type II), as $Lout \leftarrow Apply(L, f) \Rightarrow Lout \leftarrow \vec{f}(L)$. And the transformation significantly reduces the interpretation overhead of `Apply` operations in R



$Lout \leftarrow Apply(L, f)$
Caller site rewriting
Data object transformation
Function transformation
$Lout' \leftarrow \vec{f}(L')$
L'
$\vec{f}$

Figure 2: Three Tasks in the Full Vectorization Algorithm

Figure 2 illustrates the three transformations in VALOR compiler. (1) *Data Object Transformation*, which permutes input data so that the vectorized function can get direct access to the data (in vector form and stored in consecutive space) if the input data is not already an array; (2) *Function Vectorization*, which generates the vector version of the single object function; (3) *Caller Site Rewriting*, which rewrites the `Apply` function call into a vector function invocations, and performs other optimizations to reduce the overhead.

4. Evaluation

Figure 3 shows ORBIT's speedup to R byte-code interprter on the *shootout* benchmark. It achieved 3.68x speedup in average. Figure 4 presents VALOR's speedup to the original `lapply` based R code on data analytics algorithms, and it achieved 7.3x geomean speedup.

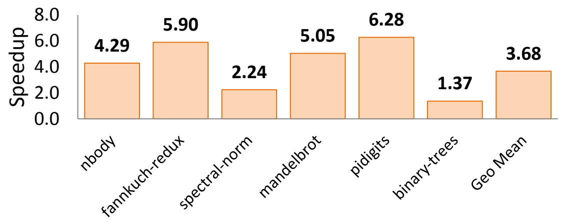


Figure 3: ORBIT's speedup to R byte-code interpreter

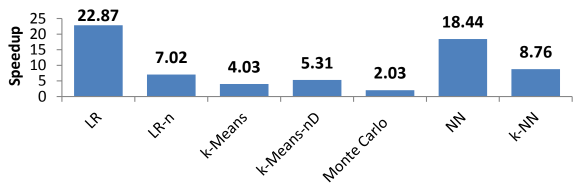


Figure 4: Speedup of VALOR to the original code

References

[1] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the r language: objects and functions for data analysis. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 104–131, 2012.

[2] H. Wang, P. Wu, and D. Padua. Optimizing r vm: Allocation removal and path length reduction via interpreter-level specialization. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 295:295–295:305, New York, NY, USA, 2014. ACM.