COMPILER AND RUNTIME TECHNIQUES FOR OPTIMIZING DYNAMIC SCRIPTING LANGUAGES

BY

HAICHUAN WANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor David A. Padua, Chair, Director of Research
Professor Vikram Adve
Professor Wen-Mei W. Hwu
Doctor Peng Wu, Huawei

Abstract

This thesis studies the compilation and runtime techniques to improve the performance of dynamic scripting languages using R programming language as a test case.

The R programming language is a convenient system for statistical computing. In this era of big data, R is becoming increasingly popular as a powerful data analytics tool. But the performance of R limits its usage in a broader context. The thesis introduces a classification of R programming styles into Looping over data(Type I), Vector programming(Type II), and Glue codes(Type III), and identified the most serious overhead of R is mostly manifested in Type I R codes. It proposes techniques to improve the performance R. First, it uses interpreter level specialization to do object allocation removal and path length reduction, and evaluates its effectiveness for GNU R VM. The approach uses profiling to translate R byte-code into a specialized byte-code to improve running speed, and uses data representation specialization to reduce the memory allocation and usage. Secondly, it proposes a lightweight approach that reduces the interpretation overhead of R through vectorization of the widely used `Apply` class of operations in R. The 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 without any native code generation and still using only a single-thread of execution. Thirdly, the `Apply` vectorization technique is integrated into SparkR, a widely used distributed R computing system, and has successfully improved its performance. Furthermore, an R benchmark suite has been developed. It includes a collection of different types of R applications, and a flexible benchmarking environment for conducting performance research for R. All these techniques could be applied to other dynamic scripting languages.

The techniques proposed in the thesis use a pure interpretation approach (the system based on the techniques does not generate native code) to improve the performance of R. The strategy has the advantage of maintaining the portability and compatibility of the VM, simplify the implementation. It is also a very interesting problem to see the potential of an interpreter.

Acknowledgments

Returning back to school to study for the Ph.D degree is a new adventure for me. This thesis dissertation marks the end of this long and eventful journey. I still clearly remember lots of sweat, pain, exciting and depressing moments in the last four years. I would like to acknowledge all the friends and family for their support along the way. This work would not have been possible without their support.

First, I would like to express my sincere gratitude to my advisor, Professor David Padua, and my co-advisor Doctor Peng Wu, for their guidance, encouragement, and immense knowledge. This thesis would certainly not have existed without their support. I also express my cordial thanks to Professor María Garzarán for her help and guidance in my research. I would also like to thank my thesis committee members Professor Vikram Adve and Professor Wen-Mei Hwu, for their insightful comments and suggestions.

I am very grateful to my fellow group members and other friends in Computer Science Department. It's my fortunate to study and research with them together.

I also thank Evelyn Duesterwald, Peter Sweeney, Olivier Tardieu, John Cohn and Professor Liming Zhang for their support in my Ph.D program. I would like to extend my thanks to Sherry Unkraut and Mary Beth Kelly for their help.

I thank my parents, who gave me the best education and built my personality.

Last but not least, special thanks to my lovely wife Bo, who inspired me and provided constant encouragement and support during the entire process.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AOT	Ahead of Time Compilation.
AST	Abstract Syntax Tree.
CFG	Control Flow Graph.
CRAN	the Comprehensive R Archive Network
DSL	Dynamic Scripting Language.
GPU	Graphics Processing Unit.
IR	Intermediate Representation.
JIT	Just-In-Time Compilation.
SIMD	Single Instruction, Multiple Data.
VM	Virtual Machine.

Chapter 1

Introduction

1.1 Overview

1.1.1 Dynamic Scripting Languages

Dynamic Scripting Languages are generally used to refer programming languages with features like dynamic evaluation (without Ahead-Of-Time compilation), high level abstraction and concepts, dynamic typing, and managed runtime. Examples of popular dynamic scripting languages include JavaScript, Python, PHP, Ruby, Matlab, R, etc.. These languages are becoming more important in the programming language spectrum recently years. About half of the top 10 most popular programming languages in both the TIOBE programming community index[30] and the IEEE Spectrum list[74] are dynamic scripting languages. Many dynamic scripting languages have been widely adopted including JavaScript for web client interface, PHP and Ruby for web front-end applications, Matlab for technical computing, R for statistic computing, and Python for a wide range of applications.

However, there is still one major problem that inhibits the pervasive usage of dynamic scripting languages, the relatively low performance compared with their static language counterpart. The shootout benchmark[4] report shows that an implementation of some common algorithms in dynamic scripting languages is typically over 10x slower than the implementation in static languages. There are mainly two reasons for the low performance. First, the *interpretation overhead* since most of the scripting languages are interpreted in a managed environment. Secondly, the *runtime overhead* to dynamically manage the runtime resources used in the dynamic execution.

Many approaches have been proposed to reduce the two kinds of overheads in the past decades, such as Just in Time Compilation (JIT) and Specialization. These techniques greatly improved the performance of dynamic scripting languages although there are still many unsolved problems.

1.1.2 R Programming Language

Thanks to the advent of the age of big data, R[14] has become a rising star among the popular dynamic scripting languages. R is a tremendously important language today. According to the NY times [81]:

> *R is used by a growing number of data analysts inside corporations and academia. It is becoming their lingua franca partly because data mining has entered a golden age, whether being used to set ad prices, find new drugs more quickly or fine-tune financial models. Companies as diverse as Google, Pfizer, Merck, Bank of America, the InterContinental Hotels Group and Shell use it.*

In other words, the reason for the growing importance of R is the emergence of big data. In the case of business, this emergence is leading to a revolution that shifts the focus of information processing from the back-office, where work was carried out by traditional data processing systems, to the front-office where marketing, analytics, and business intelligence [59] are having a growing impact on everyday operations.

One illustration of R's popularity is that most of the competitors on Kaggle.com, the No.1 online website in solving business challenges through predictive analytics, use R as their tools to solve the contest problems[7]. In fact, the number of R users in Kaggle.com is significantly higher than the number of users of other languages, such as Matlab, SAS, SPSS or Stata, (see Figure 1.1).

R can be considered as the lingua franca for data analysis today, and is therefore of great interest. Today, there are more than two million users of R, mainly from the field of data analysis, a scientific discipline and industry that is rapidly expanding. According to [73]:

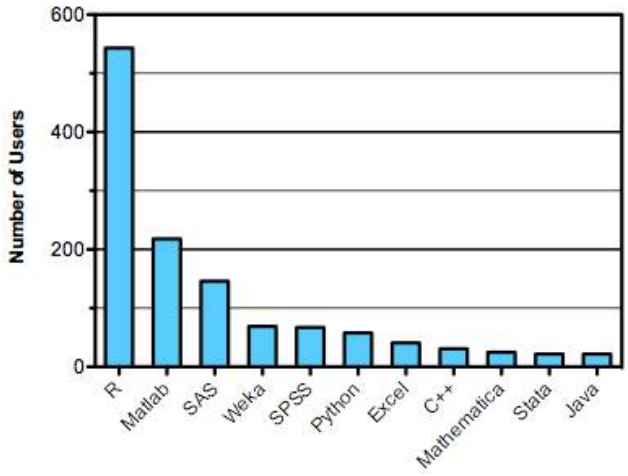


Figure 1.1: Software used in data analysis competitions in 2011[45]

> *Long gone are the days when R was regarded as a niche statistical computing tool mainly used for research and teaching. Today, R is viewed as a critical component of any data scientist's toolbox, and has been adopted as a layer of the analytics infrastructure at many organizations*

The popularity of R is mainly due to the productivity benefits it brings to data analysis. R contributes to programmer productivity in several ways, including the following two: the availability of extensive data analysis packages that can be easily incorporated into an R script and the interpreted environment that allows for interactive programming and easy debugging.

1.1.3 Performance Problems of R

The downside, and the justification for this thesis, is that R shares the limitations of many other interactive and dynamically typed languages: it has a slow implementation[1]. Unfortunately the performance of R is not properly understood. Some reported orders of magnitude slowdowns of R compared to other languages. Figure 1.2 compares the performance of a set of common

[1]R language has only one official implementation GNU R. The performance of R in this thesis refers the performance of GNU R

algorithms [4] implemented in different languages. It shows that for the implementations of those algorithms, R is more than two orders of magnitude slower than C and twenty times slower than Python (also an interpreted scripting language). Not only is R slow, it also consumes a significant amount of memory. All user data in R and most internal data structures used by the R runtime are heap allocated and garbage collected. As reported in [61], R allocates several orders of magnitude more data than C. Memory consumption is both a performance problem (as memory management and data copying contribute to the runtime overhead) and a functional problem (as it limits the amount of data that can be processed in memory by R codes). To cope with the performance and memory issues of R, it is a common practice in the industry for data scientists to develop initial data models in R then have software developers convert R codes into Java or C++ for production runs.

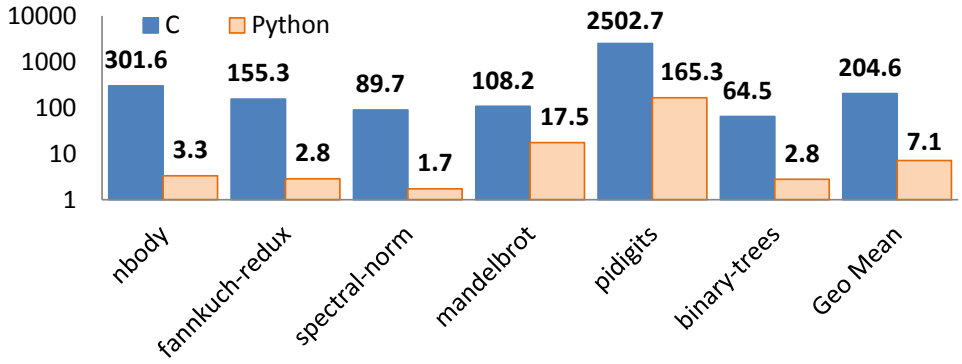


Figure 1.2: Slowdown of R on the `shootout` benchmarks relative to C and CPython.

In contrast, some users claimed that their R codes run as fast as any native code and are indeed used in production. Both claims are partially true since the performance or R codes depends on how the R codes are written.

R programs can be classified into three categories[84], Type I (looping over data), Type II (vector programming), and Type III (glue codes). The evaluation shows that the significant performance problems only appear in Type I R codes. The performance gap between R and C/Python showed in Figure 1.2 are the result of the Type I programming. Type II codes are much more efficient compared with Type I. These codes mainly suffer the performance problems of losing data

locality due to long vector computation. Finally, Type III codes' performance is purely dependent on the back-end implementation (in C and FORTRAN) of the R library, and it belongs to the static language domain.

While many R users are forced to use the Type III style in production codes due to the heavy overhead of Type I and II codes, according to [61], a significant portion of R codes still spend most of the execution time in Type I and II codes. And it is a common practice that Type I and Type II R codes are rewritten using other languages for production environment due to the slowness and memory consumption of R codes. Clearly, there are great benefits in having a highly efficient implementation of R for Type I and II codes so that R can be a productive programming environment where glue codes and main computation can be implemented in a uniform fashion.

1.2 Contributions

This thesis studies the compilation and runtime techniques to improve the performance of dynamic scripting languages. It makes the following contributions to tackle the poor performance problems of R:

- **Allocation removal and path length reduction via interpreter-level specialization**

  This thesis describes an optimized R byte-code interpreter, named ORBIT (**O**ptimized **R** **B**yte-code **I**nterpre**T**er). ORBIT is an extension to the GNU R VM. It performs aggressive removal of object 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 uses a specialization JIT and runtime that focus on data representation specialization and operation specialization. For the benchmarks of Type I R codes, the current ORBIT is able to achieve an average of 3.5X speedups over the GNU R VM and outperforms on the average most other R optimization projects that are currently available.

- **Vectorization of R Apply operations to reduce interpretation overhead**

  This thesis presents a lightweight approach that reduces the interpretation overhead of R through the vectorization of the widely used `Apply` class of operations in R. The normal implementation of `Apply` incurs in a large interpretation overhead resulting from iteratively applying the input function to each element of the input data. The proposed 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 vectorization transformation has been implemented as an R package that can be invoked by the standard GNU R interpreter. As such, the vectorization transformation does not require any modification to the R interpreter itself. The package automatically vectorizes the `Apply` class of operations. The evaluation shows that the transformed code can achieve up to 22x speedup (and 7x on average) for a suite of data analytics benchmarks without any native code generation and still using only a single-thread of execution.

- **Improving distributed R computing systems with operation vectorization**

  New compiler transformations are presented that improves the performance of SparkR[82], by extending the `Apply` vectorization technique just discussed into a two-level compiler transformation. The outer level is used to distribute data and computation while the inner level to reduce interpreter overhead. This transformation integrated with the SparkR interfaces. A two-level *Reduce* schema is also used to get the correct result. With these transformations, the SparkR's performance is doubled in the benchmark composed by a bunch of distributed algorithms.

- **Portable R benchmark suite and benchmarking environment**

  A suite of micro-benchmarks was developed as well as an R benchmarking environment. One challenge we face in the study of strategies for improving R performance is the lack

of a proper performance benchmarks for R. Another problem is that different R language implementations have different interfaces which causes the comparison of different R optimization techniques hard to conduct. Based on the classification R programs into Type I, II, and III styles, a collection of different types of R applications was developed and standardized. This collection is expected to facilitate researchers in this domain to locate the proper optimization targets. Furthermore, a flexible R benchmarking environment is provided to enable different R implementations to be measured with the benchmarks in the same manner. The benchmarking environment can be extended by plugging in additional meters to measure performance metrics such as Operation System data and hardware performance counters. The benchmark suite and benchmarking environment can help researchers in this domain easily get deep understanding of R's behaviors.

1.3 Thesis Organization

The remainder of the thesis is structured into chapters as follows:

- **Chapter 2**: Discusses the high-level background of the implementation and common optimizations for dynamic scripting languages. Then it describes the GNU R Virtual Machine (VM) structure, analyzes the performance issues for better understanding the approaches proposed in this thesis.

- **Chapter 3**: Describes the new interpreter level specialization techniques, including bytecode specialization and object representation specialization, and the implementation of these techniques in ORBIT VM.

- **Chapter 4**: Introduces the vectorization transformation of `Apply` operations in R, including the transformation framework, operation and data transformation algorithms, and the realization of the transformation in an R library.

- **Chapter 5**: Describes the compiler transformations and runtime optimizations used in applying vectorization into the SparkR distributed computing system.
- **Chapter 6**: Introduces the R benchmark suites as well as the R benchmarking environment.
- **Chapter 7**: Discusses the related work in dynamic scripting language domain and optimization work in R language area.
- **Chapter 8**: Concludes the thesis and proposes future work.

Chapter 2

Background

This chapter discusses the topics that help understand the remainder of the thesis. First, it describes the high level components that are required in implementing a dynamic scripting language. The typical performance problems and the common optimization techniques are introduced then. Secondly it gives a brief introduction of the R language, and explains the the GNU R Virtual Machine (VM) structure, analyzes the performance issues, and the optimizations used in GNU R VM.

2.1 Dynamic Scripting Language

2.1.1 Implementation of a Dynamic Scripting Language

Like static languages such as C and Java, the basic structure of implementation a dynamic scripting language (DSL) is similar. The source code is translated into the executable control sequences that the underlying hardware can understand and execute. This requires the compiler path (lexer, parser, optimizer, etc.). A dynamic language also requires object descriptors to enable the identification of different types of objects. Furthermore, the language implementation needs other components to handle tasks such as code loading, interface with other languages, etc..

Furthermore, DSLs have many unique language features, which require special implementations. The following section describes a few common components used in realizing a DSL.

Interpreter In many usage scenarios, the source code of a DSL program is input dynamically, which means an Ahead-Of-Time (AOT) compilation cannot be conducted. An interpreter is commonly used to handle this situation. The interpreter accepts the source code's representation in the

form of an Abstract Syntax Tree (AST) or byte-code program, and evaluates it.

There are two types of interpreters, AST interpreters and byte-code interpreters. An AST interpreter will traverse the AST top-down to trigger the evaluation of the current node's child nodes, and merge the result bottom-up to get the final result. Like static languages, the AST of a DSL can also be translated into a sequence of instructions, called byte-code. However, these byte-code program will not be executed by the bare hardware processor, but by a software implemented virtual machine(VM). The VM has implements the logic of each byte-code instruction, and it interprets the byte-code one instruction at a time, modifies the VM runtime environment, and finally gets the result in the memory or the VM stack.

Generic Object Representation Similar to static languages, dynamic scripting languages typically have many different types, such as boolean, integer, string, etc. However, many of these languages have the feature that only values carrie the type, not the variable, which means a variable can be bound to different values with different types at different time. Because the type of a variable cannot be decided statically, a common implementation in dynamic scripting languages is to use generic representation to describe a value object (typically a class with one field to describe the type, and one field to store the real raw data). Because the size of the generic representation cannot be decided statically either, objects are dynamically allocated in the heap by most implementations.

Furthermore, many dynamic languages contain extensible data types, for example a value in JavaScript can contain arbitrary numbers of fields, and a value in R can be set with arbitrary numbers of attributes. In order to represent these types, a map or a linked list structure is commonly used to implement these types.

Dynamic Dispatch Because the variable in dynamic languages does not carry type, the operations on the variable cannot be decided statically. For example consider an *add* operation in the JavaScript expression `a+b`. If `a` and `b` are both numbers, the addition will do number addition

(64bit floating point addition). But if `a` and `b` are all strings, the addition will do string concatenation. And if `a` is a string while `b` is a number, the addition operation will first promote the number into a string, then do string concatenation. The operation of the addition is dynamically decided according to the types of the input operands. The mechanism is similar to the implementation of polymorphic operation in object-oriented programming, and is also called dynamic dispatch here.

Dynamic Variable Lookup A variable in a DSL may be or not be defined at a specific point in the source code. So a common practice in the implementation is not to have static binding for a variable. As a result, unlike static languages, where a variable can be located through a fixed index offset, there is no fixed index to locate a variable in the corresponding context frame (a method's local frame or the global frame). The common implementation is using the variable's name to do a runtime lookup, and find the current bound location to get the value.

Dynamic Memory Management and Garbage Collection Because most of the objects and resources are allocated dynamically, there are far fewer memory regions allocated statically in a DSL. Most of the memory allocation requests are resolved as a heap allocation, and require the language's runtime to handle the allocation and de-allocation. In order to reduce the burden of the programmer, Garbage Collection(GC) is typically used by a DSL, although this is not unique to DSLs. Many static languages also have Garbage Collection, such as Java.

2.1.2 Performance Problems of Dynamic Scripting Languages

As described in Chapter 1, dynamic scripting languages have the benefit of providing interactive programming environment, high level abstraction and high productivity. The major limitation for pervasive of these languages is the low performance. According the data reported in the shootout benchmark website[4], the speed performance gap between dynamic scripting languages and static languages is typically 10x or higher. This is mainly due to their implementation as described in Section 2.1.1.

Interpretation Overhead The interpreter is indeed a software simulated processor to execute the language's instruction sequence. The byte-code interpreter is very similar to the hardware processor, which performs the tasks of fetching a byte-code, decoding the byte-code, loading operands, performing the computation and storing the result back. Each step requires the execution of many real native instructions, which are all overhead compared with a pre-compiled native binary from a static language. The AST interpreter has even higher overhead since it has to traverse the AST through all kinds of pointers. This kind of traversing is tedious and lack of locality.

Memory Management Overhead Because of the generic object representation, most of the object types are expressed as a Boxed object. For example, a primitive integer is not just a 32bit memory cell in the stack or heap, but a class object that contains the header (type and size attributes) and the raw data. An additional pointer is required to refer the class object, and the object is commonly allocated in the heap. All of these incur memory usage overhead as well as the need to execute additional instructions to traverse the pointers and get the real type and value.

Dynamic Dispatch Overhead Due to the dynamic dispatch attribute, the logic to implement the dynamic dispatch either contains a large control flow to check the types and invoke the corresponding routines, or a dispatch table lookup that still requires a table lookup and execution of comparison operations. This not only increase the native instruction path length but also slow down the hardware processor's pipeline due to the complex control flows.

Variable Lookup Overhead Because there is no fixed index for a variable in a frame, a variable lookup requires a dynamic search in the runtime environment. The lookup operation is implemented either a hashmap search or a linear search. In either way, a string comparison is required, which has to use many native instructions to get the result. Considering variable read and write are the most frequent operations in the program, the overhead is very large.

Memory Management Overhead This overhead comes from the heap allocation of nearly all the dynamic resources as well as the garbage collection time for them.

2.1.3 Optimization Techniques for Dynamic Scripting Languages

There are many optimization techniques for dynamic scripting languages, starting from the early research on Smalltalk[40] and SELF [33][32]. The most common used approaches in the modern dynamic scripting languages are briefly described here.

Byte-Code Interpreter The easiest to build are the AST interpreters. However, due to the reasons explained in the previous section, AST interpreters have very poor performance. One common optimization is to translate the program AST into a byte-code sequence, and let it be executed in a byte-code interpreter. The interpretation logic of each byte-code instruction is relatively simple and can be implemented efficiently. Furthermore the linear execution of the byte-code sequence has better locality compared with the tree traversal in the AST interpreter. As a result, a byte-code interpreter is typically much faster than an AST interpreter.

Threaded Code A simple byte-code interpreter uses a `switch` statement to do the byte-code dispatch. This approach requires a large dispatch table and numerous comparisons, which cause poor hardware processor pipeline performance. Threaded code[26] is used to optimize the dispatch. The basic idea is that each byte-code instruction has a field to store the interpretation code's address for the next byte-code instruction. After interpreting the current instruction, there is no need to jump back to the big switch to do the dispatch, but directly jump to the next byte-code instruction's interpretation logic. Depends on where the next byte-code's address is stored, where the operands of an instruction are stored, and how the byte-code execution logic is invoked, threaded code can be implemented as direct threading, in-direct threading, subroutine threading, token threading, etc..

Specialization Because of the dynamic type feature, the execution logic of one operation in a dynamic scripting language typically consists all the combinations of routines to process different types. The implementation has many type checks, branches, and unbox routines (to handle boxed object representations). *Specialization* is a general term to describe the approach that capture the behaviors of the real execution for a particular operand type combination with a special routine for each combination. The special routine have much simpler logic, less checks. On the negative side, they require a predicate to guard that the operands have the expected type. For example, if both `a` and `b` in the `a+b` expression of Section 2.1.1 are integers, a special routine to do integer addition is provided for better performance. Specialization is commonly based on *types* and contexts, where the types could be variable types or function types. As a result, specialization typically requires type analysis or type inference. The implementation of specialization may contain operation side specialization (generating different instruction sequences), memory side specialization (define special object representations).

Inline Cache Inline Cache was introduced to optimize the polymorphic procedures invocations in Smalltalk[37]. It is also used to handle the dynamic dispatch in dynamic languages in a similar fashion[48]. It is based on the observation that the object type for a dynamic dispatch is relatively stable at a specific instruction location. So a cache could be allocated there to store last time's invocation target. In the next execution at the same location, if the check proves the type is not changed, the execution can directly invoke last time's target without the expensive dynamic lookup. Besides the dynamic dispatch, inline cache can also be used to accelerate the access of a object field in a dynamic shaped object. For example, accessing the field `a` of the variable `var1` through `var1.b`. The simple implementation of object `var1` is a table or a map. Every field access requires a heavy name based lookup, comparing each entry in the table or map. If it finds one entry's name is `b`, the value of that entry is returned. With the hidden class optimization support, the field access can be transformed as a integer offset based lookup, and use inline cache to cache last time's offset.

Hidden Class Hidden Class was firstly used in SELF implementation[33] to accelerate the dynamic object's field access. If the language has the feature that one object can contain an arbitrary number of fields, the simple implementation is to use a table or a map to represent the object. Then the accessing a field would be is slow due to the need of table/map entry lookup. The fact that a piece of code has a relatively stable field access that enable the optimization described next. Consider a function with input argument `arg`, in which `arg.a` field is accessed first, and then the `arg.b`. If there is a class (like a class in a static language) to define the type of the variable `arg`, the field access can be changed to a fixed index lookup in the class instance. Hidden class approach tries to abstract the class structure from the execution sequences. During the execution, a tree of classes are generated. The parent to children linkage is typically based on the field access sequence. Then the most precise class to describe a object is found, and the field access can be accessed using a fixed offset in the class structure, and can be further optimized with the combination of Inline Cache.

JIT Just-In-Time compilation is commonly used in the optimization of dynamic scripting languages. Based on the feedback from the interpreter, for example the code execution frequency, the type information traced or profiled, the code is then passed into a runtime compiler. The compiler applies code optimizations and generates more efficient code sequences for the next time's execution. The more efficient code sequence could be but not limited to native instructions. Other kinds of representation is possible, for example, JIT AST code into byte-code. Depends on the scope of the compiler transformation, JIT can be implemented as trace JIT (linear sequence of instructions, across method boundary), method level JIT, code block JIT (for example, only a loop region).

Native Code Generation Native code generation is also called machine code generation. It translates the code executed in the interpreter to native code sequence and runs it directly on the bare processor. It removes the interpretation overhead. However, if other optimizations are not applied during the native code generation, the runtime overhead is still there. Typically, native

code generation is the last step of dynamic scripting language optimization. It has the advantage of removing the interpretation overhead, but also has the limitation of architecture dependent.

2.2 R Programming Language

R language belongs to the dynamic scripting language domain. It is based on the S language [34] from Bell Labs, and originally developed by Ross Ihaka and Robert Gentleman, where the name *R* comes from. It is now developed and maintained by a small group of about twenty people around the world, which is called R core team. It is traditionally used in statistics domain, and has been becoming very popular recently years in data science and other domains.

[13] has describes the language in detail, and [61] gave a detail evaluation of the language from a programming language design perspective. This section only describes some important features and implementations of R regarding the thesis work.

2.2.1 Taxonomy of Different R Programs

R is not only a dynamic scripting language, but also a vector language. It has built-in vector, matrix and high-dimension array support. It also has many native languages (C and FORTRAN) implemented libraries wrapped as packages. Based on how programmer write R programs, there are three distinct R programming styles. Figure 2.1 shows the examples. Each exhibits a different performance trait and requires drastically different approaches to performance optimization:

- **Type I: Looping over data.** This programming style, as shown in Listing 2.1, is the most natural programming style to most users but is also the worst performing among the three programming styles. This comes mainly from the overhead of operating on individual vector elements in R. The several orders of magnitude of performance gap shown in Figure 1.2 ensues from the fact that the codes in that figure are all of Type I.

```
# ATT bench: creation of Toeplitz matrix
for (j in 1:500) {
  for (k in 1:500) {
    jk<-j - k;
    b[k,j] <- abs(jk) + 1
  }
}
```

Listing 2.1: Type I: Looping Over Data

```
# Riposte bench: age and gender are long vectors
males_over_40 <- function(age, gender) {
  age >= 40 & gender == 1
}
```

Listing 2.2: Type II: Vector programming

```
# ATT bench: FFT over 2 Million random values
a <- rnorm(2000000);
b <- fft(a)
```

Listing 2.3: Type III: Glue codes

Figure 2.1: Three different R programming styles.

- **Type II: Vector programming.** In this programming style one writes R codes using vector operations as shown in Listing 2.2. When operating mainly on short vectors, Type II codes suffer similar performance problems as Type I codes (henceforce, Type I also refers short vector Type II). When applied to long vectors, Type II codes are much better performing, but may suffer from poor memory subsystem performance due to loss of temporal locality in a long vector programming style.

- **Type III: Glue codes.** In this case R is used as a glue to call different native implemented libraries. List 2.3 shows such an example, where `rnorm()` and `fft()` are native library routines. The performance of Type III codes depends largely on the quality of native library implementations.

2.2.2 GNU R Implementation

There is only one official implementation of R language, the CRAN R project [14], and the R language itself is defined by this implementation. A brief introduction of GNU R virtual machine with its object representation is described below.

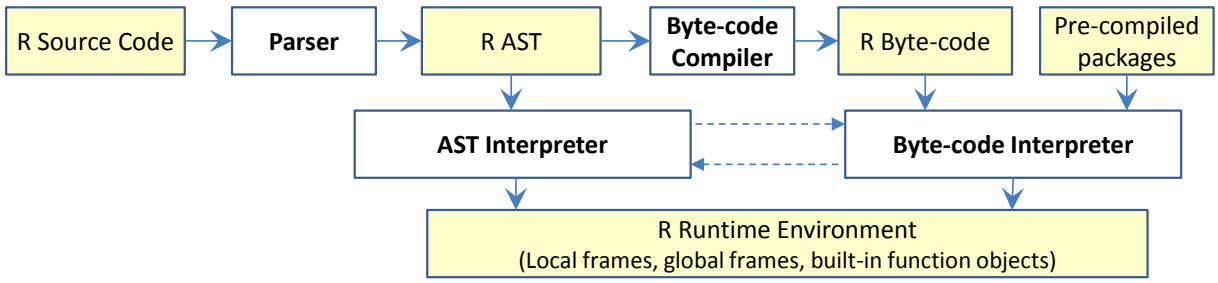


Figure 2.2: The GNU R VM.

Figure 2.2 depicts the structure of the GNU R VM since version R-2.14.0 that includes a parser, two interpreters, and a runtime system that implements the object model, basic operations, and memory management of R.

AST Interpreter and Byte-code Interpreter

The *R AST interpreter* is the default interpreter used by the GNU R VM. This interpreter operates on the Abstract Syntax Tree (AST) form of the input R program and is very slow. Figure 2.3 shows the R AST interpreter's performance comparing to C and CPython. The slowdown is even larger than the number shown in Figure 1.2, where the R byte-code interpreter is used. Since R version 2.14.0, a stack-based *R byte-code interpreter* [77] was introduced as a better performing alternative to the AST interpreter. To enable the byte-code interpreter, the user has to explicitly invoke an interface to compile a region of R code into byte-codes for execution or set an environment variable to achieve the same goal. Since the two interpreters use the same object model and share most of the R runtime, it is possible to switch between the two interpreters at well-defined boundaries such as function calls.

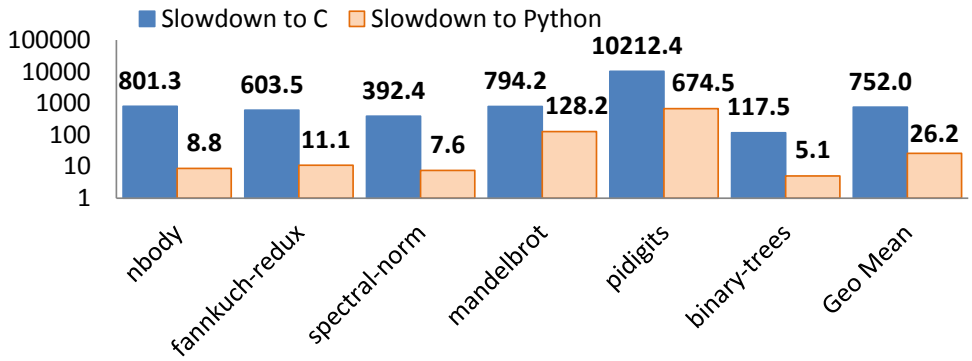


Figure 2.3: Slowdown of R AST Interpreter of the `shootout` benchmarks relative to C and CPython.

The byte-code interpreter has a simple ahead-of-time (AOT) compiler that translates ASTs generated by the parser to byte-codes. For each function, the R byte-code compiler produces two components, the symbol table and the byte-code. The symbol table records all the variable names, constant values and expressions in the source code. The byte-code instruction uses the symbol table to look for a value. Figure 2.4 shows an example byte-code sequence and the corresponding symbol table. The detail R byte-code compiler and byte-code format can be found at [78]. The AOT compiler also performs: simple peephole optimizations, inlining of internal functions, faster local variable lookup based on predetermined integer offset, and specialization of scalar math expressions. For Type I codes, the byte-code interpreter is several times faster than the AST interpreter. Starting from R-2.14.0, many basic R packages are compiled into byte-codes during installation and executed in the byte-code interpreter.

However, the compilation into byte-code is purely ahead of time, all the byte-codes are type generic, which still requires dynamic type checking and dispatching. Furthermore, the byte-code interpreter does not change the GNU R runtime implementation, and it still suffers the same issues from the runtime as the AST interpreter.

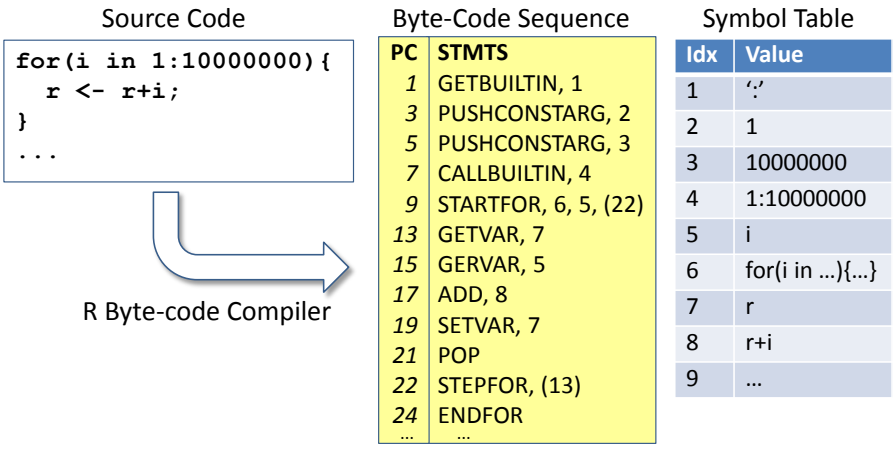


Figure 2.4: R byte-code and symbol table representation.

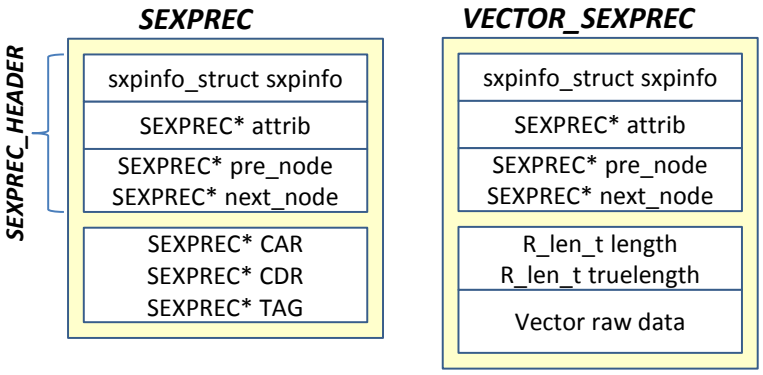


Figure 2.5: Internal representation of R objects.

R Object Model

The GNU R VM defines two basic meta object representations: *SEXPREC* (henceforth referred to as *Node* object for short) and *VECTOR_SEXPREC* (henceforth referred to as *VECTOR* or *Vector* object for short). As shown in Figure 2.5, an object has an object header (*SEXPREC_HEADER*) and a body. The object header is same for both *SEXPREC* and *VECTOR*. The header contains three pieces of information:

- *sxpinfo* that encodes the meta data of an object such as data type and object reference count
- *attrib* that records object attributes as a linked list

- *prev_node* and *next_node* that link all R objects for the garbage collector

The *VECTOR* data structure is used to represent vector and matrix objects in R. The body of the *VECTOR* records vector length information and the data stored in the vector. Scalar values are represented as vectors of length one.

The *SEXPREC* data structure is used to represent all R data types not represented by VECTOR such as linked-list and internal R VM data structures such as the local frame. The body of *SEXPREC* contains three pointers to *SEXPREC* or *VECTOR* objects: *CAR*, CDR, and TAG. Using the three pointers, a linked-list can be easily implemented by *SEXPREC* in a LISP style.

Figure 2.6 shows a local frame represented as a linked list of entries where each entry contains pointers to a local variable name, the object assigned to the local variable, and the next entry in the the linked list. And an environment is composed by several local frames, where in each frame's head node, one pointer points to the local frame linked list, one points to its parent frame, and the last one points to a hash table for fast object lookup.

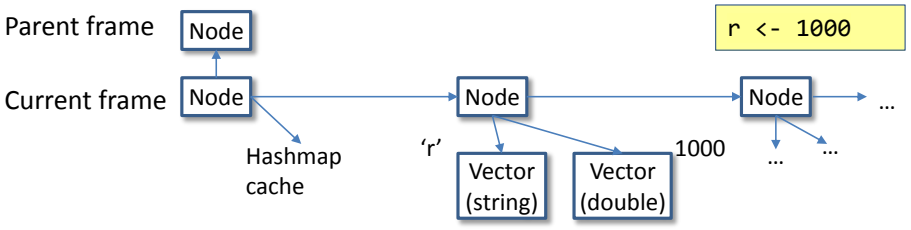


Figure 2.6: Local Frame Structure

Figure 2.7 presents the structure a matrix is expressed with the *Node* object and *Vector* object. A matrix uses a *VECTOR* object as the base, and the `attrib` field in the base object's header points to a linked list, where a *dim* attribute binding is defined. The name of the binding is a length-three string vector ("dim"), and the value of the binding is a length-two integer vector ([3,4]), which is used to define the first dimension and the second dimension sizes of the matrix.

Although the LISP like data structure is flexible and can represent arbitrary R objects, it also causes serious performance issues. Thus, traversing the linked structure requires the execution of numerous instructions. And the big header consumes much space even for simple type objects,

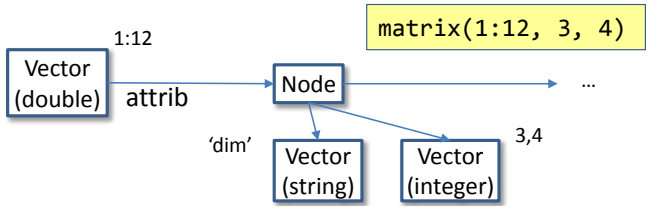


Figure 2.7: Matrix Structure

such as simple scalars and vectors. Furthermore, the *SEXPREC* node object are heavily used everywhere as the construction units for all other data type objects. As a result, the whole heap of R runtime is composed by huge amount of small *SEXPREC* and *VECTOR* objects.

Memory Management

Memory Allocator The memory allocator of R VM pre-allocates pages of *SEXPREC*. A request is satisfied just by getting one free node from a page. The memory allocator also pre-allocates some small *VECTOR* objects in different page sizes to satisfy requests for small vectors. A large vector allocation request is performed through the system malloc.

Garbage Collector R VM does automatic garbage collection (GC) with a stop-world multi-generation based collector. The mark phase traverses all the objects through the link pointers in the object headers. Dead objects are then compacted to free pages. Dead large vectors are freed and returned to the operating system.

Copy-on-write Every named object in R is a value object (i.e., immutable). If a variable is assigned to another variable, the behavior specified by the semantics of R is that the value of one variable is copied and this copy is used as the value of the other variable. R implemented copy-on-write to reduce the number of copy operations, Figure 2.8. There is a *named* tag in the object header, with three possible values: 0, 1, and 2. Values 0 and 1 mean that only one variable points to the object (value 1 is used to handle a special intermediate state[1]). By default the *named* value

[1] http://cran.r-project.org/doc/manuals/R-ints.html

is 0. When the variable is assigned to another variable, which means more than one variable point to the same underlying object, the object's *named* tag is changed to 2.

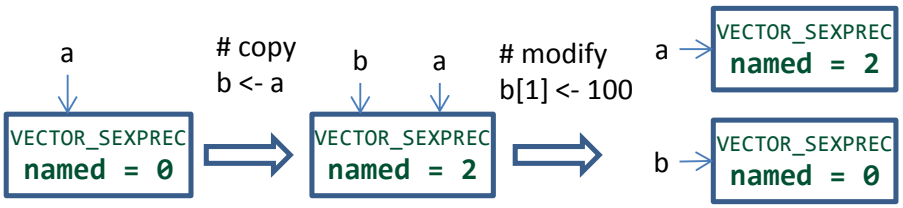


Figure 2.8: R Copy-on-Write Mechanism.

When an object is to be modified, the *named* tag is consulted. If the value is 2, the runtime first copies the object, and then modifies the newly copied object. Because the runtime cannot distinguish whether more than one variable point to the object, *named* remains 2 in the original object.

2.2.3 Performance of Type I Codes in R

Type I R programs suffer from many performance problems that have in common with other dynamic scripting languages described in 2.1.2, including

- *Dynamic type checking and dispatch* Most of the byte-code instructions and the runtime service functions are type generic. For example, the `ADD` instruction in Figure 2.4 is a type generic operation. It can supports boolean add, integer add, real number add, complex number add, and all the combinations of them. The implementations of this instruction have many checks and branches.

- *Generic data representation* As described in Section 2.2.2, GNU R uses the generic object representation, which requires complex traverse to get the real value and brings a big pressure to the memory management system.

- *Expensive name-based variable lookup* The local frame is implemented as a linked list as described in Section 2.2.2. Each variable lookup requires a linear search and many time

consuming string comparisons.

- *Generic calling convention* R's calling convention can support arbitrary numbers of arguments passing. In order to support it, R uses heap-allocated variable-length argument list, which requires heap allocation as well as linear linked-list traversal.

On the other hand, R also introduces some unique performance issues that are specific to its semantics, such as

- *Missing value NA number support* Not Available, *NA*, is very useful in statistic computing. But there is no *NA* implementation in the processor's number system, such as IEEE 754 standard. GNU R uses `INTEGER_MIN` to represent *NA* for integer, and uses a special NaN (lower word is set to value 1954) to represent *NA* for double precision float. In order to maintain the semantic of *NA* involed computation, special routines are always required in all the mathematics operations, which causes long instruction path length, and inhibit SIMD related optimizations.

- *Out of bound handling* There is no out of bound error. Accessing out of bound value just returns a *NA* value, and assigning out of bound value expands the vector, filling the missing value with *NA*.

- *No reference, assign is copy, and pass-by-value in function calls* In the language semantic, even one element in a very long vector is modified, the whole vector is changed into a new value. In real implementation, this feature is optimized by copy-on-write support introduced in the previous section.

- *Lazy evaluation* R expression is by default a promise. And the promise is bound to the environment where it is defined. In order to force a promise in the future, a new interpreting context is created with the environment the promise bound to. Creating new interpretation context and maintaining all the environments for futures both cause many overhead.

A Motivating Example

For Type I R codes, the performance problems ultimately manifest in the form of long instruction path lengths and excessive memory consumption compared to other languages.

Metrics	AST Interpreter	Byte-code Interpreter
Machine Instructions	26,080M	3,270M
SEXPREC Object	20M	20
VECTOR Scalar	10M	10M
VECTOR Non-scalar	1	2

Table 2.1: Number of machine instructions executed and object allocated for the example in Figure 2.4.

Consider the example shown in Figure 2.4 which accumulates a value over a loop of 10 million iterations. Table 2.1 shows the number of dynamic machine instructions executed and the number of objects allocated for the loop using R-2.14.1 running on an Intel Xeon processor. On average, each iteration of the accumulation loop takes over 2600 machine instructions if executed in the AST interpreter or 300 machine instructions if executed in the byte-code interpreter. The number of memory allocation requests is also high. For instance, the AST interpreter allocates two *SEXPREC* objects and one *VECTOR* object for each iteration of the simple loop. The ephemeral objects also give a large pressure to the garbage collection component later.

Two main causes of excessive memory allocations are identified in this code. There are two variable bindings in each iteration, the loop variable *i*, and the new result *r*. Each binding creates a new *SEXPREC* object to represent these variables in the local frame. The scalar *VECTOR* is the result of the addition (the interpreter does not create a new *VECTOR* scalar object for the loop index variable). Because all R objects are heap allocated, even a scalar result requires a new heap object to hold it. The byte-code interpreter optimizes the local variables binding. But a scalar vector is still required for the addition. Furthermore, a very large non-scalar vector "1:10000000" is allocated to represent the loop space.

Chapter 3

Optimizing R via Interpreter-level Specialization

3.1 Overview

The previous chapter has discussed the performance issues of Type I R codes, and revealed that the problem is mainly from the design and implementation of the GNU R VM. Many research work have tried to improve R's performance through building a brand new R virtual machine. However, these approach all require design a new memory object model that causes these new VMs not compatible with the GNU R implementation. The compatibility is the most importance concern of improving R, because thousands of R libraries hosted on CRAN relies on the internal structure of GNU R memory object model. If the techniques to improve R will break the compatibility, the adoption of these techniques will be seriously inhibited.

In this chapter, a new interpreter-level specialization based approach will be described. The approach aims at improving the performance of Type I codes, while maintaining the full compatibility with the GNU R VM. There have been many attempts in the past to improve performance of other scripting languages, such as Python and Ruby, while maintaining the compatibility. These attempts have had limited successes [31]. The approach proposed here offers a new approach to tackling the problem that combines JIT compilation and runtime techniques in a unique way:

- **Object allocation removal.** Chapter 2 has discussed that excessive memory allocation is the root cause of many performance problems of Type I R codes. As reported in [61], R allocates several orders of magnitude more data than C does. This results in both heavy computation overhead to allocate, access, and reclaim data and excessively large memory footprints. For certain type of programs such as those that loop over vector elements, more than 95% of

the allocated objects can be removed with optimizations. In order to significantly bridge the performance gap between R and other languages, the approach need to remove *most* of the object allocations in the GNU R VM not just *some* of them.

- **Profile-directed specialization.** Specialization is the process of converting generic operations and representations into more efficient forms based on context-sensitive information such as the data type of a variable at a program point.

  When dealing with overhead in the runtime of a generic language, such as R's, the profile-directed specialization is a more effective technique than using traditional data-flow based approaches. This is because program properties obtained by the latter are often too imprecise to warrant the application of an optimization. For instance, traditional data-flow based allocation removal techniques, such as escape analysis, often cannot achieve the target of removing most allocation operations. Instead, the framework based on the proposed approach relies heavily on specialization and shifts many tasks of a static compiler to the runtime. For instance, the object representation specialization is simple yet more effective than traditional approach of unboxing optimization based on escape analysis.

- **Interpretation of optimized codes.** The approach operates entirely within the interpreted execution. The JIT compiles original type-generic byte-codes into new type-specific byte-codes; and the R byte-code interpreter is extended to interpret these new specialized byte-codes.

  The approach focuses on interpretation for two reasons. First, not having generate native codes in the initial prototype greatly simplifies the implementation without affecting the ability to focus on the objectives: allocation removal and specialization. Secondly, the approach is a new solution space for scripting language runtime that delivers good performance while preserving the simplicity, portability and interactiveness of an interpreted environment.

This approach has been implemented as ORBIT (**O**ptimized **R** **B**yte-code **I**nterpre**T**er), an extension to the GNU R VM. On Type I codes, ORBIT achieved an average speedup of 3.5 over

the GNU R byte-code interpreter and 13 over the GNU R AST interpreter, all without any native code generation.

3.2 ORBIT Specialization Example

The key optimization technique used in ORBIT is specialization. Considering the generic object representation of GNU R is one of the most important factor that impacts the performance, ORBIT's specialization expands not only operation side specialization, but also memory representation specialization. Figure 3.1 shows the high level idea.

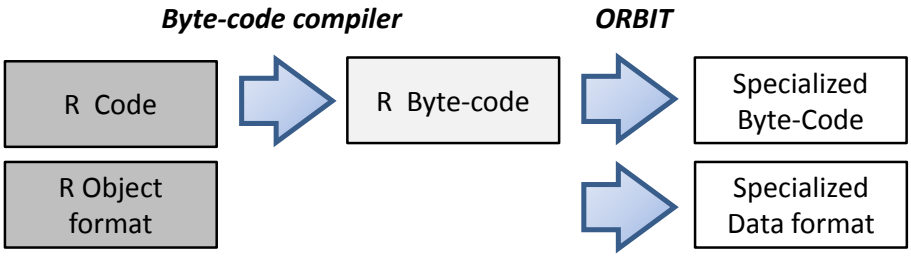


Figure 3.1: Specialization in ORBIT VM

A small specialization example is described here first to explain the key operations of the two kinds of specialization, and the next section will describe each components in ORBIT VM.

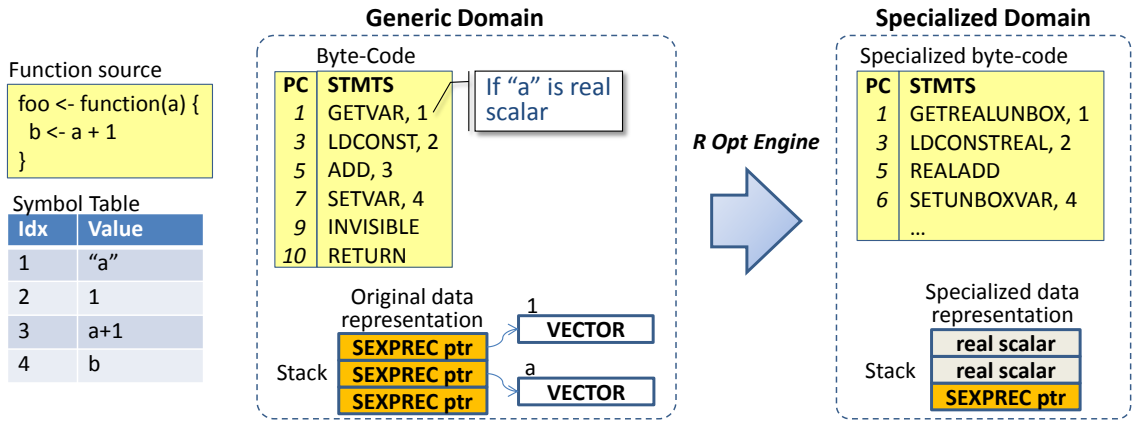


Figure 3.2: An example of ORBIT specialization.

Figure 3.2 shows a small example of specialization. As described in Section 2.2.2, the byte-code instructions are type generic. For example, `GETVAR` looks up the variable `a` in the local frame, and pushes the pointer to it into the stack. `LDCONST` first duplicates (creating a new `VECTOR`) the value in the constant table, then pushes the pointer to the newly created vector into the stack. The `ADD` checks the types of the two operands at the top of the stack, and dynamically chooses the addition routine according to the type of the two operands. A new `VECTOR` is created during the addition to hold the result, and the pointer to it is pushed into the stack.

In order to do the specialization, ORBIT needs to know the type of `a`. The specialization component in ORBIT starts with a runtime type profiling. Then, it uses the profiled type to do a fast type inference. In this example, the type of the constant is known statically as *real*. If the type of `a` is profiled as *real*, too, the compiler will generate specialized code assuming that the `ADD` operates on *real* values. Furthermore, the compiler uses specialized data representation, unboxed real scalar in this case, to represent the values. The right hand side of Figure 3.2 is the specialized result. The compiler makes use of a new class of byte-code instructions and a new data format for the specialization. The specialized byte-code does not require the dynamic type checking and dispatching. The specialized data representation saves the copy of the constant value and the new heap object to store the result.

However, the type of the variable `a` may not be *real* scalar the next time this code segment is executed. To handle this, the compiler adds a guard check in the instruction `GETREALUNBOX`. A guard failure translates the specialized data format into the original generic data representation, and rolls back to the original type generic byte-code.

This simple example illustrates the main characteristics of the proposed approach, including

- Runtime type profiling and fast type inference
- Specialized byte-code and runtime function routines
- Specialized data representation
- Redundant memory allocation removal

- Guards to handle incorrect type speculation

3.3 ORBIT Components

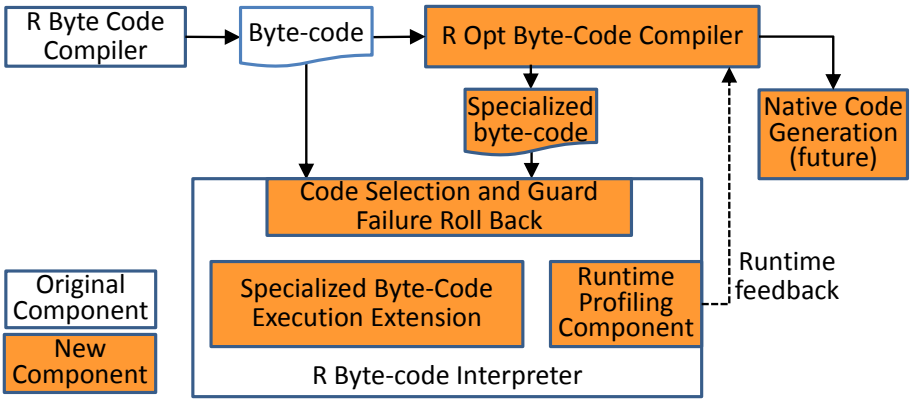


Figure 3.3: The ORBIT VM.

Figure 3.3 shows the diagram of the architecture of ORBIT VM. It is an extension to the GNU R interpreter. It does a lightweight type profiling the first time a function's byte-code is executed. The second time the function is executed, ORBIT compiles the original byte-codes into specialized byte-codes with guards. Specialized byte-codes use the extended data representation and are interpreted in a more efficient way. If a guard detects a type speculation failure, the interpreter rolls back to the original data format and byte-code sequence, and uses the meet (union) of the types as a new profile type.

3.3.1 Runtime Type Profiling

Although type inference can be done without runtime type profiling, pure static type inference is complex and not sufficiently precise especially in the presence of dynamic attributes like those of R. By only instrumenting a few instructions, ORBIT simplifies the type inference and get a more precise result. The interpretation logic of a few instructions is modified to insert the profiling logic. Table 3.1 lists these instructions.

Table 3.1: Instrumented R Byte-code instructions in ORBIT

Category	Instructions
Load	GETVAR, DDVAL,GETVAR_MISSOK, DDVAL_MISSOK
Function Call	CALL, CALLBUILTIN, CALLSPECIAL, ENDASSIGN
Vector Sub-elements Access	DFLTSUBSET, DFLTC, DFLTSUBSET2, DOLLAR

The profiler first gets the type of the object on top of the interpreter stack after an instrumented instruction is interpreted, and stores the type into a profiling table indexed with the interpreter's PC. The type information of a generic R object is stored in three places: the *type* in the header, *attrib* also in the header to specify the number of dimensions (there is no *dim* attribute if the number of dimensions is one, i.e. in the case of a vector), and *length* in the body section to specify the vector length of *VECTOR*. The profiler checks all these attributes, and combines them into a type (see next section) defined by ORBIT. If one instruction is profiled several times (in the same PC location), the final type is the meet of all the types profiled. Because of the R object structure, the type profiling is more complex than other dynamic languages. By carefully design the profiling component, the overhead of profiling is typically less than 10%.

3.3.2 R Optimization Byte-code Compiler and Type Specialized Byte-code

After the profiling information is captured, R optimization byte-code compiler (henceforth referred to as R Opt compiler) use the profiling information to translate the original type generic byte-code into a type specialized byte-code.

The compiler has the following passes

- *Decode* Translate the binary chunk of the byte-codes into R Opt compiler's internal byte-code representation.
- *Build CFG and stack* Build the control flow graph, and analyze the stack shape before and after each instruction.
- *Type inference* Analyze each object's type, including objects in stack and variables defined in the symbol table.

- *Optimizations* A few optimization passes that do the real code specialization and byte-code rewriting.
- *Redundant clean* Clean redundant instructions (Scalar Value Cache load/store, and invalidation).
- *PC Recalculation* Calculate the new PC value for each byte-code instruction.
- *Reset Jump PC* Set the jump target values of the control flow related instructions.
- *Encode* Translate the R Opt compiler's internal byte-code representation back into the binary format of the R byte-codes.

The *Decode*, *Build CFG and stack*, *PC Recalculation*, *Reset Jump PC* and *Encode* passes follow the standard algorithms described in [21]. The techniques used for other passes are described following.

The type specialized byte-codes include about 140 byte-codes as supplemental to the original R byte-code interpreter's 90 type generic byte-codes. These byte-codes are formatted and encoded in the same way as the original byte-codes. The R Opt compiler just performs byte-code replacement. If the type is known, and an optimization is identified, the original type generic byte-code is replaced by the type specialized byte-code. Otherwise, the original byte-code is remained. This design leads to a fast transformation and the best possible compatibility to the original byte-code interpreter. A detail list of the type specialized byte-codes could be found in the ORBIT source code.

3.3.3 Type Inference

A new simple type lattice system of R is defined here for the type inference, shown in Figure 3.4. All vector types have two components, base type (logical, integer, real, etc.) and the length.

The initial type information comes from the type profiling and the constant objects' type. The initial type of a stack operand generated by a profiled instructions is set to the profiled type. If

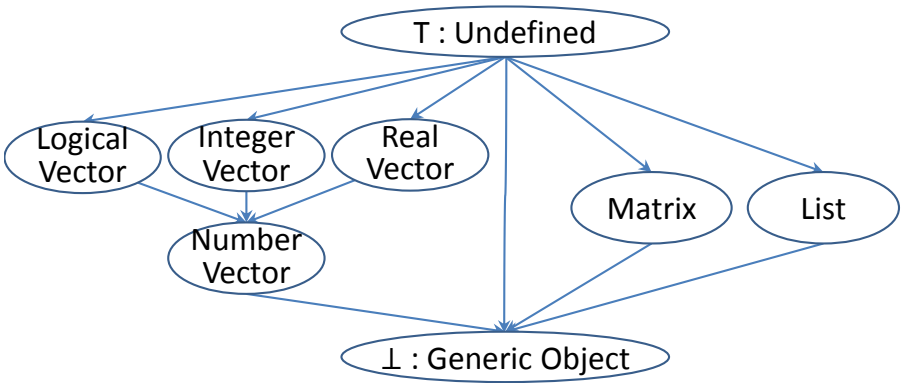


Figure 3.4: The type system of ORBIT.

there is no profiling result (the path is not executed during the profiling run), the type is set to the bottom type, generic R object. The initial type of a stack operand generated by a load constant instruction is set to the type of the constant. All other types of stack operands and local frame variables are set to the undefined (top) type.

The type inference algorithm used here is the standard data flow based algorithm. The algorithm follows the byte-code interpretation order and uses each instruction's semantics to compute types until all the types are stable. Different to the traditional type inference, all the types that rely on profiling are marked as speculated types. All the specialized instructions that use speculated types contain a guard to do the check.

3.3.4 Object Representation Specialization

In order to efficiently represent a typed R object, specialized data structures are used in the ORBIT VM, including 1) a specialized interpreter stack, and 2) the Unboxed Value Cache to hold values in the current local frame.

Stack with Boxed and Unboxed Values

The original R vector type object is always represented as R's *VECTOR* object, even if a scalar object. In the specialized ORBIT runtime, scalar numbers (boolean, integer and real) are stored

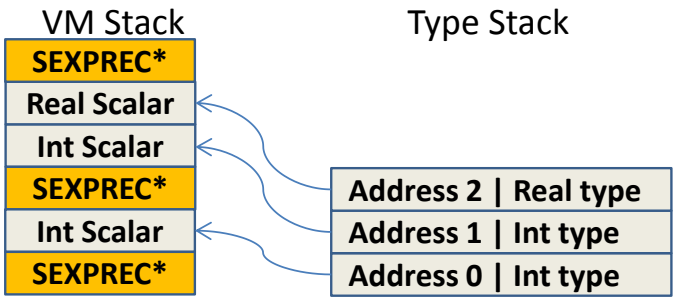


Figure 3.5: The VM stack and type stack in ORBIT.

as unboxed values, and are stored directly into the VM stack. As the VM stack can store both object pointers and unboxed values, another data structure to track all the unboxed values in the VM stack is required. It is called *Type Stack*, and is illustrated in Figure 3.5. Each element in the type stack has two fields: the physical address of the unboxed value in the VM stack, and the type of that value. Type specialized byte-code instructions operate on unboxed values in the VM stack as well as update the records in the type stack.

The type stack is used for two purposes. First, during a garbage collection process, the marker uses it to ignore unboxed values stored in the VM stack. Secondly, during a guard failure roll back, the guard failure handler uses it to restore the VM stack to the type generic object representation.

Unboxed Value Cache

The values of local variables of GNU R VM are stored in the local frame (a linked list). Load and store operations traverse the linked list, and read or modify the binding cells. When storing a new variable, the interpreter must create and insert a new binding cell *SEXPREC* object. If the object value can be represented as an unboxed value, ORBIT optimizes the load or the store by making use of an *Unboxed Value Cache* that avoids the need to do a traversal of the frame linked list as well as create a new binding cell object for store operation.

This cache is only used to store the values of local frame variables. Each cache entry is used for one local frame variable. The index of the variable in the byte-code symbol table is used to locate the cache entry. Each cache entry has three fields, the *value* to store the unboxed value, the

type of the value, and the *state* of the cache entry. There are three cache entry states, *INVALID*, *VALID*, *MODIFIED*, Figure 3.6. The initial state is *INVALID*. A scalar value `LOAD` instruction checks the cache entry's state. If it is *INVALID*, the instruction loads the original object, unboxes and stores it into the value cache and sets the entry as *VALID* state. The instruction then pushes the unboxed value on top of the VM stack. If the entry is *VALID* or *MODIFIED*, it directly pushes the unboxed value on top of the VM stack. A store scalar value instruction directly modifies the unboxed value in the cache entry and sets the entry state as *MODIFIED*.

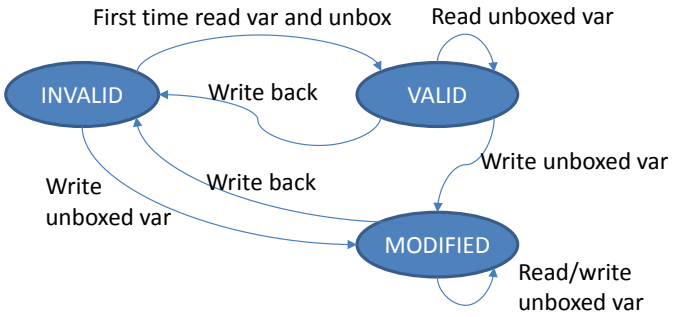


Figure 3.6: States of Unboxed Valued Cache

Because there are two places to store one value, the unboxed value cache and the local frame, a write-back mechanism is used to do the synchronization. The write-back process creates a new R generic object, and binds it back to the local frame. A write-back could be a global write-back or a local value write-back. A global write-back happens when the control flow leaves the current context, such as a non built-in function call. It writes back all the modified entries in the cache. R's semantic allows a callee to access the frames of the caller. The global write back ensures the callee always accesses the latest value in the caller. A local value write-back is performed before a non-specialized load variable instruction. This type of instructions still access the local frame's linked list, and the local write-back only redefines the variable accessed by the instruction. Compared with the original local frame operations, scalar variable read and write only access the unboxed values, and save numerous operations needed for traversing the linked list and creating new R objects.

3.3.5 Operation Specialization

As described in Section 3.3.2, ORBIT introduces many type specialized byte-codes. By using the result of type inference, R Opt compiler translates the original generic instructions into the type specialized instructions. These new instructions are interpreted more efficiently in the extended interpreter. Furthermore, thanks to the object representation specialization, much of specialized operation only needs to interact with the new data representation, which is faster and requires less memory allocation.

Load and Store of scalar values Based on the type inference result, a scalar object's load and store instructions will be changed to the specialized load and store instructions. The new load instruction just looks up the variable in the local frame (if not found, it looks up in the parent frames), unboxes it and puts it into the Unboxed Value Cache and onto the top of the VM stack. A load constant instruction puts the unboxed value onto the top of the VM stack. The new store instruction only needs to update the value in the Unboxed Value Cache.

Mathematics Operations The mathematics operations instructions involving scalars and vectors are transformed into type specialized math instructions. For scalar values, these new instructions are applied to unboxed values, and the result is also an unboxed value in the VM stack. Mathematics operations involving vectors use the type information to do a direct dispatch, saving runtime checks.

Conditional Control Flow Operations Most of the conditional control operations use scalar values to determine the destination of the branch. These instructions are changed to specialized conditional control flow instructions, that can use the unboxed value on the top of the VM stack to control the branch.

Built-in Function Calls Some built-in functions only uses scalar values as arguments such as the docolon (:) function which accepts the range limits as arguments. In the byte-code interpreter,

docolon always creates a linked list to store the arguments. By leveraging the object representation specialization, the docolon call is changed to a special byte-code, which will use a simplified calling convention, and use the unboxed scalar values as arguments. Furthermore, if the result of a docolon function is used only to specify the loop iteration space, ORBIT ignores the docolon function call, and uses the unboxed start and end values directly in the loop.

For-loop The for-loop could benefit from the known type of the loop variable. If the loop variable is a number vector, they are stored into the Unboxed Value Cache and the write-back to the local frame is delayed until the end of the loop (or even further). If the loop index variable is a result of *docolon*, e.g. `for(i in 1:1000)`, a specialized for-loop byte-code sequence is generated to directly use the lower and upper bounds for the loop.

SubSet and SubAssign SubSet and SubAssign are similar to the built-in function calls. The default calling convention needs to create an argument list to store the value and the index. By using the type information and unboxed values, new specialized byte-codes are introduced with the new calling convention that uses the unboxed arguments.

3.3.6 Guard and Guard Failure Handling

All the specializations depend on the type information of the stack operands and local variables. However, many types are inferred from the profiled type. This speculated type may be wrong in subsequent executions of the code segment. Suppose the variable `a` in Figure 3.2 is an integer vector in another function invocation, the specialized instruction *GETREALUNBOX*, *REALADD*, and *SETUNBOXVAR* cannot accept the new `a`. A guard is used to handle this situation.

A `GUARD` has two operands, the expected type of the operand on top of the stack, and the jump back PC value in the original byte-code sequence. Because all the type runtime checks are only needed after a load or a function call or a subset instruction, a `GUARD` is appended after the specialized instruction. The type of the operand on top of the stack is compared with the type

specified in the GUARD instruction. If the types do not match, a guard failure is triggered.

During a guard failure handling, the VM stack will be restored to its generic form. All the unboxed values in the VM stack will be boxed using the record in the type stack. The Unboxed Value Cache will be globally written back. Finally, the interpreter will switch back to the original generic byte-code sequence. And the new type of that object will be recorded in the profiling table for future type inferences.

3.3.7 Other Optimizations

In order to reduce the overhead of ORBIT interpreter, except the specialization optimizations, a few other byte-code level optimizations and runtime optimizations are used.

Super Byte-code It is similar to Superops described in GNU Smalltalk implementation [29]. A few super byte-codes are defined by combining some commonly used sequences of byte-codes to reduce the interpretation overhead. For example, GETVAR, UNBOXINT and GUARD are combined into GETINTUNBOX in ORBIT.

Redundant Box and Unbox Removal Because all the byte-code rewriting transformations are pee-hole optimizations, it's possible a BOX instruction is followed by an UNBOX instruction, or vice versa. This transformation just removes this kind of redundant.

Redundant Write-back Removal Write-back instructions are very heavy since a local variable binding should be linear searched, and new R objects should be allocated. It's important to minimize the number of write-backs. Because of the same reason of pee-hole optimization, some write-back instructions may be redundant in the global scope. A global data flow algorithm is used here to model the cache state of all the local variables. If a write-back instruction is presented as the variable is still in *VALID* state in the unbox value cache, this write-back instruction will be removed.

Redundant GUARD Removal It is similar to Redundant Write-back Removal. Based on the global data flow algorithm, if a `GUARD` has already been guarded by a previous `GUARD`, the later `GUARD` will be removed.

Reuse No Reference Vector Storage GNU R uses *NAMED* tag in the object header to indicate how many variables are bound to the value object, Section 2.2.2. But it does not specify how the *NAMED* should be for an intermediate value object in the stack. Supposing an expression `a+b+c`, the sub-expression `a+b` is first evaluated, and the result is stored on top of the stack, and then `c` is loaded into the stack, and do the addition again. The GNU R will allocate a new vector to store `a+b`, and then another new vector to store `a+b+c`. An simple runtime optimization is used in ORBIT to check the *NAMED* value of the intermediate result on top of the stack for the specialized byte-code that ORBIT introduced. If the *NAMED* value is 0 (for example the result of `a+b`), the vector will be reused to store the next math computation's result (here the result of `a+b+c`). The simple optimization is very effective for long vector math operation, which is quite common in R since it is a vector language.

3.4 Evaluation

The performance of ORBIT has been evaluated to show the effectiveness of the specialization techniques used in ORBIT. The method is measuring the running time and the number of memory object allocations, and comparing these values towith their counterpart in the GNU R AST interpreter and the byte-code interpreter.

3.4.1 Evaluation Environment and Methodology

The evaluation was performed on a machine with one Intel Xeon E31245 processor and 8G of memory. The turbo boost of the CPU was disabled to fix the CPU frequency at 3.3GHz. The Operating System is Fedora Core 16. GCC 4.6.3 was used to compile both the GNU-R interpreter

and the ORBIT VM. All the R default packages were pre-compiled into byte-code as the default installation configuration.

Because the current implementation of ORBIT targets Type I codes, the evaluation focuses on comparing the running time and memory allocations between ORBIT and GNU R interpreters on Type I codes. In order to measure the maximum performance improvement at the steady-state, the evaluation measures the running time of ORBIT only when it runs into stable, not counting the overhead of profiling and compiling. The overhead is separately discussed in Section 4.5.4. All the execution times reported here are the average of five runs. The number of memory allocation requests was collected inside ORBIT, which is instrumented to profile memory allocations.

3.4.2 Micro Benchmark

The first benchmark suite measured is a collection of micro benchmarks, which include `CRT` (Chinese Reminder Problems), `Fib` (Fibonacci number), `Primes` (Finding prime numbers), `Sum` (Accumulation based on loop), `GCD` (Greatest Common Divisor). Although these benchmarks mostly operate on scalars and vectors, they covers a variety of control constructs such as conditional branches, loops, function invocations and recursion. They are common across R applications. The R byte-code compiler[77] uses the similar benchmarks to measure performance improvements.

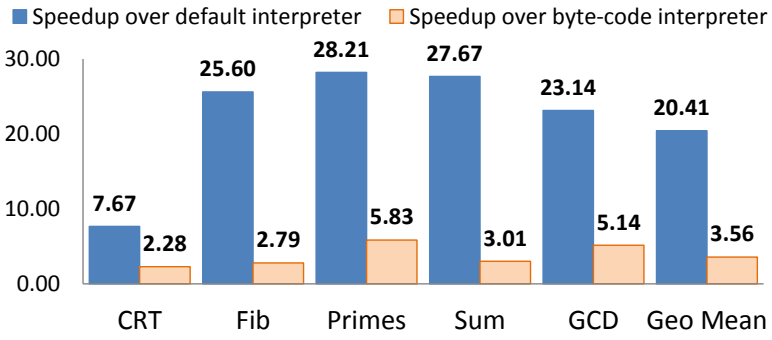


Figure 3.7: Speedups on the `scalar` benchmarks.

Figure 3.7 shows the speedups of ORBIT over the AST interpreter and the byte-code interpreter. ORBIT is more than 20X faster than the AST interpreter. The R byte-code interpreter is

very good at this type of benchmarks because it reduces a significant amount of the interpreting overhead. With the additional optimization and focusing on memory allocation reduction, ORBIT achieves an additional 3.56X speedup over the R byte-code interpreter.

Table 3.2: Percentage of memory allocation reduced for `scalar`.

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%

Table 3.2 shows the percentage of allocated memory in the byte-code interpreter that is removed by ORBIT. For instance, ORBIT is able to remove between 80% to 99% of scalar objects (labeled as VECTOR scalar in Table 3.2) allocated by the byte-code interpreter.

Table 3.3: Metrics of Optimized For-loop Accumulation

Metrics	Byte-code Interpreter	ORBIT
Machine instructions	339M	98M
GC time(ms)	25.94	0
SEXPREC object	20	17
VECTOR scalar	1,000,011	10
VECTOR non-scalar	1	0

Regarding the for-loop example, ORBIT further removed nearly all *VECTOR* scalar object allocation compared to R byte-code interpreter, Table 2.1. With other optimizations, about 98 cycles required for one iteration.

3.4.3 The `shootout` Benchmarks

The `shootout` benchmarks are frequently used in computer science research to compare the implementation of different languages and runtimes. The project reported in [61] uses it to study the behavior of the GNU R implementation. Six of the eleven benchmarks were ported to R here. The benchmarks use a variety of data structures including scalar, vector, matrix, list, which cover most of the data structures in R. The ignored benchmarks are either related to multi-thread, which R doesn't support, or heavily operates on characters, which is not a typical usage of R.

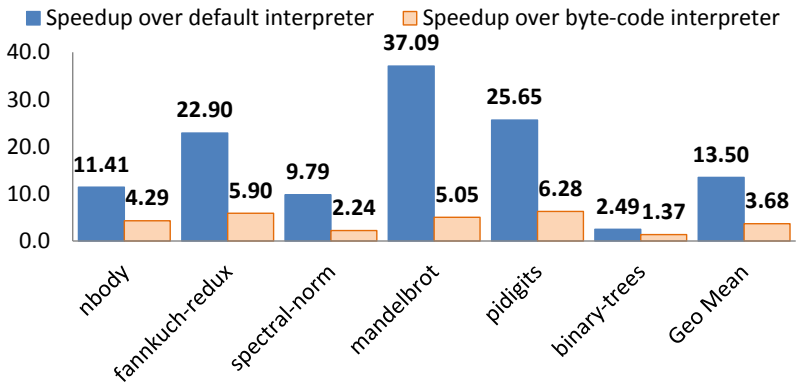


Figure 3.8: Speedups on the `shootout` benchmarks.

As shown in Figure 3.8, ORBIT achieves a significant speedup over the GNU R VM except for `binary-trees`. The `binary-trees` benchmark is dominated by recursive function call overheads thus its performance is heavily dependent on the efficiency of the calling convention. Since the current implementation does not optimize user-level function calling convention, improvements on `binary-trees` are relatively low.

Table 3.4: Percentage of memory allocation reduced for `shootout`.

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%

As shown in Table 3.4, ORBIT reduces significant number of memory allocation of the byte-code interpreter, especially for VECTOR scalar objects. Table 3.5 shows detail metrics taken during the execution of `fannkuch-redux`. Besides the reduction in the number of memory allocations, GC time is also reduced by 95%.

3.4.4 Other Types of Benchmarks

ORBIT was also evaluated on the ATT benchmark [1], the Riposte benchmark [75] and the benchmarks used in [52]. For Type I dominated codes in these benchmarks, ORBIT achieves good

Table 3.5: Runtime measurements of `fannkuch-redux`

Metrics	byte-code interpreter	ORBIT
Machine instructions	1,526M	263M
GC time(ms)	12.06	0.57
SEXPREC object	2,477,740	239,468
VECTOR scalar	2,878,561	20,182
VECTOR non-scalar	854,588	81

speedups similar to what was reported for the scalar and the `shootout` benchmarks. But it only gets small improvements (as low as 15% faster) on Type II codes and nearly no improvements in Type III codes. This is expected as current implementations of ORBIT focuses exclusively on Type I codes. Further optimizations for Type II codes are left for future work. And ORBIT does not intend to address Type III code performance as compilation at R level will not help Type III codes.

3.4.5 Profiling and Compilation Overhead

The overhead of ORBIT comes from two sources. The first one is runtime type profiling. Because ORBIT only profiles a part of the instructions (Section 3.3.1), the profiling overhead is dependent on the percentage of these instructions. Based on the measurement, the overhead is less than 10% in most cases. For example, the overhead of the `shootout` benchmarks is less than 8%. Considering the huge potential of the speedup, this overhead is acceptable.

The second overhead is the JIT time. It is only related to the size of the benchmark codes (including the benchmark itself and all the package codes it invokes). The JIT time in ORBIT is very small, ranging from 2-5 ms for the `scalar` benchmarks, and 10-30 ms for the `shootout` benchmark. ORBIT's JIT is fast compared to other JITs (e.g., Java) because it focuses on specialization not on data-flow analysis and does not generate native codes. The JIT time could be ignored since the running time of the benchmarks in the byte-code interpreter ranges from seconds to minutes.

3.5 Discussions

This section has described ORBIT, an extension to the GNU R VM, to improve R performance via interpreter-level profile-driven specialization. Without native code generation, ORBIT achieves up to 3.5x speedup over the byte-code interpreter on a set of Type I R codes.

Although the techniques described in this section was only implemented and evaluated in the R's context, the interpreter level specialization approach can be straightforwardly applied to other dynamic scripting languages. As described before, interpreter level specialization only introduces additional byte-codes to the original compiler/interpreter. If the byte-code encoding space is not a problem, adding additional byte-code is relatively simple. Most importantly, these new byte-codes can co-exist with the original byte-codes, which simplifies the developing and performance evaluation.

Another important feature of the interpreter level specialization is simple. Many of the dynamic scripting languages are maintained by the open source community, and a simple solution is important for the adoption and maintenance.

Chapter 4

Vectorization of Apply to Reduce Interpretation Overhead

4.1 Overview

The conventional approach to reduce scripting languages' overhead is to either build a new more efficient interpreter or apply Just-In-Time (JIT) compilation techniques. For example, the interpreter level specialization described in Chapter 3 is a combination of these two approaches. However, either approach requires a significant engineering effort, thus imposes a high barrier for adoption. Sometimes, a non-intrusive approach is required to reduce the interpreter's overhead in case there is limitation to change a scripting language's virtual machine.

Chapter 2 classified R programs into three categories, Type I (looping over data), Type II (vector programming), and Type III (glue codes), and showed the major performance problem only appears in Type I R code. But if R program extensively uses vector programming (Type II), the performance gap between R and C is much smaller than the gap in the Type I case. The reason is that many R vector operations are implemented in native C functions, and as the computation is shifted from the interpreter to native C built-in functions, the overhead due to interpretation (which is the most significant source of overhead in Type I codes) is significantly reduced. Figure 4.1 shows the performance of two shootout benchmarks [4] written in the Type II vector programming style. As shown in Figure 4.1, R is much faster than Python, and is less than 10 times slower than the equivalent C implementation. This is a good result considering that in these evaluations the R programs are still interpreted while the C codes are compiled.

Because writing vector codes is much less intuitive than writing scalar codes for most programmers, one possible approach to reduce the interpretation overhead is through automatic vectoriza-

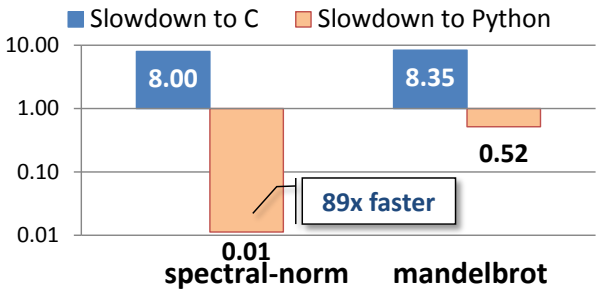


Figure 4.1: Slowdown of R on two `shootout` benchmarks relative to C and CPython.

tion, translating Type I code into Type II code. However, automatic vectorization of arbitrary codes is very difficult, even for static language programs. A recent evaluation shows that "despite all the work done in vectorization in the last 40 years, 45-71% of the loops in the synthetic benchmark and only a few loops from the real applications are vectorized by the compilers we evaluated"[58]. Considering the dynamic features in R, this type of vectorization is even harder.

Yet it does not mean that there is no opportunity to reduce interpretation overhead through vectorization. In this chapter, one common and widely used type of operations in R, the `Apply` class of operations, is studied. The `Apply` operations are similar to the *Map* function in Map-Reduce framework, but have more features and are more flexible. With the popularity of the Map-Reduce programming model, many data analytics problems have been successfully rewritten with the Map-Reduce model [35]. The data analysis programs using `Apply` operations will become more common, because this type of operations can be easily parallelized. The program paradigm is also the foundation of many R based big data analytics framework, such as Rabid[56], SparkR[82], and RHadoop [68].

Although the `Apply` class of operations is simple, the execution of these operations in R is typically very slow due to the large interpretation overhead from iteratively applying the input function to each element of the input data. For example, R code `lapply(L, f)` maps the input data L to the output data with function f. If there are one million data elements in L, the function f will be interpreted one million times, which introduces huge interpretation overhead. In essence, R

codes written in the `Apply` operation style are executed as Type I codes and suffer from significant overhead from interpretation.

An lightweight approach is proposed in this chapter that reduces the interpretation overhead through the vectorization of the `Apply` class of operations in a single thread of execution. The basic idea is to convert `Apply` operations (Type I codes) into a sequence of R vector operations (Type II codes). This approach combines two transformations

- **Function Vectorization** Transform the function used by `Apply` operations into a function that accepts vectors as input. The original function takes one single element at a time. After the transformation, the vectorized function can process a vector input, and generate a vector result.

- **Data Transformation** If necessary, transform the input data (which could be a list of a structure) into the vector form, so that the vectorized function can directly access the input data items in a dense storage form.

After the transformation, the original looping-over-data execution will be replaced by a direct vector function invocation. The vector operations in the vectorized function can take advantage of the built-in vector functions of R, and have much less interpretation overhead in execution.

The rest of this chapter is organized as follows. The `Apply` class of operations, the interpretation overhead and the vectorization's potential is discussed in Section 4.2. Section 4.3 explains the vectorization algorithm. Section 4.4 outlines the realization of the vectorization transformation in R. Section 4.5 presents the empirical results. Finally, the current implementation status, the limitations of this approach, and the application of this approach to other dynamic scripting languages are discussed in 4.6.

4.2 Motivation

4.2.1 R `Apply` Class of Operations and Its Applications

The `Apply` class of operations in R include several built-in functions. In its simplest form, the most common used `lapply` function accepts a list, $L = \{e_1, e_2, \ldots, e_n\}$, and a function f. The value of $lapply(L, f)$ is the list $\{f(a_1), f(a_2), \ldots, f(a_n)\}$. In this context, f is called *Single Object function*. Other functions in the `Apply` family include `apply` which accepts a matrix or multi-dimensional array and returns a dense vector instead of a list, `eapply` which operates on a environment, and `by`, `mapply`, `rapply`, `sapply`, and `tapply` whose semantics can be found in any R manual. Table 4.1 is a summary of these functions.

Table 4.1: Apply Family Operations 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
sapply	A wrapper of lapply to return a vector or matrix
tapply	Apply a Function Over a Ragged Array

Many computations can be naturally written using the `Apply` class of operations including, for example, all the machine learning kernels in [35]. The use of `lapply` is illustrated in Listing 4.1, which shows an R version of the gradient descent Linear Regression.

Applications implemented in terms of `Apply` can be accelerated using R packages, such as SNOW[79], SNOWFall[54] and Foreach, which contain parallel implementations of `Apply` class of operations. The `Apply` programming paradigm is also the foundation of R based big data analytics frameworks, such as Rabid[56], SparkR[82], and RHadoop[68], all of which provide distributed memory parallel implementations of `Apply`.

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

yx <- ... #A list, each element is a [y x] vector
for(iter in 1:niter) {
  delta <- lapply(yx, grad.func)
  theta <- theta - alpha * Reduce('+', delta) / length(yx)
}
```

Listing 4.1: Linear Regression with lapply

4.2.2 Performance Issue of Apply Class of Operations

The underlying interpretation of `lapply` is illustrated in Listing 4.2. Other functions of `Apply` class have the similar interpretation form. This looping-over-data form incurs in huge interpretation overhead, and is very slow. Because the functions of `Apply` class are widely used, GNU R implements them as C functions. However, the interpretation is still used in each invocation of f. Thus, if the input L has one million elements, f will be interpreted one million times.

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

Listing 4.2: Pseduo code of `lapply`

4.2.3 Vector Programming and Apply Operation

The `Apply` class of operations is a general form of array operation. Assume, for example, a vector `a` with one million elements. To add one to each of the elements, the result can be calculated by either `lapply(a, function(x){x+1})`, or `a+1`. Although the results are the same in both

cases[1], the underlying interpretation mechanisms are totally different. The former will iterate over each element of `a`, and invoke the function `function(x){x+1}` as many times as the length of `a`. In the later case, the interpreter will only invoke one *add* operation, which is a built-in function implemented in C. As a result, the `Apply` form requires over one second to execute, while the `a+1` form only needs a few milliseconds.

In the above example, rewriting the `Apply` operation to an array operation is simple, but rewriting arbitrary operations in terms of `Apply` could be very complex. For example, rewriting the vector code for `grad.func` in Listing 4.1 is not as straight forward as the above simple example. It is much easier to write a scalar function that only works on one element of the input, especially the single input element is a complex data structure, such as vector, matrix or a structure composed by vector, matrix and lists. The proposed vectorization method is used to fill the gap. So that it is possible to take advantage of the programmability of single object functions and at the same time benefit from the performance of array computations.

4.3 Algorithm

4.3.1 Vectorization Transformation

The concept of `Apply` vectorization is expressed as

$$Lout \leftarrow Apply(L, f) \Rightarrow Lout \leftarrow \vec{f}(L) \tag{4.1}$$

where L, f and $Lout$ have the same meaning as in Section 4.2. f is called *Single Object Function*. $\vec{f}$ is a new function, in the form of a *Vector Function*, that can process all the elements of L and return $\{f(a_1), f(a_2), \ldots, f(a_n)\}$. In this way, the original loop shown in Listing 4.2 is transformed into a single function invocation.

Figure 4.2 shows the changes of the interpretation logic after `Apply` vectorization. Before the

[1]The result data representations are different in R

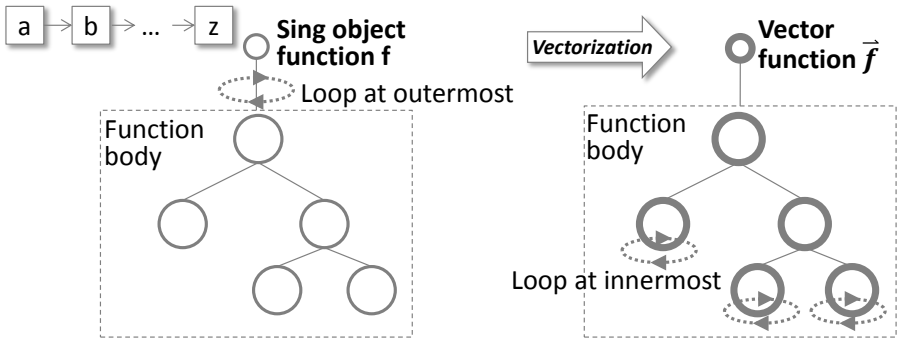


Figure 4.2: Function Vectorization Transformation

transformation, the loop over all the input data can be considered as an outermost loop. Thus, if f is inlined into Listing 4.2, the statements in f would be the loop body of the loop in Listing 4.2. After the vectorization, the loop is moved from the outermost to the innermost (inside each statement of $\vec{f}$) so that the function $\vec{f}$ is only executed once. In the ideal case, each single statement of $\vec{f}$ is an R's vector built-in function, then $\vec{f}$ is only interpreted once, and the interpretation overhead is dramatically reduced.

Although the concept of `Apply` vectorization is simple, and this SPMD type vectorization does not require data dependence analysis and all its complexity and inaccuracies, automatic vectorization of the function is not a trivial task in a dynamic scripting language, like R. There are several difficulties

- *Vector language as input* The single object function could directly operate on vector objects (vector, matrix, higher dimensional array) or other complex composite data types with built-in vector operations. The vectorization here should handle *vectorizing vector operations* correctly and efficiently.

- *High level functional language features* The function and operation are first-class objects, that could take the form of user written R code, library defined functions, user written native code, built-in (implemented through native libraries) functions, or arbitrary combinations of these. Vectorization must support all the possibilities, which is much more complex than the vectorization of a conventional language (such as C).

- *Dynamic language* The function, operation and data are all resolved dynamically. The vectorization here must dynamically generate correct code to support the mixture of the vectorized execution context and the single object execution context.

This section describes the automatic algorithm that translates the original `Apply` operation into a vector function invocation. It will first describe a basic algorithm under some simplifications. Then a the full algorithm will be explained afterwards.

4.3.2 Basic Algorithm

The basic algorithm's goal is to get the vectorized function $\vec{f}$ of the single object function f that satisfies Equation 4.1.

In order to simplify the discussion, Three assumptions are set here (1) The input data is already a well stored array, so that there is no need to convert the input data into an array as would be the more general case where the input data is a list; (2) The control flow inside f does not depend on the function's formal parameters; (3) f is normalized so that all assignments are of the form $v_3 \leftarrow op(v_1, v_2)$.

The algorithm changes the operations and variables inside f. Some variables in f will be changed from representing a *single* object to representing a *vector* of objects. They are defined as EXPANDED variables. Because R is a vector language, the *single* object in the single object function can be a scalar, a vector, or even a matrix. The end result is that the expanded version of the *single* object becomes a vector, a matrix, or a three dimensional array. Then variables still representing a *single* object are defined as UNEXPANDED variables. Algorithm 1 illustrates the process of vectorizing the simple class of functions just described.

If the basic vectorization algorithm is applied to the `grad.func` function in Listing 4.1, `yx` will be marked as EXPANDED at the beginning, then `y`, `x`, `error`, and `delta` will be marked as EXPANDED because they are in the use-def chain of `yx`. Regarding the operation side, if the input data `yx` is $\{[y_1, x_1], [y_2, x_2], \ldots, [y_n, x_n]\}$, `yx[1]` should return the first item of each elements in

Data: The single-object function f in $apply(L, f)$
Mark formal parameters of f as EXPANDED;
while *There is a change* **do**
  foreach $v_3 \leftarrow op(v_1, v_2)$ *where either* v_1 *or* v_2 *is* EXPANDED **do**
    Rewrite op to a new $\vec{op}$ that satisfy;
    if v_1 *and* v_2 *are both* EXPANDED **then**
      $\vec{op}$ will do element-wise operation, and assign the result to v_3. Mark v_3 as EXPANDED
    end
    if *Only one of* v_1 *or* v_2 *is* EXPANDED*, say* v_1 **then**
      $\vec{op}$ will use the UNEXPANDED v_2 to operate with each element in v_1, and assign the result to v_3. Mark v_3 as EXPANDED
    end
  end
end
foreach *Access to an* EXPANDED *variable* **do**
  Rewrite it with array access operation;
end
foreach *return*(v) *statement* **do**
  if v *is* UNEXPANDED **then**
    Rewrite $return$ so that it expands v (by replication to the same length as the input)
  end
end

Algorithm 1: Basic function vectorization algorithm

`yx`, which is $[y_1, y_2, \ldots y_n]$. `yx[2]` should perform in the similar way. The vectorized `c` operation should combine the UNEXPANDED `1` with each element in the EXPANDED `yx[2]`, and return an `Expanded` result. The other vectorized operations $\star$, $-$, `sum` should perform in the same way. The function can return the result `delta` directly, since it is already an EXPANDED variable.

Following the basic algorithm, the only changes are those shift of the computation to simple operations and accesses, following the structure defined in Figure 4.2.

If all operations in the original function f are flattened, the vectorization transformation of `Apply` operations can be seen as loop distribution, Figure 4.3. Because all input formal arguments dependent variables have been expanded, the loop distribution is a legal transformation. If one statement in f is another loop, besides the loop distribution, a loop interchange is also required to move the `Apply` level loop into the innermost. Because all the data dependence can only appear in that statement's loop, and there are no dependences across iterations of the `Apply` level

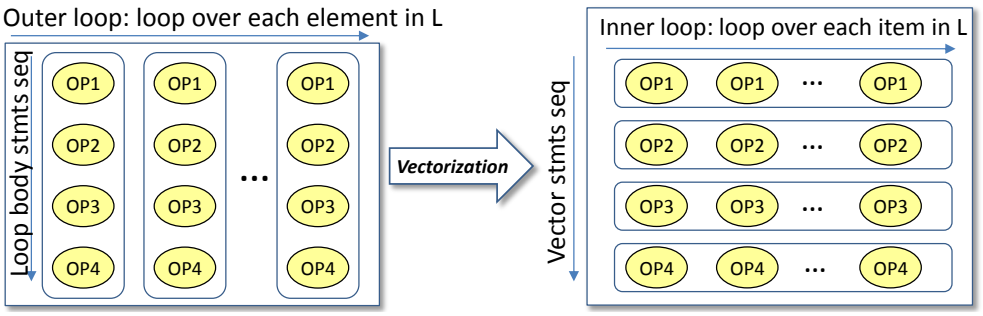


Figure 4.3: Loop Distribution in Vectorization Transformation

loop, the loop dependence vector can only be $[=, <]$. Then the loop interchange is also a legal transformation. After `Apply(f, L)` is re-wrote to $\vec{f}(L)$, the original behaviors are maintained.

Although the basic algorithm is correct, the operation transformation can be complex. The behaviors of the corresponding vector operations in $\vec{f}$ for each operation in the single object function f should be defined. For example, define how to handle EXPANDED v.s. UNEXPANDED variables. And the basic algorithm does not support control divergences (that does not support control flows that depend on EXPANDED variables). The full vectorization algorithm discussed next will address all the limitations.

4.3.3 Full Algorithm

Figure 4.4 illustrates the three tasks in the full vectorization algorithm. (1) *Data Object Transformation*, which permutes input data so that the vectorized function can get direct access to the data item (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 direct vector function invocations, and perform other optimizations to reduce the overhead. The original Apply(L, f)'s context is named as the single object context, and the transformed vector function's context is named as the vector context.

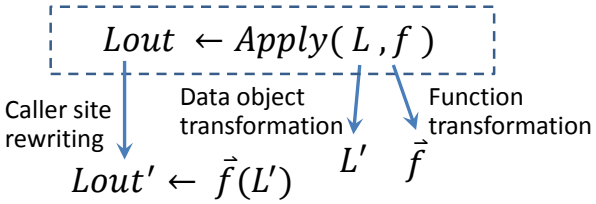


Figure 4.4: Three Tasks in the Full Vectorization Algorithm

Data Representation and Transformation

Data Representation is the shape(structure) of the input and output data. Below D and L describe how each dimension of a data object is stored. D means dense storage, L means list storage, Q means either D or L. Then the Data Representation of any data objects from a scalar (zero dimension) to a multi-dimensional structure can be described. Table 4.2 lists some examples.

Table 4.2: Data Representation of Different Types

Scalar	$[\,]$
1-D	Vector $[D_1]$; List $[L_1]$
2-D	Matrix $[D_1 \times D_2]$; List of Vector $[L_1 \times D_2]$; List of List $[L_1 \times L_2]$
Higher Dimensions	Array $[D_1 \times D_2 \times \ldots \times D_n]$; List $[L_1 \times L_2 \times \ldots \times L_n]$

Based on the notation, two data transformation operations, `PERM_DOWN` and `PERM_UP`, are defined. `PERM_DOWN` moves the first dimension of the data to the innermost position, and if the first dimension is stored in the list format (L), it will be converted into dense form (D). The transformation is expressed as $[Q_1 \times Q_2 \times \ldots \times Q_n] \Rightarrow [Q_2 \times \ldots \times Q_n \times D_1]$

Once the data is transformed with `PERM_DOWN`, the data access in the vectorized function $\vec{f}$ can get direct access to the vector data without a gather operation. Figure 4.5 shows three examples, all of which can still use simple operations, even the original expressions to access the input data as a vector.

The `PERM_UP` does the reverse transformation. It moves the innermost dimension to the outermost position. This data transformation is typically used to turn the vector function's return value back to the original shape.

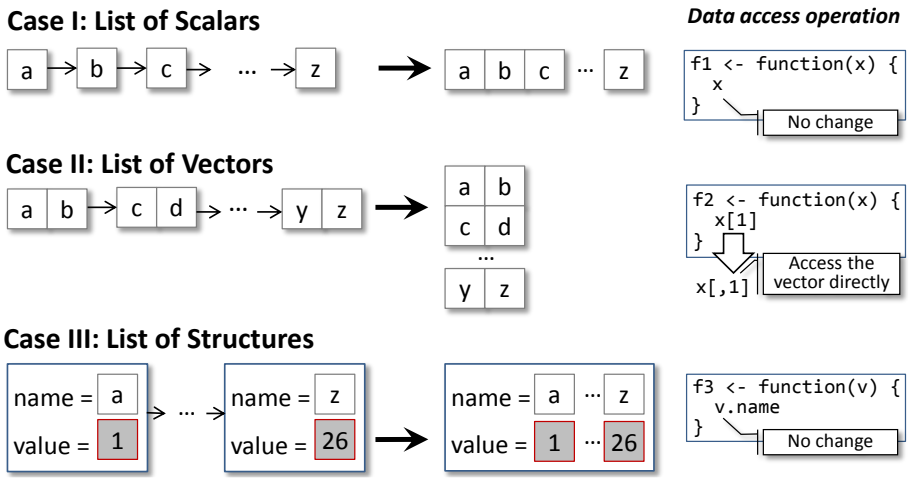


Figure 4.5: Data Access after PERM_DOWN Transformation

With these two transformations, each data object has two representations, the DOWN shape and the UP shape. The single object context always uses the UP representation, and the vector context always uses the DOWN representation. These two transformations are extremely powerful in processing nested `Apply` vectorization. In each `Apply` vectorization, the data can be transformed by applying PERM_DOWN to move the data's outermost dimension to the innermost place to match the movement of the computation loop from the outermost to the innermost.

A REP_EXPAND operation is defined here, too. It appends another vector dimension to the data object by replicating its value along the last dimension. For example REP_EXPAND of a scalar will generate a 1-D vector, $[\ ] \Rightarrow [D]$. REP_EXPAND of a 1-D list or of a vector will generate a 2-D data structure, $[Q_1] \Rightarrow [Q_1 \times D_2]$. REP_EXPAND to a higher dimensional data structure behaves similarly. REP_EXPAND is used in support of the vectorization of operations. Suppose there is an expression $a \oplus b$ in the single object function f, and in the transformation of f into $\vec{f}$, a is marked as EXPANDED (i.e. a is a vector) $\vec{a}$. If the vector operation $\vec{\oplus}$ requires both operands to be EXPANDED, REP_EXPAND can be used to fix this limitation by generating expression $\vec{a}\vec{\oplus}$REP_EXPAND(b).

In the above notation, the interface change of the function vectorization from the single object function to the vector function can be represented as $f(v : [Q_1 \times \ldots \times Q_n]) \Rightarrow \vec{f}(\vec{v} : [Q_1 \times \ldots \times$

$Q_n \times D_{n+1}]$).

Function Vectorization

The goal of function transformation is to transform the single object function f into the corresponding vector version

$$f(v : [< \ldots >])body \Rightarrow \vec{f}(\vec{v} : [< \ldots > \times D])\vec{body} \tag{4.2}$$

A new operation `VEC_FUN` is defined here to do the function vectorization, where $\vec{f} \leftarrow$ VEC_FUN(f). There are three situations that `VEC_FUN` operates with

- *Direct replacement* If the function f is an elemental function, such as most of the math functions in R, the transformation just returns f. Elemental functions are those that can operate on a single object or on a vector of objects. If the vector version of f is another function g, such as `cbind()` is to `c()` in R, just replace f with g.

- *Generic replacement* If f is a built-in function that cannot be vectorized, or f is too complex to be analyzed, a generic vector version of f is created:

$$\vec{f}(\vec{v} : [< \ldots > \times D])\{\text{PERM\_DOWN}(\text{Apply}(f, \text{PERM\_UP}(\vec{v})))\}$$

  The generic function just uses `Apply` to simulate the vector execution. Because $\vec{f}$'s input data has been transformed with `PERM_DOWN`, $\vec{v}$ must be `PERM_UP`ed so that the `Apply` operation can iterate over the right data. And the result must be `PERM_DOWN`ed again to comply with the vector function's output data shape. This *fall-back mechanism* makes sure that the vectorization algorithm can support arbitrary code.

- *Body vectorization* If the body of f can be analyzed,the following algorithm is used to transform f into $\vec{f}$.

The vectorization of f requires the formal parameter rewriting, which changes the original parameter's type into the EXPANDED type by appending one additional dimension, and the body transformation, which enables the new body process the EXPANDED input correctly. The body transformation has four passes: variable type inference, loop transformation, branch transformation, and code rewriting.

Variable Type Inference The goal is to decide each variable's shape, EXPANDED or UNEXPANDED. Assuming a gated SSA representation [63], if a variable is on the left hand side of any assignment (including assignments from gated PHI functions) in the use-def chain starting at the function's formal argument input list, it is EXPANDED. Otherwise, the variable is UNEXPANDED, which means the variable is the same as the variable in the single object function. Algorithm 2 shows the algorithm.

Data: SSA form based function body
Result: All variables' shape type
Set f's arguments' types as EXPANDED;
Set all other variables' types in the body as UNEXPANDED;
while *Type changes* **do**
  foreach $v_3 \leftarrow op(v_1, v_2)$ **do**
    $shape(v_3) \leftarrow shape(v_1) \sqcup shape(v_2)$
  end
  foreach $v_3 \leftarrow \phi(v_1, v_2)$ *with gated condition* c **do**
    $shape(v_3) \leftarrow shape(v_1) \sqcup shape(v_2) \sqcup shape(c)$
  end
end

Algorithm 2: Variable Type Inference

Loop Transformation It processes the loop statements that might have control divergences in the function body. These loops can be identified by examining the condition expression that terminates the loop. If the expression contains a variable whose type is EXPANDED, the loop must be processed. One possible approach to transform this kind of loop is rewriting the loop body that each statement in the body is controlled by a mask expression as discussed in [53]. But it requires modifications to the interpreter to support predicated execution and introduces wasted computation

due to the masked operation. Here the fall-back mechanism described in the *generic replacement* is used to simulate the vector loop body with sequential loop over the original loop body. A synthesized loop is generated. The original loop is put into the synthesized loop, and is interpreted asif it had not been vectorized.

The synthesized loop is generated as follows. First all EXPANDED variables in the loop are categorized into the two groups: (1) Variables read in the loop but not defined in the loop: $\{v_1, \ldots v_n\}$; (2) Variables written in the loop and live after the loop: $\{u_1, \ldots u_m\}$. The vectorized loop body takes the form shown in Listing 4.3.

```
1 for(idx = 1 .. vector_len) {
2   v1 <- vec_v1[idx], ... , vn <- vec_vn[idx]
3   original loop
4   vec_u1[idx] <- u1, ... , um <- vec_um[idx]
5 }
```

Listing 4.3: Generic Loop Body Transformation

Branch Transformation This pass transforms control divergence branches. Specifically, it transforms `if` statements with conditional expressions containing EXPANDED variables into data dependence linear statements. the algorithm follows the approach in [22]. Given an `if` statement with EXPANDED condition c. All the EXPANDED variables will be located in the true and false blocks that are written in the blocks but still live after the `if` branch. These variables are the operands of the ϕ nodes in the `if` statement's post-dom basic block. Then, a flattened basic block is generated, containing in the order: the pre-dom block, true block, false block, and post-dom block. Finally, the post-dom's ϕ nodes is replaced with `Select` statement (`ifelse` in R), $u_3 \leftarrow \phi(u_1, u_2) \Rightarrow u_3 \leftarrow \text{select}(c, u_1, u_2)$.

Code Rewriting The final pass performs the code rewriting to replace all operands related to EXPANDED variables to the corresponding vectorized operands. This pass checks each statement $r \leftarrow op(v1, \ldots, v_n)$. If any operand in the right hand side is EXPANDED, it replaces op with VEC_FUN(op), and replaces each UNEXPANDED variable, say v_i, on the right hand side

with REP_EXPAND(v_i). This rule is also applied to ϕ node, which means a ϕ node's operand could be REP_EXPAND(v_i). In the final phi node remove step, if one operand of the ϕ node is REP_EXPAND(v_i), where v_i is from the basic block `B`,the statement $v_i \leftarrow$ REP_EXPAND(v_i) is inserted at the end of `B`.

Figure 4.6 shows the full vectorization steps of a simple function.

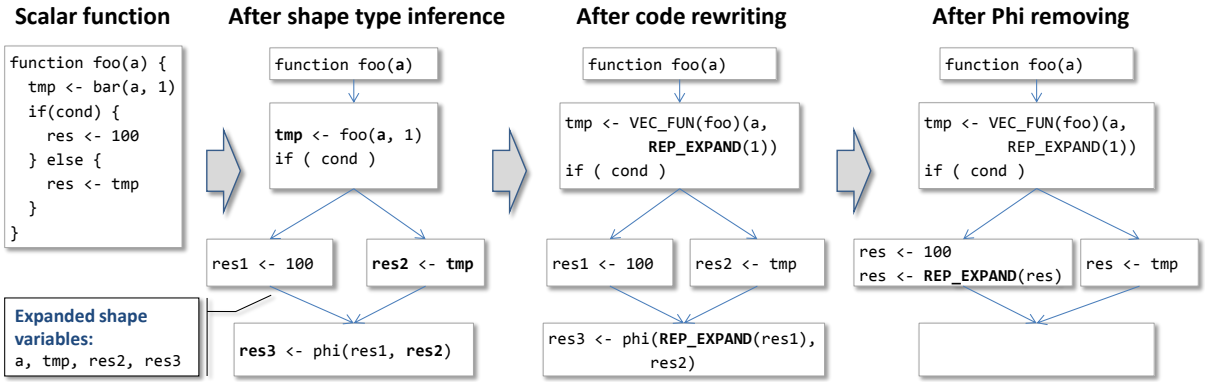


Figure 4.6: Function Vectorization Transformation Example

Caller Site Rewriting

With the single object function $f(v : [< \ldots >])$ and the corresponding vectorized function $\vec{f}(\vec{v} : [< \ldots > \times D])$, the `Apply` function call $Apply(f, data : [L\times < \ldots >])$ has the same behaviors as the direct vector function invocation PERM_UP($\vec{f}$(PERM_DOWN($data : [L\times < \ldots >]$)))

The caller site rewriting performs a local code transform to replace all `Apply` function calls $Apply(f, data)$ with the vector function invocations PERM_UP(VEC_FUN(f) (PERM_DOWN($data$))). For a dynamic scripting language like R, the body of f is not known at the compiler time, and the compiler just replaces f with VEC_FUN(f). In the real execution, the `VEC_FUN` will do a dynamic function transformation.

After the code rewriting, the vector function invocation has much less interpretation overhead compared to the original `Apply` function call. However, the new code also introduces overhead from `PERM_DOWN` and `PERM_UP` data permutation. One important task of the caller site rewriting

is to do code optimization to reduce the data transformation overhead. First, the generated code in the caller site may contain PERM_UP(PERM_DOWN(v)) or PERM_DOWN(PERM_UP(v)). These connected reverse transformations can be removed easily. Second, as described in Section 4.3.3, each data object has two representations, the UP and the DOWN. A runtime memorization could be built to record the internal link between the two representations. In the first time either `PERM_DOWN` or `PERM_UP` is invoked, the runtime will record the linkage. If either UP or the DOWN data object is modified, the linkage must be broken. When `PERM_DOWN` or `PERM_UP` is called again, the runtime will check whether the internal link is still valid. If it is, the data transformation function could directly return the corresponding data representation without the heavy data permutation. This optimization is very effective to iterative data analysis algorithms.

4.4 Implementation in R

The vectorization algorithm has been implemented an R package. Since it is written in R, the implementation is interpreted and installation does not involve compilation.

4.4.1 Runtime Functions

Data Transformation Functions

The three data transformation functions `PERM_DOWN`, `PERM_UP`, and `REP_EXPAND` are all implemented in a recursive style so that they can support arbitrary nested data structures. The first two functions only take one input argument, the data, and return the DOWN or UP representations accordingly. `REP_EXPAND` must know how many times to expand the input data. So the function takes two input arguments: the UNEXPANDED data object, and an EXPANDED variable, whose shape will be used to compute the result (i.e to expand the UNEXPANDED parameter). The second argument is typically the first formal parameter of the function that is being vectorized.

Listing 4.4 shows the core `PERM_DOWN`. The implementation of `PERM_UP` and `REP_EXPAND`

have a similar structure.

```
1  PERM_DOWN <- function(l) {
2    e1 <- l[[1]] # the first element in the list
3    if(is.list(e1)) { #e1 is a list structure
4      sov <- list() #result, structure of vector
5      for(i in 1:length(e1)) { #transform sub-item
6        sov[[i]] <- PERM_DOWN(lapply(l, function(e){e[[i]]}))
7      }
8      sov
9    } else { # e1 is a vector or an array
10     tmp <- simplify2array(l)
11     if(is.null(dim(tmp))) {
12       tmp # a simple vector case
13     } else { # do permutation
14       ndims <- length(dim(tmp))
15       aperm(tmp, c(ndims, 1:(ndims-1)))
16     }
17   }
18 }
```

Listing 4.4: Implementation of PERM_DOWN

Vectorization Functions

VEC_FUN has been implemented as described in Section 4.3.3. When it is invoked at run time with an input R function, it parses the function, creates its AST, and applies the following transformations

Data Access Operations Because the actual parameter of the function has been transformed with PERM_DOWN operation, most data accesses maintain their original form or are changed slightly. To describe the transformation to memory accesses, consider a variable x in the original function f, that is marked as EXPANDED in $\vec{f}$, there are three basic classes of accesses involving x:

- x: the operation in $\vec{f}$ will remain x, return the whole EXPANDED variable.
- $x[a]$: where x is a vector in the original function. In the vectorized function, x will be a matrix, and we should use $x[, a]$ to access the ath column. The reason why we must access by columns is that R uses column-major storage.

- $x\$label$: in this case x is a structure (list) in the original function. In the vectorized function, x is a structure of a vector, and $x\$label$ does not need to be changed to access the EXPANDED value.

More complex data access operations can be decomposed into the above cases.

Direct Replacement of operations R has many operations with a higher dimensional counterpart that is to be invoked when the operands are expanded. It is by using these higher dimensional operations that the proposed system reduces the overhead. Whenever `VEC_FUN` finds one of these operations in the original function, it replaces the operation with its higher dimension equivalent as shown in Table 4.3.

Table 4.3: R Functions Supporting Direct Replacement

Low Dim	High Dim	Notes
+,-,*,/, ∧	+,-,*,/, ∧	They support vector by default
sum	rowSums	Sum along the row sides
mean	meanSums	Mean along the row sides
length	nrow	Length in the row side
c	cbind	Column binds
unlist	Simplify2array	Transform list of vectors into matrix
which.max	col.max	Find the positions of each row's maximal value

Generic Replacement If the `VEC_FUN` routine cannot analyze or transform an operation (a native implemented function that is not in Table 4.3, or a very complex R implemented function), it generates a generic expressions using `Apply` to simulate the vector execution as described in Section 4.3.3.

Recursive Transformation In many cases, the operation in a function body will be transformed by making a recursive call to `VEC_FUN`. For example, *op* will be changed into VEC_FUN(op). It will invoke the vectorization routine at run time to get the vectorized operation function of *op*.

4.4.2 Caller Site Interface

The vectorization package provides two translation APIs, `va_compile` and `va_vecfun`. The former takes an R expression as input, and the later accepts a function. Both of them will go through the input object's AST, and do caller side code rewriting, translating `Apply` class of function calls to direct vector function invocations. Listing 4.5 shows the compiled code of Listing 4.1 by `va_compile`.

```
yx <- ... #A list, each item is a [y x] vector
for(iter in 1:niter) {
  delta <- PERM_UP(
             VEC_FUN(grad.func)(PERM_DOWN(yx))
           )
  theta <- theta - alpha * Reduce('+', delta) / length(yx)
}
```

Listing 4.5: Code Generated from `va_compile`

And List 4.6 is the vectorized `grad.func` function generated from the expression `VEC_FUN(grad.func)`.

```
grad.func <- function(yx) {
  y <- yx[,1]
  x <- cbind(REP_EXPAND(1, yx), yx[,2])
  error <- rowSums(x * REP_EXPAND(theta, x)) - y
  delta <- error * x
  return(delta)
}
```

Listing 4.6: Vectorized `grad.func` Function

4.4.3 Optimizations

The code in List 4.5 and List 4.6 have not been optimized. There are many redundant computations. Several optimizations have been applied to remove these redundancies.

Remove redundant data transformation

This kind of redundancy is mainly from the iterative part in a program. For example, `PERM_DOWN(yx)` in Line 4 of Listing 4.5 will be invoked in each of the loop iteration. But `yx` is loop invariant. LICM (Loop Invariant Code Motion) can be used here to remove the redundancy. In the implementation, the runtime memorization technique described in Section 4.3.3 is used. List 4.7 is the optimized code. Because R has neither a map data structure nor references, a hidden variable approach is used to do the memorization. The variable `.va.yx` is the DOWN shape representation of `yx`, and `.vasrc.yx` records where the DOWN variable is from. If the function requires a DOWN variable, the code first checks whether a DOWN variable exists (Line 4), then checks whether the source variable has not been changed (Line 5) with `identical` function. If both conditions are satisfied, the DOWN variable will be directly returned, otherwise the `PERM_DOWN` is invoked to do the data transformation, and set the linkage.

```
1  yx <- ...
2  for(i in 1:50) {
3    .va.delta <- VEC_FUN(grad.func)
4        ({if(!exists(".va.yx", inherits = FALSE)
5           || !identical(.vasrc.yx, yx)) {
6              .va.yx <- PERM_DOWN(yx)
7              .vasrc.yx <- yx
8         }
9         .va.yx
10       })
11   delayedAssign("delta", PERM_UP(.va.delta))
12   theta <- theta - alpha * va_reduceSum(.va.delta) / length(yx)
13 }
```

Listing 4.7: Optimized Code from `va_compile`

R's copy-on-write mechanism makes sure any modifications to the source object break the runtime linkage. In the previous example, both `yx` and `.vasrc.yx` point to the same memory object after the runtime linkage is created. If `yx` is modified, the copy-on-write mechanism will create a new memory object for `yx`. Then the `identical` function will return false in the checking.

The runtime linkage approach can also be used to the function vectorization. `VEC_FUN(grad.func)` of Line 4 in Listing 4.5 is also loop invariant, and the same approach is

used to save redundant VEC_FUN function calls.

Another optimization to remove redundant data transformation is in Line 11 with R's delayedAssign function. The direct vector function invocation returns a DOWN shape variable .va.delta. Line 11 creates a promise of delta. If there is no usage of delta, the promise will never be forced, and the PERM_UP function call is saved.

Remove redundant data replication

The REP_EXPAND in x * REP_EXPAND(theta, x) from Line 4 Listing 4.6 may be redundant because * supports implicit value replication [2]. Static compiling and runtime check are used to remove the redundant data replication. If the operation supports implicit data replication(all R functions supports a scalar's implicit replication), the vector compiler will do a static type check to see whether an UNEXPANDED variable is a scalar. If it is, the REP_EXPAND can be removed statically. Dynamic type checking inside REP_EXPAND is also inserted. If REP_EXPAND finds out the input UNEXPANDED variable is a scalar, the function will directly return that variable.

Optimize Reduce Function Call

R's Reduce function call is interpreted in the similar style of Apply class of operations through a looping-over-data style, which also has large interpretation overhead. If the reduction's operator is +, sum or colSums are used to replace Reduce. These two functions are implemented by native library, and have much less interpretation overhead. One limitation is that sum and colSums can only take dense vector object as input. However, many input data of Reduce come from the result of Apply function calls in the Map-Reduce framework. After vectorization, the result of the vector function invocation is a dense vector representation (DOWN shape). and it can be used by sum or colSums without calling PERM_UP. Line 12 of Listing 4.7 is the optimized code. The va_reduceSum is the runtime function that checks the input data's type and call either sum or colSums or the original Reduce.

[2]The REP_EXPAND in Listing 4.6 cannot be removed because theta is a length two vector

4.5 Evaluation

The performance of the vectorization compiler is evaluated in this section by comparing the running time of the vectorized code with the time of the original code using `Apply` class of functions.

4.5.1 Benchmarks

Table 4.4: Benchmarks and configurations

Name	Descriptions	Configurations	Base Input Size
ICA	Independent Component Analysis	Un-mixing 2 signals	1M samples
k-Means	K-Means clustering of one dimensional points	10 clusters	1M points
K-Means-nD	K-Means clustering of n dimensional points	3D points, 10 clusters	1M points
LogitReg	Logistic Regression of one variable	Scalar sample	1M samples
LogitReg-n	Logistic Regression of n variables	Length 10 vector sample	1M samples
LR	Linear Regression of one variable	Scalar sample	1M samples
LR-n	Linear Regression of n variables	Length 10 vector sample	1M samples
NN	Nearest Neighbor	10K training samples(3D point), 10 categories	10K testing samples
kNN	k Nearest Neighbor	10K training(3D point), 10 categories, k=5	10K testing samples
LR-OST	Ordinary Least Squares method of one variable	Scalar sample	1M samples
LR-OST-n	Ordinary Least Squares method of n variables	Length 10 vector sample	1M samples
Monte Carlo	Monte Carlo Pi Calculation	Each sample is a 2D variable	1M samples
PCA	Principle Component Analysis	Length 10 vector sample	1M samples

The benchmarks in the evaluation are kernels of data analytics and machine learning algorithms collected from [56], [82] and [35]. hese benchmarks are slightly modified so that they use the `Apply` class of functions and run as a single R process. Table 4.4 lists the kernels and the configurations used in the evaluation. The first seven kernels use iterative algorithms and the last six use direct methods. The number of iterations for iterative algorithms are fixed so that the running time is not dependent on the data value. The benchmark kernels include both the single-variable configuration and multi-variable configuration for some algorithms, for example LR and LR-n, to make a more extensive evaluation. In fact, if only the single-variable configurations is included, the performance speedup would be very good, but the result would be biased. Regarding the implementation, the single-variable configuration and multi-variable configuration are different. Some special routines are used to optimize the single variable's case, and its base performance is slightly better than the multi-variable implementation when restricted to a single variable.

4.5.2 Evaluation Environment and Methodology

The evaluation was performed on a machine with an Intel E5-2670 processor, 64G memory, Linux CentOS 6.3, and GNU-R 3.1.2. All the R packages including our vectorization compiler were pre-compiled into byte-code as the default installation configuration.

The running time for iterative benchmarks are *one iteration*'s computation time, and measured as the average time between iteration 6 to iteration 15, when the application runs into a steady state. As a result, the time of the vectorized code does not contain the initial (in iteration 1) data permutation time. The running time for direct algorithms are the *end-to-end* running time, which includes the data permutation overhead in the vectorized code case. We will report the data permutation's overhead for both the iterative benchmarks and direct benchmarks. Table 4.4 only lists the base input size. The input size of 4x and 16x were also evaluated. For example, 4M and 16 M samples as input for ICA.

4.5.3 Vectorization Speedup

Figure 4.7 and 4.8 show the speedup number of the kernels with `Apply` operation vectorization versus the default built-in `Apply` function call. Our approach can achieve up to 35x, with an average 15x's speedup for iterative benchmarks and 5x for direct benchmarks. The high speedup is mainly from the machine instruction reduction. For example, the machine instruction of the vectorized LR is only 1/40 of the original version in the base input case. But the vectorized version has a higher CPI (1.13 to 0.85) from a higher cache miss rate due to the lack of data locality. The final 29.7x speedup is the combination of these two effects. Other benchmarks have similar hardware performance counter metrics.

The vectorization approach is not sensitive to input size. With the increase of the input size, the speedup number even becomes higher as a result of lower CPI due to long sequence of consistent operations, and no worse cache miss rate.

The speedup number of different benchmarks varies a lot, depending on different factors such

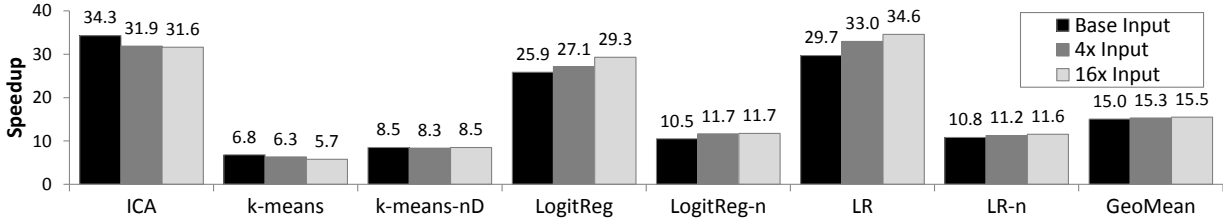


Figure 4.7: Speedup of `Apply` operation vectorization (Iterative benchmarks)

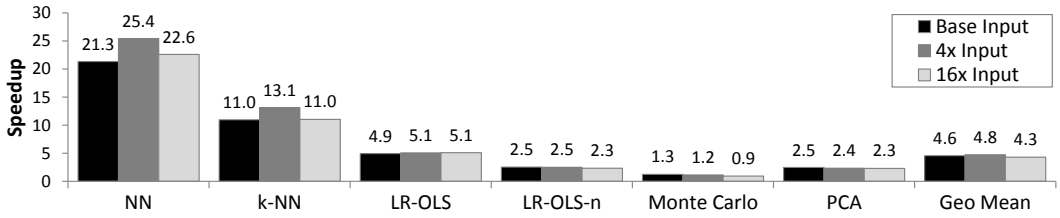


Figure 4.8: Speedup of `Apply` operation vectorization (Direct benchmarks)

as the algorithms used (iterative or direct), the overhead of data transformations, the coverage of the vectorized operations, and the degree of vector programming in the original code. We will discuss these factors in the following sections.

4.5.4 Overhead of Data Transformation

The data transformation is a necessary step for the vectorized function to access the data in a vector form (Section 4.3.3). However, this kind of data permutation is very expensive, especially when the input data's size is large. In general, the data transformation overhead arises from `PERM_DOWN` and `PERM_UP`. Because the output of the vectorized function is either consumed by the following vectorized function or by a reduce function, the overhead from `PERM_UP` is very small. So we only discuss here the input data transformation's time. We normalize this time in terms of the running time defined before (*one iteration*'s time for iterative benchmarks and all the time for direct benchmarks)

Table 4.5 shows overhead of iterative benchmarks, which is very large in most cases. The

Table 4.5: Data transformation overhead (Iterative benchmarks)

Benchmark	Base Input	4x Input	16x Input
ICA	358.8%	387.1%	425.6%
k-Means	46.2%	57.9%	69.0%
k-Means-nD	50.5%	54.9%	58.8%
LogitReg	824.6%	941.4%	995.4%
LogitReg-n	465.4%	616.6%	733.3%
LR	1064.6%	1217.8%	1271.1%
LR-n	526.8%	688.8%	762.3%

computation time of many benchmarks after vectorization is small, and the data transformation's overhead is relatively large. But it's not a big problem for iterative algorithms, since the overhead will be amortized by all the iterations. Considering the 1271% overhead of LR, if there are 100 iterations, each iteration only increases 12.7% running time in average, and end-to-end speedup number is still very high.

Table 4.6: Data transformation overhead (Direct benchmarks, Base Input)

Benchmark	Overhead%	Speedup w/o OH	End2end Speedup
NN	0.3%	21.4	21.3
kNN	59.7%	27.2	11.0
LR-OST	70.9%	17.0	4.9
LR-OST-n	33.3%	3.8	2.5
Monte Carlo	72.7%	4.6	1.3
PCA	40.2%	4.2	2.5

Table 4.6 shows data transformation overhead of direct benchmarks with base input size. The overhead is also very high in most cases as expected. It's a problem for direct algorithms since the overhead cannot be amortized. A much higher speedup number could be achieved without the overhead, , shown in the *Speedup w/o OH* column.

There are two reasons that the data transformation overhead of Nearest Neighbor is very small. First, the data set is relatively small, only two 10K 3D points, and the data transformation can be performed in the higher level cache. Second, the computation of Nearest Neighbor is complex. However, in the k-NN case, some functions in the single object function has no corresponding vector version. The vector computation has several fall-backs to generic transformation cases, which introduce heavy back-and-force data permutation.

4.5.5 Vectorization of Nested Apply Functions

Some the kernels in Table 4.4 contain nested `Apply` function invocations. For example, k-Means and Nearest Neighbor both have two `Apply` operations nested. The outer `Apply` of k-Means loops over all the points, and calculates each point's distances to all the centers, while the inner `Apply` loops over all the centers, and calculates one point's distance to each center. The outer `Apply` of Nearest Neighbor loops over all the testing points, and calculates each point's distances to all the training points, and do the classification. And the inner `Apply` loops over all the training points, and calculate one testing point's distance to each training point. The vectorization could be applied to the outer loop, the inner loop or both.

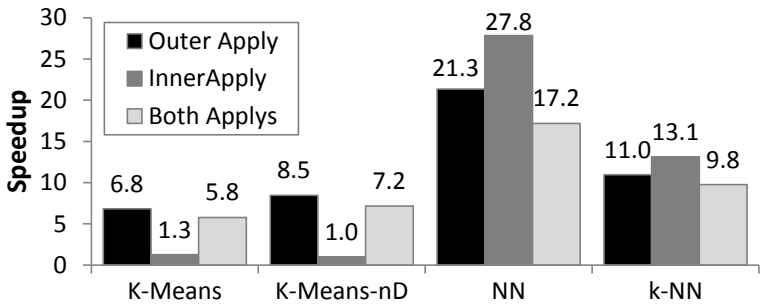


Figure 4.9: Speedup of different levels' Apply vectorization

The vectorization in static languages typically happens in the inner loop for better performance. However, it's impossible to decide which `Apply` call is the outer or inner in R, because all the functions are resolved dynamically. In the experiments conducted in previous subsection, vectorization happens when the compiler meets the first `Apply` function call. The performance of this schema(outer only) by comparing it with other two schemas (inner only and both). Figure 4.9 shows the result.

In k-Means, the inner `Apply` only loops over 10 cluster centers, and the inner only schema has limited performance improvement. Vectorization of both `Applys` achieves the best performance, but the outer only schema still gets most of the benefit. The inner part and the outer part of Nearest

Neighbor both have 10K iterations, and the outer only and inner only have similar speedup. If both levels are vectorized, the data object will be expanded into a 10k by 10k sized object, which causes large memory requirements, and has less speedup. Based on these observations, the outer only schema seems to be good enough for the benchmarks. A better schema would take the input data's length into consideration to decide when to vectorize dynamically.

4.5.6 Vector Programming in the Applications

Another factor that impacts the speedup of the proposed vectorization is the usage of vector programming in the original single object function. For example, Figure 4.7 shows that LR-n has smaller speedup than LR. The reason is that each sample in the input data of LR-n contains 10 variables, and it is represented as a length 10 vector in the code. The single object gradient descent function uses vector programming to calculate the `error` and update the `theta`. This is a Type II R program, which has relatively low interpretation overhead. The proposed vectorization algorithm can handle the single object function with vector programming well. Because the base case has less interpretation overhead, the vectorization achieved relatively low additional speedup. Figure 4.10 shows the speedup number by varying the n values of LR-n, LogitReg-n, k-Means-nD and LR-OLS-n. As discussed before, the smaller the n value, the base function is more likely a Type I R code, and has large interpretation overhead. Then, the proposed approach can achieve better speedup. On the other hand, the larger the n value, the smaller speedup of the vectorization approach can achieve due to the relatively efficient of the base case.

4.5.7 Tiling in Vectorization

One negative effect of `Apply` operation vectorization is that it increases the memory footprint. The `Apply` operation must be applied to relatively long input data lists or arrays so that the overhead can be offset by reducing interpretation overhead. In the vectorized function, each single vector computation operates on long operands, and store the intermediate result into another long vector.

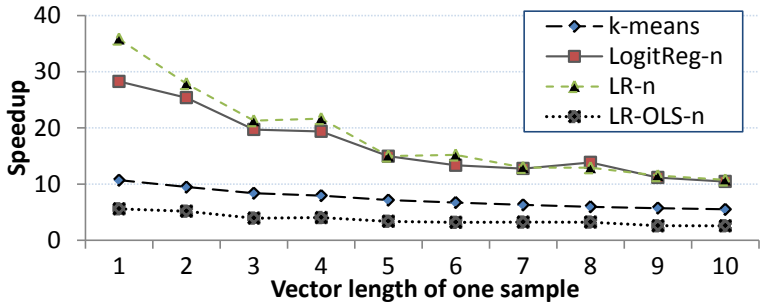


Figure 4.10: Speedup of different vector lengths (Base Input)

These sequences of long vector operations hinder data locality. And the long intermediate results also consumes more memory.

The next experiment configured the Linear Regression example with different tiled sizes, and measured the performance impacts. The smaller tiled size, the more synthetic loops are required to implement the vectorized function, and the overhead increases accordingly. For example, if the tiling size is 1000, and the input data's length is 1 million, there are 1000 invocations of the vectorized function in each algorithm iteration of the Linear Regression example.

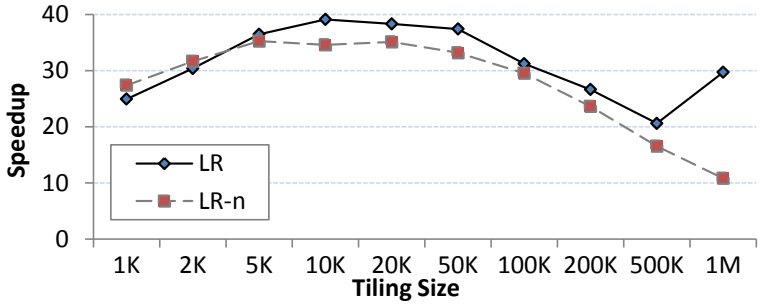


Figure 4.11: Speedup of Different Tiling Sizes (Base Input)

Figure 4.11 shows the speedup number of different tiled sizes in the vectorized function to the original base case. A good tiled size can achieve much higher performance improvement compared to the no tiling case. The smaller tiling size suffers large interpretation overhead. The data collected from the hardware performance counter showed that the smaller tiled size case executes more native instruction, and has a much higher instruction cache miss rate (R has very

large instruction footprint, there is nearly no native instructions reuse for different loop iteration's interpretation routines). The larger tiled size suffers large data cache miss rate. The hardware performance counter reveals the larger tiling size case has much higher cache miss rate (up to 10x more) due to loss of data locality.

In order to get the best performance speedup, a optimal tiling size is required, which is quite hard in real practice. A better way is to do native level vector code fusion [75]. After the proposed vectorization, the function contains long vector based computations, which are very suitable to apply native level code fusion.

4.5.8 Built-in Vector Function's Support

One of the most important factors that impacts the performance of vectorization is the vector computing support from R's built-in functions. If all the operations in the vectorized function are in Table 4.3, a higher speedup could be expected, which explains the high speedup in Linear Regression. However, many functions do not have the corresponding higher dimensional functions. For example, in the early phase of the research, the vector version of `which.min` is not found in R. `which.min` is used in k-Means to locate the closest center of one data point. It accepts a vector, and returns a scalar. The vectorized function of `which.min` should take a matrix in, search the index of the min value in each row, and return a vector out. There is no such a built-in function in R. Then the vectorization compiler had to use the generic replacement transformation that uses `Apply` to go through each row of the input matrix to simulate the vector execution. Because the function is a time consuming part in k-Means, the initial speedup of the vectorized k-Means is not high. After a detail search, the vector version of `which.min(v)` can be expressed by `col.max(-v)`, and additional 30% speedup is gained. Howver, there is still no vector version of `crossproduct` function in R, which limits the speedup of LR-OST and PCA.

4.6 Discussions

4.6.1 Pure R Based Implementation

The current implementation of the proposed vectorization is purely based on R. The reason is to make the maximal possible compatibility. Based on the design decision, all the runtime functions in the package are implemented in R code, and there is no additional native implemented functions introduced as the vector version of some commonly used R computation functions. The advantage of the approach is (1) compatibility and simple, no need any modifications to GNU R VM; (2) Easy to install, the installation does not involve native level C or FORTRAN compilation. But the obvious disadvantage is the performance limitation.

If the design decision is changed. Suppose, the interpreter of GNU R can be changed, then a predicated execution mode could be introduced into the GNU R interpreter, With the interpreter support, there is no need to use a synthesized loop to handle the loop transformation, and there is no need to use control branch linearization, (both described in Section 4.3.3). And the interpretation of these constructs could be much faster.

Another less intrusive way is to provide the native implemented vector version of commonly used R functions. For example, the vector version of `crossproduct`. It only requires native compilation during the package installation, and no need to change the GNU R VM. A better performance is expected with more engineering efforts, which is not the scope of this thesis.

4.6.2 Combine Vectorization with Parallelization

As the code is transformed into long vector computation (Type II R codes), it exposes more opportunities to explore the parallelism in different levels. First, it is much easier to use the processor's SIMD unit if the operation is already a vector operation. Secondly, thread level parallelism can be introduced to get better performance. Thirdly, the approach can be integrated into R cloud computing frameworks, such as Rabid and SparkR to explore the distributed parallelism. The integration

can help reduce the interpretation overhead of each single node in these frameworks, and achieve a good performance improvement for the whole system.

The three levels' parallelism is orthogonal to the proposed approach, and a better performance can be expected with any combinations. Chapter 5 will describe the combination of this proposed approach with SparkR in a distributed environment.

4.6.3 Different to Conventional Automatic Vectorization

This approach differs significantly from conventional automatic vectorization. First, the transformation does not need to determine whether the operation can be parallelized because parallelism is implied in the semantics of `Apply` operations. Secondly, the speedup of the transformation shown in this chapter does not come from exploiting parallel hardware resources (such as multi-threading, SIMD, or GPU), but from reducing the interpretation overhead of Type I codes. So the speedups manifest even in a single-thread of execution. In the mean time, new challenges appear as the approach tries to *vectorize a vector language* in a dynamic scripting language context.

4.6.4 Limitations

The lightweight approach proposed in this section can get very good performance comparing with the original sequential `Apply` operation. However, it still has some limitations in applying this method.

- *No side effects of the single object function* Although it is an implicit requirement for `Apply` function call, some single object functions used by `Apply` still has side effects in real practice. It's quite hard to identify these side effects in a dynamic scripting language context. But if the proposed approach is used to transform this kind of `Apply` calls, the semantics of the original statement is changed.

- *Homogeneous data structure of each element in the input list* The `PERM_DOWN` requires each element of the input list of `Apply` has the homogeneous structure, so that the output DOWN

shape object can be constructed. In some rare cases, this condition is not satisfied, and the `PERM_DOWN` cannot be performed correctly. In R, this limitation can be solved by filling *NA* to make the data homogeneous. But this filling will introduce additional data transformation overhead.

- *Possible Large Memory Usage* In many usage scenarios, the vectorization technology proposed here will reduce the memory usage. The reason is that the original data is transformed into a more dense vector storage. Because R's list data structure is composed by the small *SEXPREC* objects (Section 2.2.2), the transformed vector storage uses much less memory. However, in some cases, vectorization will create large intermediate vector object. For example, the original distance calculation in k-NN will generate a distance vector (or list) for one test sample to all the train samples, and the distance vector (or list) will be consumed and dead after the minimal distance is found. Then it will be garbage-collected. However, after the vectorization transformation, the vector distance computation will generate a distance matrix that contains all test samples' distances to all the train samples. Although the matrix will be garbage collected finally, it may use a large memory space in the intermediate stage if the number of the testing samples and the number of train samples are large.

- *Possible Slower Running Speed* As described in the evaluation section, the proposed approach requires the built-in low overhead vector functions. If there is no built-in vector function support, or the vector function coverage is low, the transformed vector version may run slower than the original sequential version. However, this can be decided at the compilation time. If such a situation is met, the transformation can be ignored, and still use the original sequential execution.

4.6.5 Extended to Other Dynamic Scripting Languages

The technique of this section can be easily extended to vectorize *Map* operations in other interpreted dynamic scripting languages. With the popularity of the Map-Reduce framework, many

dynamic scripting languages are featured with *Map* style operations. Besides R's `Apply` class of operations, Python has built-in `map` functions and list comprehensions syntax [3], JavaScript's `Array` data type supports `map` and `foreach` operation, and Matlab has a number of similar functions (`arrayfun`, `cellfun`, et al.) that can be used with anonymous functions to perform map operations.

The proposed approach can be applied to other dynamic scripting languages in three ways

- *Direct Application* If the target language has built-in vector computation support, such as Matlab, this approach can be applied directly, and a similar performance improvement could be expected.

- *Add Vector Extension* If the target language has no built-in vector support, but can be extended to add vector support, such as NumPy[6] to Python, this approach can still be applied.

- *Expand the Vectorizable Constructs* The current algorithm can only be applied to language constructs with the semantic `MAP`. However, if a loop in a dynamic scripting language with each loop instance is independent and has no side effect, this transformation can still be very useful to reduce the interpretation overhead. A dynamic scripting language may be extended with some annotations introduced (similar to OpenMP) to expose these loops and use the proposed approach to vectorize them.

[3] Example code: `squares = [x**2 for x in range(10)]`

Chapter 5

R Vectorization in Distributed R Computing Framework

5.1 Introduction

Chapter 3 and Chapter 4 have described two approaches to improve R's performance through interpreter level specialization and operation vectorization. These two techniques successfully improved the single R instance's performance. However, because of the information explosion, it's impossible for a single R instance to process the real BIG data. For example, the Airline on-time performance data set from DataExpo 2009 [2] contains 120M records stored in 12 GB raw data. It's very hard for a typical R instance to process all the data quickly, even not consider whether a single R instance can load all the data into the memory.

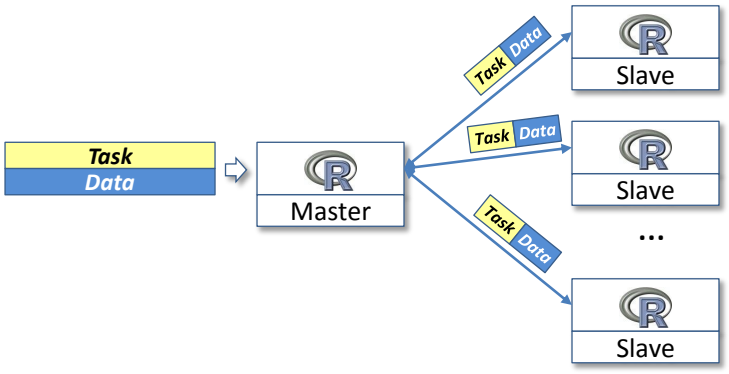


Figure 5.1: Parallel Computing of R in Master-Slave Model

Parallelism is the only solution to help R scale out to process the real BIG data. There are many different levels of parallelism in R. Due to the thread-unsafe implementation of GNU R, there is no thread concept introduced. In a shared memory machine with parallel hardware processes,

process level parallelism has been used to process R's data in parallel. SNOW [79] is based this model, which provides the parallel version of `lapply`. The master R instance receives the data and the single object function from the user, and launches several slave R instances to distribute the computation. This mechanism is a stand Master-Slave parallel model, shown in Figure 5.1. Because the implicit parallelism of `lapply`, it fits very well with SNOW. SNOW and its extension SNOWFall[54] supports this type of parallelism to distributed memory machines using socket communication or MPI. In this model, there is one R instance working as a master, and this master R instance will be responsible for distributing both the R task and the data to all the slave R instances.

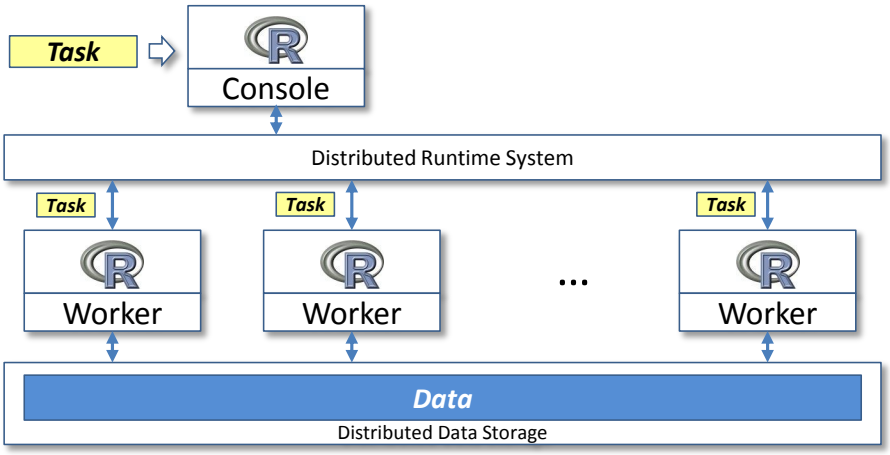


Figure 5.2: Parallel Computing of R based on Distributed Runtime System

One limitation of the Master-Slave system is the master node must take all the data in. It's highly possible the master node will become the bottleneck, for example, the case that the master node cannot read all the data into its memory.

Recently, there is a new trend to scale R into a distributed environment via integrating R with an as-is distributed computing framework, and leverage the distributed framework's capability to do task and data scheduling. The structure of this kind of distributed R system is illustrated in Figure 5.2. This model can scale up to processing much larger data set than the Master-Slave model, because both the task and data are managed by the runtime system. The console node is

not the bottle neck any more. SparkR[82], RHadoop[68], Rabid[56], RHIPE[42] are all following this model. Although different systems have different architectures and implementations, the underlying computing models of them are all based on Map-Reduce. As a result, all of them provide `lapply` or similar interfaces.

Both the SNOW style Master-Slave parallelism model and the distributed Map-Reduce style model use the `lapply` interface as the basic building block to provide a global `lapply`. Then the runtime system schedules the computation to each individual process or node. In the finest granularity of each R instance, R's default `lapply` is used to perform this level's computation. As described in Chapter 4, `lapply` is interpreted as Type-I R code, and it suffers large interpretation overhead. Although the high level system leverages the parallelism in different levels to accelerate the data processing, the single node's computation is still very slow, which leaves a large room for improvement. In this chapter, the vectorization techniques described in Chapter 4 will be applied into these frameworks to improve their performance. SparkR will be used as an example to show how `Apply` vectorization is integrated in to accelerate the computation. But this approach is not limited to SparkR, and it can be applied to other R parallel frameworks, too.

The rest of this chapter is organized as follows. SparkR is briefly introduced in Section 5.2, as well as its APIs and a simple example. Section 5.3 explains the integration implementation of the vectorization into SparkR. Section 5.4 presents the empirical evaluation results. Finally, the related issues and possible improvements to SparkR are discussed in Section 5.5.

5.2 SparkR Background

SparkR is an open-source project that integrate GNU R with Apache Spark[12]. It provides a light-weight front-end to use Apache Spark's distributed computation power from R.

5.2.1 Basic Structure

The architecture of SparkR follows the structure described in Figure 5.2. The runtime system of SparkR uses Apache Spark, and the data storage could use HDFS[72], in memory, or other distributed storage systems. Spark uses RDD (Resilient Distributed Dataset) to express the large data object stored in the underlying storage layer, and exposes APIs to manipulate the RDD. SparkR exposes RDD to the R user by defining RDD as an R S4 type object. It also provides R APIs to manipulate the R RDD object.

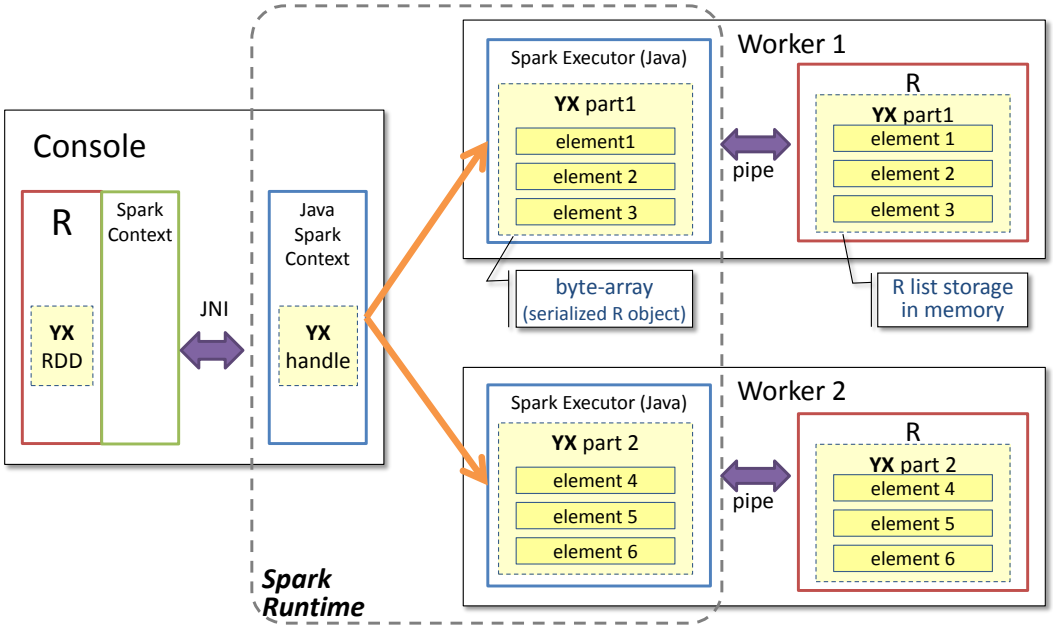


Figure 5.3: SparkR Architecture and Object Storage

Because `lapply` is the basic building block of SparkR, the RDD is internally an R distributed list object. Each partition of an R RDD object in one single R instance's memory is indeed a normal R list object. Figure 5.3 show SparkR's architecture. `YX` is an R RDD object. The RDD object in the console is just a handler (reference) of this distributed object, and the real data is expressed as Java byte-array (serialized R object), and distributed in the Spark Executors across the whole system. In each R computation operation, the Java byte-array is de-serialized back into an R instance, and the corresponding R operations are invoked to process the object.

5.2.2 SparkR APIs

SparkR provides a bunch of APIs to create, transform, and aggregate R RDD object. The full list of the APIs could be found at [15]. Among them, the most important API is `lapply`, which overrides R base package's `lapply` for processing RDD data type. Another important API is `reduce`, which is similar to R base package's `Reduce`, but it is used with RDD data type. With SparkR's API, user can simply transform the original sequential `lapply` based application into a distributed application that can process large amount of data set. Listing 5.1 shows the Linear Regression example in SparkR.

```
1  grad.func <- function(yx) {
2    y <- yx[1]
3    x <- c(1, yx[2]) #Add 1 to est interception
4    error <- sum(x * theta) - y
5    delta <- error * x
6    return(delta)
7  }
8
9  sc <- ... #A Spark context
10 YX <- ... #A RDD list in the sc context. Each element is a [y x] vector
11 alpha <- ... # Control the changes in each iteration
12 for(iter in 1:niter) {
13   delta <- lapply(YX, grad.func)
14   theta <- theta - alpha * reduce(delta, '+')
15 }
```

Listing 5.1: Linear Regression with lapply in SparkR

Compared to the standalone R's Linear Regression example, Listing 4.1, there are only a few changes, including `YX` is now an RDD object, and `Reduce` is changed to `reduce` with some parameter order changes. After the application is launched, SparkR will create the distributed `YX` object, and it will also distribute the function `grad.func` and the console's data `theta` to all the workers. The runtime also handles the reduction from the workers to the console. This model greatly simplifies the programmer's effort for distributed computing. The programmer now only needs to think about the algorithm, and no need to worry about how to schedule data and the task.

5.2.3 Performance Problems

SparkR is still in the very early development phase, and one big problem is that it runs very slow. In some workloads, SparkR in a cluster is even slower than a single node standalone sequential R. Although SparkR can process large data set, but if the processing speed is too slow, the adoption of SparkR is still hampered.

There are at four sources that cause the slow performance. (1) The slow processing speed of GNU R itself; (2) The data exchange between Spark Java executor and R instance in the worker; (3) The R instance launching overhead for each SparkR blocking operation; and (4) The slow `lapply` interpretation mechanism in each worker. The first problem also exists in single node's GNU R, and can be solved by an efficient R implementation, for example ORBIT introduced in Chapter 3. The second problem can be solved by an efficient R-Java interaction mechanism and fast serialization/de-serialization implementations. The third problem can be solved by reusing R instance approach. And the SparkR project is working on solving the second and third problems. The last problem is exactly the one described in Chapter 4.

In order to solve the slow interpretation speed problem of `lapply`, SparkR provides an additional API, called `lapplyPartition`. Similar to `lapply`, this API still accepts two parameters, the *data*, which is now a chunk of data (typically the whole portion of an RDD in a worker), and the *mapper function*, which is a function to process the whole data chunk. The API itself will not run faster. But if the programmer organizes the chunk of data into a vector storage, and provides a vector function to process the vector data, the `lapplyPartition` will only be invoked once for one worker, and the underlying invocation is a vector function over a vector data, the interpretation overhead will be much smaller than the original `lapply` based case.

However, in order to use `lapplyPartition`, the programmer needs to think about how to organize data, and how to write the vector function, which is tedious and complex. Furthermore, the algorithm should also be changed compared with the original `lapply` based implementation. For example, if the `YX` in Listing 5.1 is changed to vector partition storage, and the `grad.func`

is changed to a vector function, the `lapply` then could be changed into `lapplyPartition`. But the reduce operation should be also changed, because the reduce's input now is not a list of elements, whose reduce semantic is clear, but a few vector values. The programmer should take care of the new reduction semantics in the worker, or across workers, or both.

All of these changes bring large burden to the programmer, and an automatic solution is desired to handle these transformation, and improve SparkR's performance.

5.3 Integration R Vectorization to SparkR

The `Apply` vectorization technique described in Chapter 4 is used to improve the performance of the standalone R instance's performance in executing `Apply` class of operations. Because the main performance problem of the `lapply` in SparkR exists in each worker's R instance, the `Apply` vectorization technique could be adopted into the SparkR context to improve SparkR's performance. In the original single node context, the transformation is changed from looping-over-data `lapply` into a direct vector function invocation. But in the SparkR's context, the transformation goal is changing the distributed `lapply` call over a RDD list into a distributed `lapplyPartition` which invoke a vector function over each partitioned data in each cluster.

Similar to the single node's vectorization, there are also three transformation tasks

- *Function Vectorization* Transform the original single object function into a vector function that can work on a chunk of vector data in one worker. The vectorized function will be fed into `lapplyPartition`.

- *Data Transformation* Transform the distributed RDD list into another RDD object, in which each chunk is stored as the vector format. The vector chunk can be directly processed by the vectorized function above.

- *Caller Site Rewriting* Change the original application code containing `lapply` into the new code containing `lapplyPartition`, and perform optimizations to reduce the overhead.

5.3.1 Function Vectorization

The function vectorization in this context is exactly the same as the function vectorization in Section 4.3.3. The same algorithm and the same API `VEC_FUN` is used here.

5.3.2 Data Transformation

The data transformation transforms the whole RDD from the original list storage into the vector storage. The original list RDD in SparkR is already partitioned, and each R worker instance owns one partition, shown as the upper part of Figure 5.4. After the data transformation, the data should be expressed as the lower part of Figure 5.4.

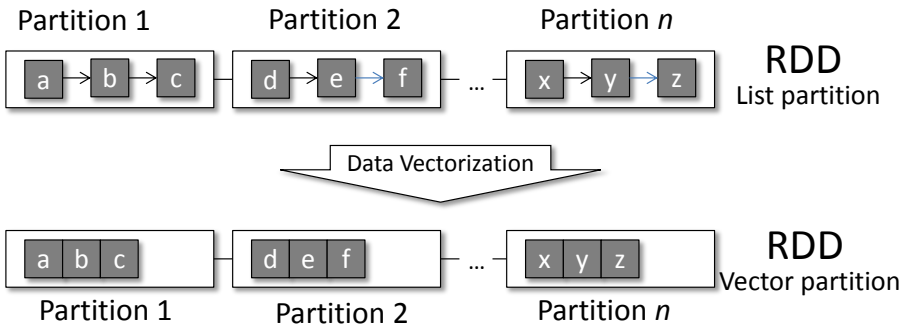


Figure 5.4: RDD Data Vectorization

The individual partition's data transformation (e.g. Partition 1 in Figure 5.4) is exactly the same as the data transformation in Section 4.3.3. The algorithms there and the APIs including `PERM_DOWN`, `PERM_UP` and `REP_EXPAND` work in the same way. However, from the global RDD view, these functions should be applied to all the partitions. The SparkR's `lapplyPartition` is used here to perform this task. For example, in order to transform the `YX` object from a list RDD into a vector RDD, an expression `lapplyPartition(YX, PERM_DOWN)` is generated. The other two transformations, `PERM_UP` and `REP_EXPAND`, work in the similar way.

5.3.3 Caller Site Rewriting

The caller site rewriting in Section 4.3.3 transforms the `lapply` function call into a direct function invocation. The transformation in SparkR's context is slightly different. It translates the `lapply` code into a `lapplyPartition` call. For example the original expression of `lapply(data, foo)` will be changed as the code in Listing 5.2.

```
1 lapplyPartition(
2     lapplyPartition(
3         lapplyPartition(listData, PERM_DOWN),
4         VEC_FUN(foo)
5     ),
6     PERM_UP
7 )
```

Listing 5.2: Expression generated from `lapply(data,foo)`

The `lapplyPartition` of Line 3 transforms the data from the list partition data to the vector partition data. The `lapplyPartition` of Line 2 invoke the real function with the vectorized function `VEC_FUN(foo)`. The `lapplyPartition` of Line 1 transform the vector partition data back to the original list partition to maintain the original program behavior. When the transformed code is launched, the SparkR's runtime will be responsible for scheduling each partition to the corresponding work and getting the right result back.

5.3.4 Other Transformations

Reduce Rewrite In many applications, the result of `lapply` is fed into the following `reduce` function call. In the standalone single node R's context of Chapter 4, the following `reduce` function invocation is also transformed into a more efficient reduce call. This technique can be applied to the distributed context here, too. But because the `reduce` will go through the whole RDD object, a direct function replacement is impossible. A two level reduce mechanism is used. In the single R instance level in each worker, a more efficient reduce function (for example, `colSums` to `sum`) is used to do the reduce over the DOWN shape result from `lapplyPartition`. After

that, the original `reduce` function is used to perform the cluster level reduce with the original reduce operator. With the two level reduce mechanism, the `lapplyPartition` of Line 1 in Listing 5.2 could be saved.

`lapply` for Non RDD object call The code transformation proposed in this section is a pure compiler based static transformation. The `lapply` overrides the R base package's `lapply`, which means the `lapply` can receive either an RDD object or a normal R object. And this type information cannot be checked in the code rewriting phase. The current implementation of the vectorization transformation dynamically checks whether the SparkR package is loaded. If loaded, the `lapply` will be transformed into `lapplyPartition` style. Furthermore, a new `lapplyPartition` is also defined in the base package, which performs the direct function invocation. With this runtime change support, even the `lapply` receives a non-RDD object in the SparkR context, the generated `lapplyPartition` code can still get the right result.

5.3.5 Optimizations

All the optimizations described in Section 4.3.3 can be applied to the SparkR's context.

- *Runtime Memorization* Each RDD object has the UP and DOWN data representation. A runtime memorization mechanism based on hidden variables are used to do the runtime linkage to save the redundant data transformation.
- *Delayed Evaluation* The *delayedAssign* mechanism of R is used to delay or even remove the data transformation from the DOWN shape back to the UP shape if the UP shape object is not used afterwards.
- *Remove Redundant Data Replication* It is the same as the redundant data replication removing mechanism mentioned in Section 4.3.3, and only works in each single worker's R context.

- *Optimize Reduce Function Call* Similar to the reduce function call optimization in Section 4.3.3, some specific reduce operation, such as `sum` can be replaced by an efficient high dimension function call, like `colSums`. This optimization can only be applied at each single worker's R context, because the cross cluster's reduce must be performed one by one.

Another optimization here is using the *cache* mechanism of Spark to cache the transformed DOWN shape object in the memory of each worker's Spark executor. This will save a lot of memory serialization/de-serialization and data communication overhead.

5.3.6 Code Transformation Example - Linear Regression

With all the code transformation described in the previous section, the Linear Regression example in Listing 5.1 is transformed as the code in Listing 5.3.

Line 15 to 21 of Listing 5.3 is the runtime memorization optimization. The `cache` in line 17 is the Spark object cache optimization. Line 22 will be invoked at each worker's R instance to do function vectorization transformation, which will generate the same code as Listing 4.6. Line 24 is the *delayedAssign* optimization. Line 26 to 30 is the two level reduce in SparkR, and line 27 to 28 is the reduce optimization in each single R instance level. With all this transformation and optimization, the code in Listing 5.3 runs much faster than the original code in Listing 5.1.

5.4 Evaluation

The performance of the SparkR with the vectorization integration was evaluated by comparing the running time of the vectorized code to the time of the original SparkR execution.

5.4.1 Benchmarks

The benchmarks in the evaluation are kernels of data analytics and machine learning algorithms similar to those used in Chapter 4. Because both Spark and SparkR are mainly target solving

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

sc <- ... #A Spark context
YX <- ... #A RDD list in the sc context. Each element is a [y x] vector
alpha <- ... # Control the changes in each iteration

for(iter in 1:niter) {
  .va.delta <- lapplyPartition(
                { if(!exists(".va.YX", inherits = FALSE)
                     || !identical(.vasrc.YX, YX)) {
                    .va.YX <- cache(lapplyPartition(YX, PERM_DOWN))
                    .vasrc.YX <- YX
                  }
                  .va.YX
                },
                VEC_FUN(grad.func) )

  delayedAssign("delta", lapplyPartition(.va.YX, PERM_UP))

  theta <- theta - alpha * reduce(
             lapplyPartition(.va.delta,
                             function(vData){list(colSums(vData))},
             '+')
}
```

Listing 5.3: Transformed Linear Regression with lapply in SparkR

iterative problems, all the iterative benchmarks in Chapter 4 are used in this evaluation. The benchmark collection also includes *k-NN* problem, which is a direct algorithm based benchmark. It is used to show the effectiveness of the vectorization can also be applied to direct algorithm in SparkR's context. The implementation of these application is almost the same as the code used in Chapter 4 except some minor changes to create SparkR context and use SparkR's API.

Table 5.1 lists the kernels and the configurations used in the evaluation. Because the goal of SparkR is solving Big Data problem, the input sizes to these benchmarks vary in the evaluation, starting from 1 millions samples for all iterative algorithms, and increasing up to 16 million samples. The input of k-NN starts from 10k testing samples, and increases up to 160k samples.

Table 5.1: Benchmarks and Configurations

Name	Descriptions	Configurations
LR	Linear Regression of n variables	Each sample 10 dimensions
LogitReg	Logistic Regression of n variables	Each sample 10 dimensions
ICA	Independent Component Analysis	Un-mixing 2 signals
K-Means	Clustering of n dimensional points	3D points, 10 clusters
k-NN	k Nearest Neighbor	3D points, 10k training, 10 categories, k=5

5.4.2 Evaluation Environment

The evaluation was performed on the campus cluster [10] of University of Illinois at Urbana-Champaign. The detail hardware and software configuration can be found at the website of the campus cluster. The Golub cluster was used in all the following measurement. The head node is equipped with one Intel E5-2660 2.2GHz 8-Core processor and 128GB memory. Each computing node is equipped with one E5-2670 2.6GHz 8-Core processor and 64GB or 128GB memory[1]. The software configuration is listed in Table 5.2.

Table 5.2: Software configuration in SparkR evaluation

Name	Version
OS	Cent OS 6, 2.6.32-504.8.1.el6.x86_64
Java	Sun Java 1.8.0_31-b13 64bit
Scala	2.11.4
Spark	1.1.0 pre binary built
R	3.1.2
SparkR	Github head build v2271030

Up to eight computing nodes were used in the evaluation. Because each node has a 8-core processor, each Spark executor can launch eight R instances in one computing node and there are total **64 R instances** in the evaluation at the maximal scale.

5.4.3 Evaluation Methodology

There are two variables changed in the evaluation, the input data size, and the number of computing nodes (workers) used in SparkR. The input data size will vary from 1 million to 16 millions for the

[1]The resource scheduler of the campus cluster chooses the available 64GB or 128GB machines at the request time. But the application use less than 64GB memory

iterative algorithms and 10k to 160k for k-NN, and the number of computing nodes will vary from 1 worker (8 R instances) to 8 workers (64 R instances) in the evaluation.

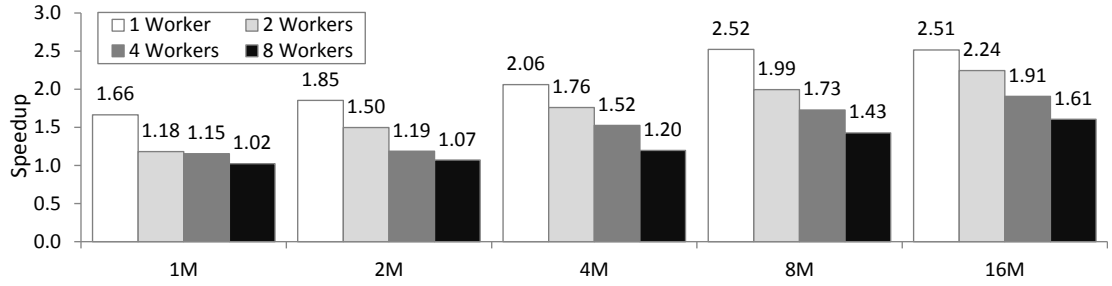
For iterative algorithm based benchmarks, the computation time is defined as one iteration's computation time. In the test, the time is calculated by the average time of iteration 6 to iteration 15, when the system runs in a stable stage. Because of the runtime memorization optimization, different to the evaluation in Chapter 4, one iteration's computation time except the first iteration doesn't contain the input data's transformation time. Because the input data's transformation time could be amortized by the iterations, this method to calculate the time is reasonable. The k-NN's computation time is the total time of the computation, which includes the data transformation time. For a direct method, data transformation time cannot be amortized, so the overhead is incorporated in the evaluation.

The following subsection will report the speedup number of the SparkR with vectorization versus the original SparkR, as well as the absolute running time in different input sizes.

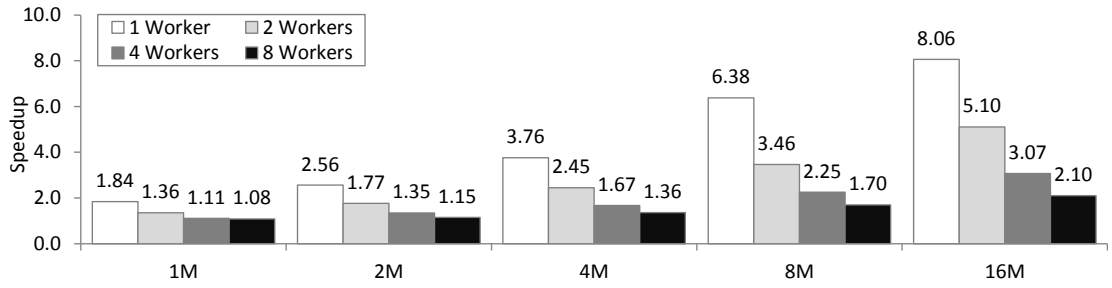
5.4.4 Vectorization Speedup

Figure 5.5 shows the speedup number of the SparkR with vectorization integration to the original SparkR. The vectorization can improve the performance in almost all the cases, and can achieve over 10x speedup in many cases. The speedup number are impacted by the workload, the input size, and the number of workers.

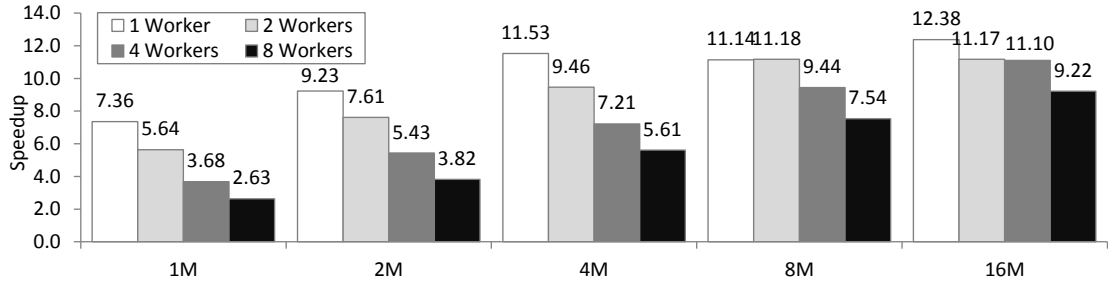
First, the more complex the computation of the workload, the higher the speedup. Although all these algorithms can achieve very good speedup in single node's standalone R (Section 4.5), the final speedup number in SparkR is not only impacted by the algorithms' computation time, but also by the SparkR's scheduling overhead.



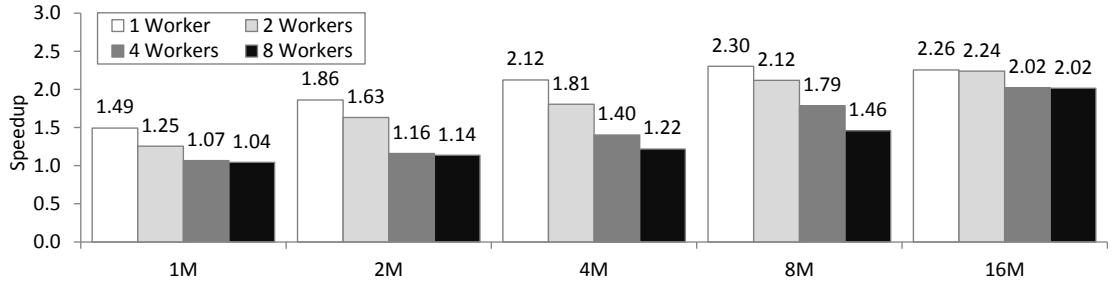
(a) LR



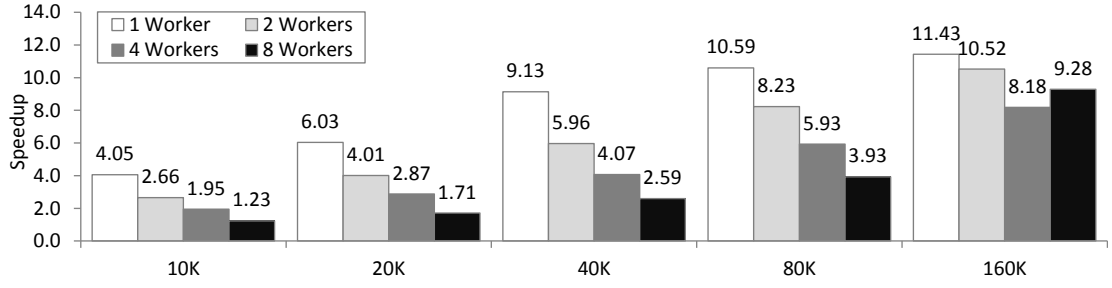
(b) LogitReg



(c) k-Means



(d) ICA



(e) k-NN

Figure 5.5: Speedup of vectorization in SparkR

For example, k-Means and k-NN has the most complex computation. Both k-Means and k-NN have a lot of distance computation, and finding minimal location operations, which require huge amount of computation power. The vectorization can greatly save these operations. And the SparkR's scheduling overhead is relatively small compared with the computation time. As a result, these two benchmarks can still get good speedup numbers. However, LR and ICA has small computation in each iteration, and the SparkR's scheduling overhead is relatively large in these cases, which explains the low final speedup numbers.

Secondly, the larger the input size, the higher the speedup. If the number of the workers is fixed, the speedup number increases as the input size increases. This is still caused by the SparkR's scheduling overhead. In a simple model, SparkR's scheduling overhead is a constant. The larger the input size, the larger the computation, and the relatively smaller the overhead. So the speedup number is higher.

Thirdly, the more workers, the lower the speedup. If the input size is fixed, as the number of the workers increases, the speedup number drops. The reason is the same as above. As the number of worker increases, each R instance in SparkR gets smaller portion of computation, and each worker has a relatively larger overhead. So the speedup number drops.

Although SparkR is a distributed system, it still follows the basic rules of parallel computing systems. It can be analyzed with these parallel models, such as Amdahl's law or Isoefficiency[41].

5.4.5 Comparing with Manually Transformed Vector Code

Because SparkR provides the API `lapplyPartition`, user can use the API directly with a vector functions and vectorized RDD object. The Logistical Regression example in SparkR project is written in this way. Here the performance of the automatic vectorized LogitReg (with `lapply` invocation) is compared with the manually written vector version of LogitReg (with `lapplyPartition` invocation), and the speedup number is shown in Figure 5.6.

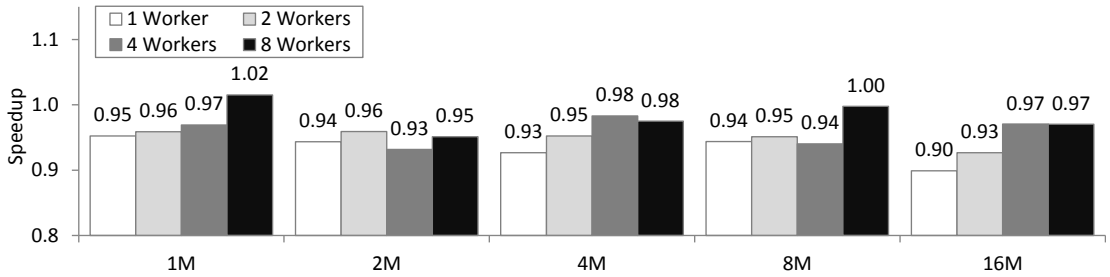


Figure 5.6: Speedup of the vectorized LogitReg to the manually written vector LogitReg

The automatic vectorization technique proposed in this section can achieve the similar performance as the manually written vector version, only with a small gap (maximal 10%). One reason for the gap is the manually written version uses matrix-matrix multiplication to perform a sum and a reduction combination, while the generated code from automatic vectorization uses one matrix element wise multiplication and a reduce operation to get the same effect. But matrix-matrix multiplication can only be performed in a single R instance, and there are still cross R instance reductions. Because the effect, when the number of R workers increases, the gap is reduced, since both the manually written version and the automatic generated version have more cross-nodes reductions, which have relatively larger overhead.

5.4.6 Comparing with Single Node Standalone R

The performance of the benchmarks used in SparkR evaluation was also measured in the standalone R with the same hardware environment. This subsection reports the performance of these benchmarks with SparkR and with a standalone R. The standalone R only uses one single thread, where each SparkR's worker can launch 8 R instances to run in parallel. However, because of the Intel Turboboost feature in the processor of the testing environment, the standalone R may runs at a CPU frequency upto 3.3GHz, while the R instance in SparkR can only runs at 2.6GHz. Due to the resource control limitations, the Turboboost cannot be disabled in the testing environment. The absolute number reported in this section may be biased, but the trend should be valid.

Speedup to Standalone R

Figure 5.7 shows the speedup number of SparkR without vectorization to the standalone R without vectorization. SparkR can achieve speedup in all cases except the k-Means. Because one worker in SparkR has 8 R instances, one worker can achieve over 1 speedup in many cases. The k-Means' performance is relatively low in SparkR. SparkR only get little speedup, or even slowed down the computation if the input size is relatively small. The reason may be from the *reduce by key* implementation in SparkR, which is out the scope of this thesis. For all the other cases, the parallelism improved the overall performance, and in k-NN's case the speedup received up to 43x with total 64 R instances.

Figure 5.8 shows the speedup number of SparkR with vectorization integration to the standalone R with vectorization. The speedup number is much smaller compared to the case without vectorization. The reason is that after vectorization, the computation is greatly reduced, and the overhead of SparkR is relatively much larger in this case. So the speedup to standalone R drops. But because each R instance's computation is reduced, the whole system can process more data. According to the basic parallel computing rules, if the input size increases, SparkR with vectorization should be able to get relatively good speedup.

Running time of Different Problems Size

Figure 5.9 reports the absolute running time of all the benchmarks in standalone R and SparkR (with different workers). The running time for iterative algorithms are one iteration time and all computation time for k-NN, which are defined earlier in this section. The absolute running time gives a better view to explain the high or low speedup number. k-Means and k-NN has very long computation time, which explains both the SparkR and the vectorization in SparkR can help achieve good speedup. Others have relatively small computation. Without the vectorization, the computation time is still relatively large, then SparkR can reduce the total running time compared with the standalone R. With the vectorization, these benchmarks only spend a few seconds in

one iteration, which is in the same scale of the SparkR's scheduling overhead, and SparkR with vectorization sometimes even run slower than standalone R with vectorization.

5.5 Discussion

As discussed in Section 5.4, `Apply` vectorization can effectively improve the performance of SparkR in many cases. The speedup number could be higher if the input size is large or there are few workers (where each worker can receive more input data). However, all the speedup number reported here are much smaller than the number reported in Chapter 4. The main reason is the overhead introduced by the parallel system.

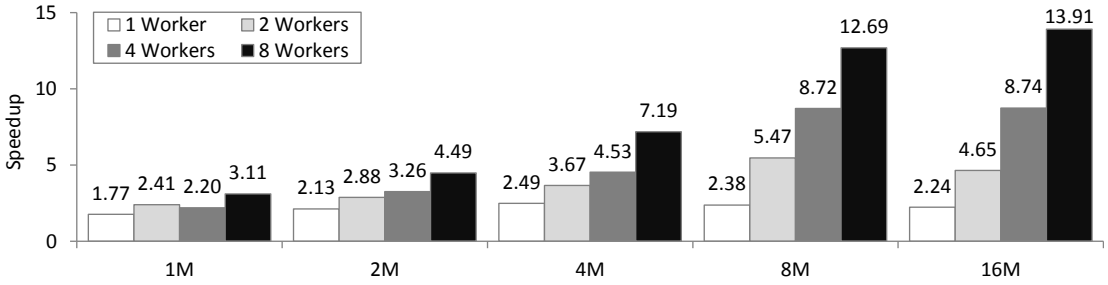
Based on the analysis of SparkR's implementation, two important source of the overhead have been identified.

- *Data exchange between Spark executor and R instance* Because Spark framework is based on Java, and R is a standalone application, there must be some data exchange between the Spark executor and the R instance in each worker, as shown in Figure 5.3. The current data exchange mechanism in SparkR is based on object serialization through disk. Spark executor only stores R objects as binary-array. If the data object should be passed from the Spark executor to the R instance, the object is first written in to the disk as files, then R instance reads the files and do object de-serialization. There are two disk operations, Java write and R read, and one object de-serialization, which are all high overhead operations. The communication from R instance to Spark executor has the same amount of work.

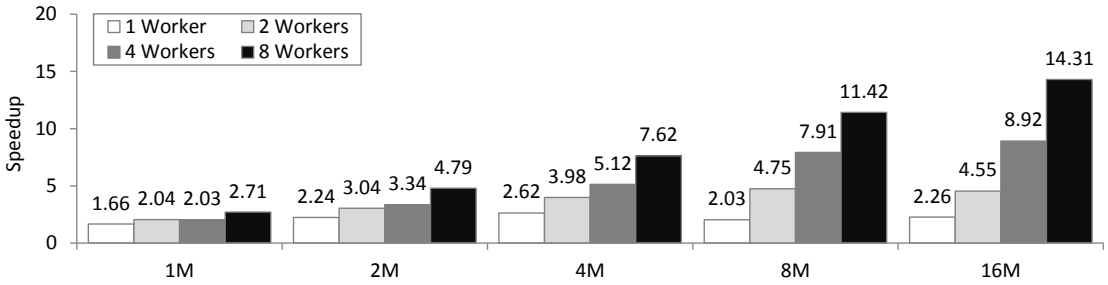
- *Launching R instance for each R blocking operation* The current implementation of SparkR uses a new R instance for each R operation scheduled to the worker. This is a typical design for most of the distributed frameworks, such as Hadoop[9]. Spark uses lazy evaluation to fuse a few R operations into a single large R operation, and only evaluates it if the operation will cause blocking to others. For example, a few piped `lapply` will be fused together,

until a `Reduce` is reached. However, for iterative algorithms, each iteration will have one fused operation at least, and Spark executor will create a new R instance for it. For a new R instance, it must first communicate with the Java Spark executor to get the data, and communicate with the Java Spark executor to store the data for the next round's iteration. These communication have very large overhead as described before. The R instance launching operation itself also brings large overhead.

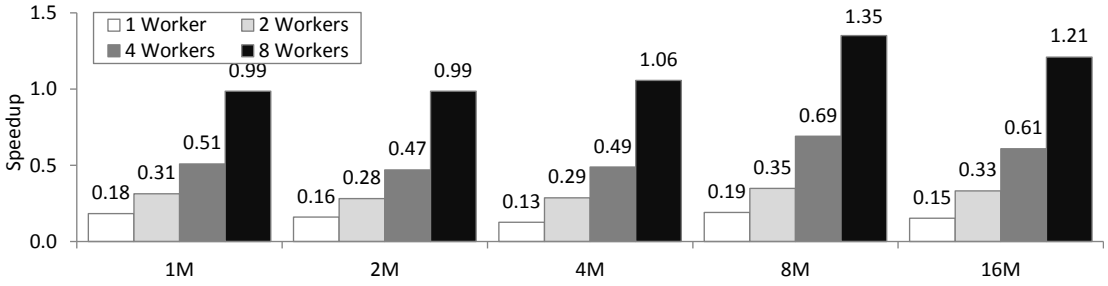
There are many possible ways to reduce the two overhead mentioned above, for example designing an effective Java/R interaction mechanism through memory, implementing a better object serialization for R, or implementing R instance reuse across different R operations. All these techniques are out of the scope of this thesis. But in order to improve the whole system's performance, these optimizations must be incorporated, otherwise the vectorization's benefit is greatly reduced in SparkR.



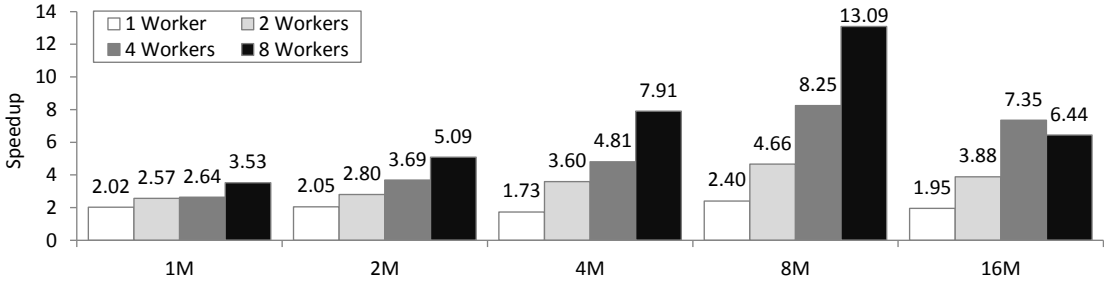
(a) LR



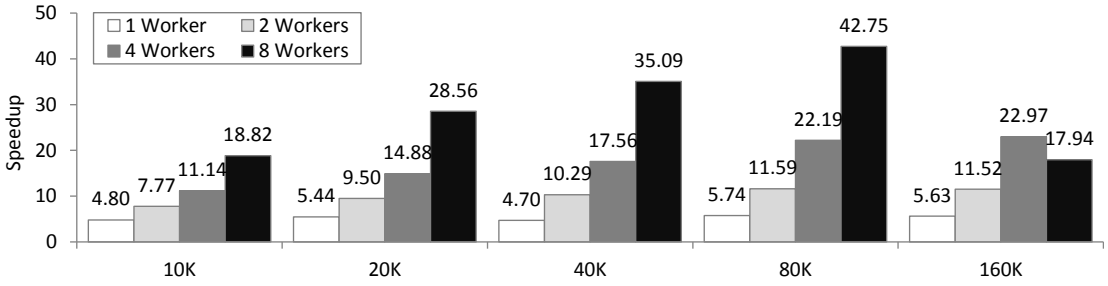
(b) LogitReg



(c) k-Means

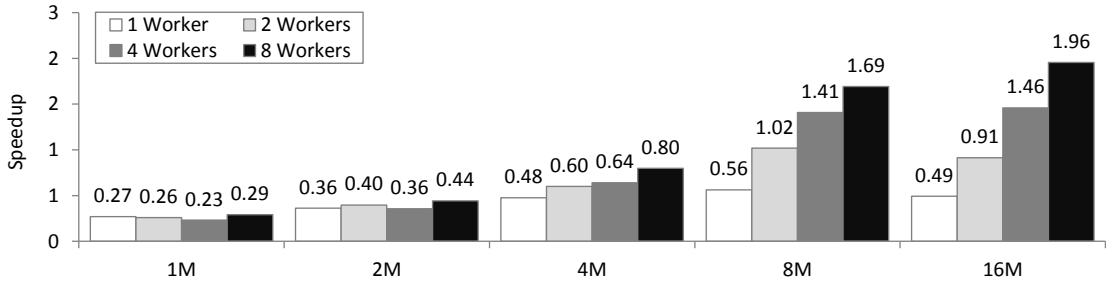


(d) ICA

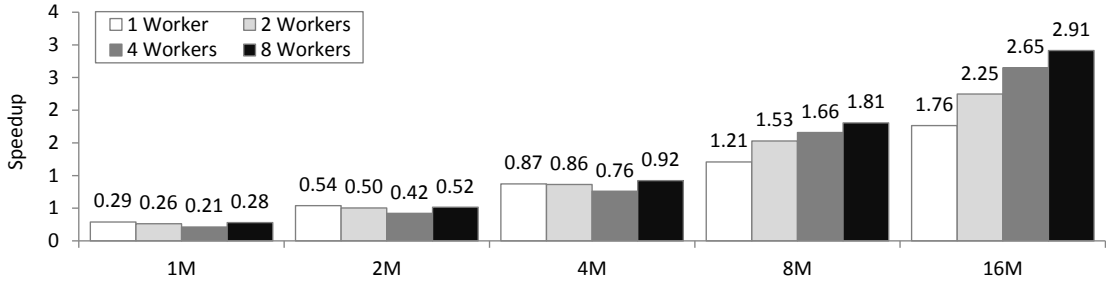


(e) k-NN

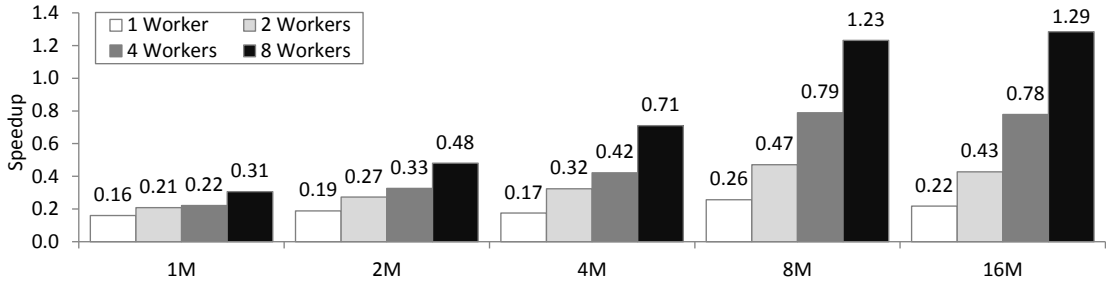
Figure 5.7: Speedup of SparkR to standalone R



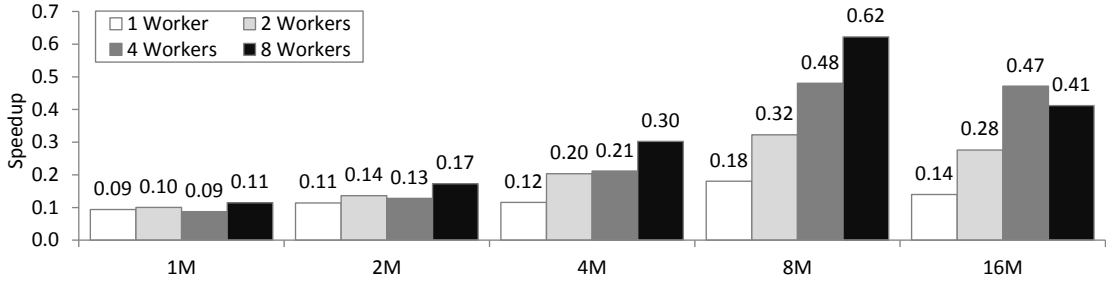
(a) LR



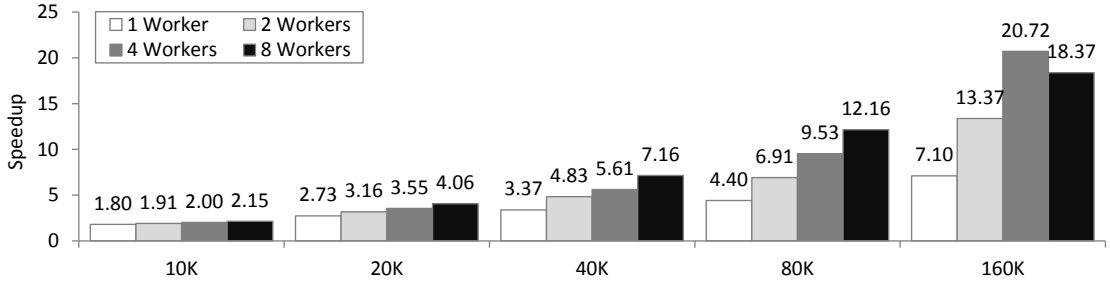
(b) LogitReg



(c) k-Means



(d) ICA



(e) k-NN

Figure 5.8: Speedup of vectorized SparkR to vectorized standalone R

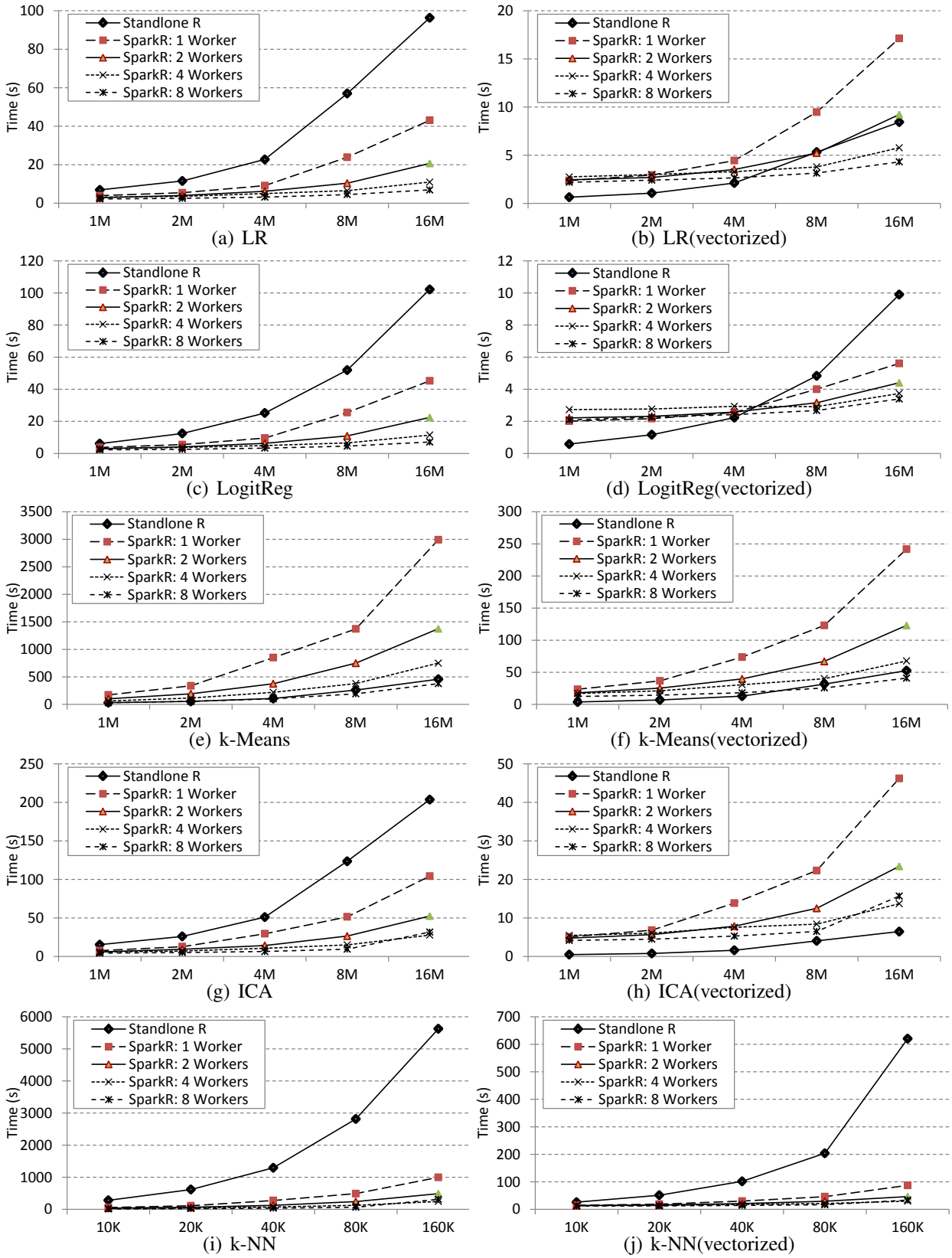


Figure 5.9: Absolute running time(second) of standalone R and SparkR

Chapter 6

R Benchmark Suite

6.1 Introduction

This thesis has discussed some techniques to improve dynamic scripting languages, specifically R language's performance. For all performance improvement related research, a good set of benchmarks is required to identify the performance problems and measure the effectiveness of different optimization methods. Many benchmarks available in system level and static languages' research, such as the SPEC benchmark collections[16], the LINPACK benchmark for HPC, the DaCapo benchmark[28] for Java, as well as in dynamic scripting languages area, such as V8 benchmark[18], SunSpider[17] and JSBench[5] for JavaScript.

However, there is no proper benchmarks for R. Different research teams working on R's performance improvement use their own small collections of R benchmarks. There are two problems. It's not clear whether these benchmarks are representative. and it's hard to compare the result of different research techniques in a common set of benchmarks.This problem is an obstacle not only to the research of this thesis but also to other researchers in R research domain.

Besides the problem of no proper benchmarks, another problem is lacking of benchmarking environment to automatically measure the performance of different benchmarks. The naive way to measure the performance is through R's `system.time()` interface, which reports the running time of the input expression. Other R packages, such as *rbenchmark*, *microbenchmark*, provide additional interfaces to repeatedly run the input function and do simple data analysis. However, all these require user's explicit invocation. All the measurement are inside R process, and can only report the execution time. Furthermore, some research R VMs are not compatible with GNU R,

and cannot use these benchmark packages.

The R benchmark suite work described in this chapter are trying to solve the two problems. First, a bunch of R micro-kernels and real application problems are collected, and standardized. They are organized according to the taxonomy system defined in this thesis (Type I, II, and III, see Section 2.2.1). Second, a flexible R benchmarking framework was design and implemented. It can automatically measure the performance not only GNU R but also all available research R VMs with the benchmark collections mentioned. With the extensible design, it can not only report the time, but also report the software and hardware performance metrics from OS and hardware performance counters.

The rest of this chapter is organized as follows. Section 6.2 describes the benchmark collections. The design and implementation of the benchmarking framework are explained in Section 6.3. The current status of the benchmark suite is briefly discussed in Section 6.4.

6.2 Benchmark Collections

A good benchmark collections should have the following features, representative to the real application, good coverage, self contained and easy to be measured. The benchmark collection described here are the collection of micro kernels, real applications and R version implementations of other benchmarks. Table 6.1 is a short list of these benchmarks.

Table 6.1: Benchmark Collections in R Benchmark Suites

Name	Short Description	Type
Shootout	R version ofComputer Language Benchmarks Game	Type I, II
R-benchmark-25	Also calledATT benchmark, Math computations	Type I, III
Scalar	Micro benchmarks	Type I
Mathkernel	Math kernels	Type I, II and III
Riposte	Vector dominated benchmark	Type II
Algorithms	Data Analytics problems	Type I and III
Misc	Some random collections	Type I, II and III

- *Shootout* It is the R implementation of the shootout benchmark[4]. The original implementations are from the ORBIT research work described in Chapter 3, and FastR[52][51]. The

code mainly uses the Type I style. But some applications also have Type II implementation to get better performance.

- *R-benchmark-25* It is also named as ATT benchmark[1]. It contains 15 micro math kernels in three groups (Matrix_calculation, Matrix_functions and Programmation). The code either uses Type I or uses Type III direct native function invocations. It has been refactorized here so that each math kernels can be measured individually.

- *Scalar* It is a collection of micro benchmarks, which include `CRT` (Chinese Reminder Problems), `Fib` (Fibonacci number), `Primes` (Finding prime numbers), `Sum` (Accumulation based on loop), `GCD` (Greatest Common Divisor). It was used in the ORBIT research in Chapter 3.

- *Mathkernel* It includes some basic math kernels such as Matrix-matrix multiply, Vector add, etc. It has Type I, Type II and Type III implementations for each math problem so that the researcher can compare them in the same context.

- *Riposte* It was used in Riposte [75] , and contains several vector computing dominated benchmarks.

- *Algorithms* It contains data analytics problems, such as Linear Regression, Logistical Regression, k-Means, k-NN, etc.. It was used in `Apply` vectorization research described in Chapter 4 and Chapter 5. it was also used in Rabid[56] and SparkR [82]. The collection also contains the Type III implementations, which directly invoke the R's built-in functions, such as `lm()` for Linear Regression, and `kmeans()` for k-Means.

- *Misc* It contains random collections of some real R applications, such as 2D Random Walk.

The classification of R codes into Type I, II, and III in Section 2.2.1 is the first step to classify the understanding of R performance, so all these benchmarks in the collection are labeled with Type I, II, III to get a good coverage of all different types of R programs. These applications have

been standardized to follow the interface defined in Section 6.3, so that the benchmark driver can launch them and measure them with different benchmarking approaches.

6.3 Benchmarking Environment

As described in the introduction section, another problem for studying R's performance is the lack of R benchmark environment. Currently, the way to do performance measurement is calling R's `system.time()` interface, or using some similar interfaces in other R packages. User may write R script to automate the process, but some research R VMs have different command line invocation interfaces. In order to measure the performance, user should customize the script invocation for each R VM. It's very tedious to compare the performance of all the R VMs in a fair and stand way. Furthermore, the as-is measurement can only provide the time information. Sometimes, it's very useful to capture the OS level's metrics (such as page fault rates), or processor level's metrics (such as instruction cache miss rate or data cache miss rate). There is no such support at all.

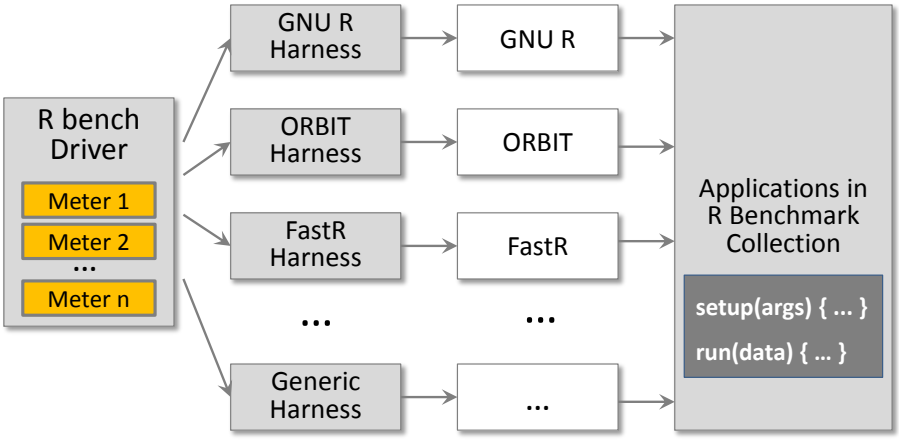


Figure 6.1: Benchmarking Environment

In order to solve these problems, a flexible R benchmarking framework was designed and implemented. Figure 6.1 shows the architecture. All the grey box components are part of the chapter's work. There are three layers in the framework, the benchmark interfaces (right), the harness (middle), and the driver (left).

6.3.1 Application Interface

The applications in R Benchmark Collections follows the code skeleton in Listing 6.1. The `setup()` function is optional. It parses the command line's arguments, and generate the input data. It makes sure the application is self contained. The `run()` function is the main measurement target, which will be invoked in the performance benchmarking. The code in Line 10 to 13 are helper code. With this helper code, the application can be executed by a standalone R without the benchmark driver. For example, using command line command `Rscript example.R` to run the benchmark `example.R`. The benchmark driver will define the variable `harness_argc`. If no such variable (in standalone execution mode), the `run()` can still be invoked with the input data from `setup()`.

```
1  setup <- function(cmdline_args){
2    ... # generate input with cmdline_args
3  }
4
5  run <- function(input) {
6    ... # application logic with input
7  }
8
9
10 if (!exists('harness_argc')) {
11     input <- setup(commandArgs(TRUE))
12     run(input)
13 }
```

Listing 6.1: Interfaces for an Application in R Benchmark Collection

6.3.2 Benchmark Driver

The benchmark driver is used to control the benchmarking for different R VMs. It will first launch the R process, then invoke the `setup()` method to get the benchmarking input, and invoke the `run()` method with the generated data from `setup()`. If the benchmarking only wants to know the the execution time of `run()`, the benchmark driver only needs to insert the time measurement around the `run()` function. However, some system level metrics cannot be easily measured, such

as the hardware performance counter. In many cases, the measurement can only be performed in the whole R process' level. As a result, the value captured contains not only the `run()` method execution, but also the R process launching and quiting, and the `setup()` execution, as shown in the upper part of Figure 6.2. The benchmark driver here used the delta approach, shown in Figure 6.2. It will invoke the application twice with different times of executing `run()`, and the difference of the two runs only contains the execution of `run()`. User can control how many iterations as the warm-up phase, and how many iterations as the steady measurement phase.

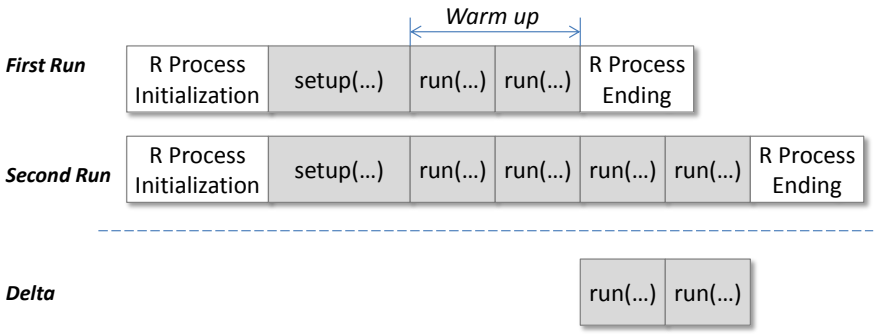


Figure 6.2: The Delta Approach in the Benchmarking

Many meters can be plugged into the driver, including the time, OS level counters and hardware performance counters (through Linux perf tool). Because of the delta measurement approach, new meters can be added into the benchmark driver without modifying the applications in the benchmark collections.

6.3.3 Benchmark Harness

The benchmark harness acts as an adapter to transform the control commands from the benchmark driver to the real R VM. Because different R VMs have different command-line invocation parameters, there are different harnesses. The harness also perform other environment preparing work and patch work. For example, ORBIT requires an internal interface be invoked to start the profiling and JIT optimization, the harness of ORBIT will turn on it before the invocation of `run()`.

There are many harnesses developed in the R benchmark suites, which can cover all the available R VMs in industry and academia. A list of the R VMs will be described in Chapter 7.

6.4 Discussion

The R benchmark suite work has been launched as an open-source effort, and can be found at [11]. Although this work is an engineering oriented effort, it is very important for the research work described in previous chapters. The work simplifies and formalizes the performance measurement work so the the techniques proposed in this thesis can be measured more precisely, and compared with related work fairly.

Furthermore, the benchmarking framework is very flexible. It is implemented in Python, and can measure not only R application, but also any shell invocable applications, such as Python script, C application, etc. It can be easily extended to benchmarking other dynamic scripting languages.

Chapter 7

Related Work

7.1 Optimizing R Language

7.1.1 The Landscape of Existing R Projects

Improving the performance of R is the focus of many research projects. Figure 7.1 summarizes all the major projects on improving R performance through JIT- or Virtual Machine (VM)-level optimizations today. These project are classified according to the R programming styles they target(x-axis) and the compatibility with the GNU R VM (y-axis). Since there is no formal specification of R, the GNU R VM is considered the de facto specification of the language. Such a phenomenon is quite common in scripting languages (JavaScript being one exception) and is sometimes known as *"language specification by implementation"*.

The projects shown at the top half of Figure 7.1 all build their own R VMs with independently designed object models and external interfaces (if supported) that are incompatible with the GNU R VM. The compatibility here is defined as following the memory object model of GNU R (Section 2.2.2), and provides the same interfaces that GNU R exposes. The projects shown at the bottom of Figure 7.1, on the other hand, are compatible to GNU R VM, which means they are extensions/variations to GNU R implementation.

7.1.2 Building new R Virtual Machines

Building a new R VM and restricting the language are popular approaches because much of the overhead of R comes from either the GNU R VM implementation (e.g., the object model or stack

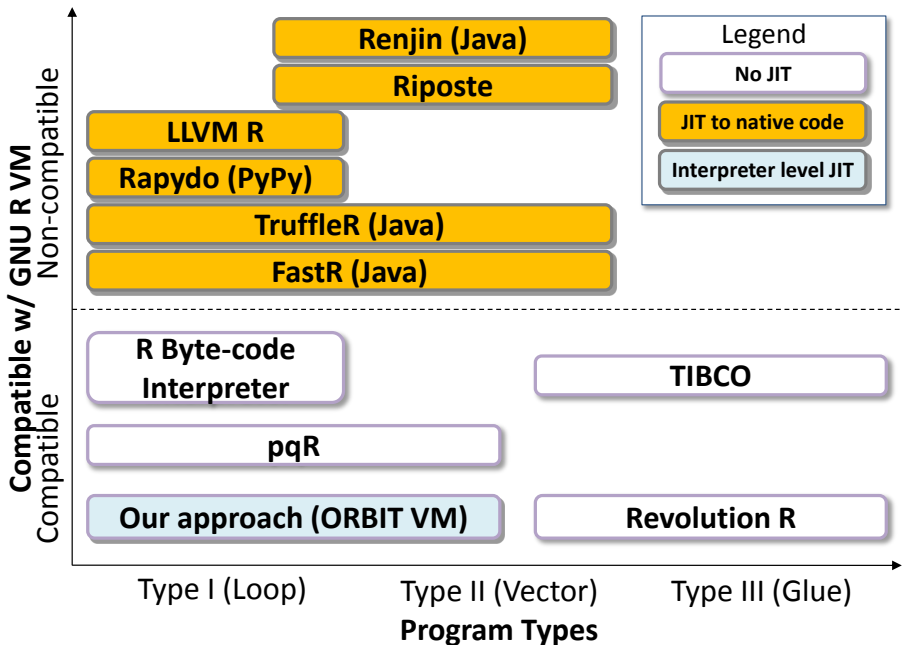


Figure 7.1: Landscape of R optimization projects.

frame design) or language semantics (e.g., lazy evaluation or Value Object, no reference). By developing a brand new R VM and excluding some of the rarely used but expensive language features, one can significantly boost the performance of certain Type I and II codes.

Several Research projects belongs to this category implement their brand new R VMs in Java, such as Renjin [8], fastR [52] [51], and TruffleR [46]. Renjin [8] implemented an R AST interpreter on top of JVM. The AST interpreter is a mimic to the GNU R interpreter, and tries to follow the semantics and behaviors of GNU R. It also applied code fusion and lazy evaluation optimizations for efficient processing Type II vector code. FastR [51] used the self optimizing AST interpreter approach from Truffle [86], and built a new JVM based R interpreter. It also applied data and code specialization optimizations. TruffleR is the next generation of FastR. It uses the same self optimizing AST interpreter approach from Truffle, and applied additional optimizations such as partial evaluation. It integrated with Graal[86] for native code JIT to get better performance.

Riposte [75] targets long vector R code (Type II). It achieved high speedup for long vector code through optimizations including trace JIT, vector code fusion, lazy evaluation, SIMD, and parallel execution. The latest version [76] implemented vector length specialization for short vector code.

Rapydo [44] is another experiment R VM based on the PyPy framework. It relies on the PyPy trace JIT engine and native code generation to achieve high performance. But it only supports a small portion of R semantics.

Rllvm [55] translates dialect of a subset of R constructs with type annotation into LLVM IR, and uses LLVM's optimizations and code generation to get executable native routines. However, it only supports few R operations.

Challenge for this kind of approaches is that R incompatible VMs cannot easily support Type III codes, which heavily depend on the internals of the GNU R VM. That means the thousands of packages available from public R repositories, such as CRAN (the resource hub for the R user community) [67] and Bioconductor [3], may not be able to run on these new R VMs. As a result, it's hard for these approaches gain high adoption in the R user community.

7.1.3 Extensions/Variations to the GNU R

The projects shown at the bottom of Figure 7.1, on the other hand, are compatible extensions to the GNU R VM. Most performance-conscious CRAN package developers provide highly optimized external packages for Type III R codes. In contrasts, it is a lot more challenging to optimize Type I and II codes while preserving compatibility with the GNU R VM.

The R bytecode interpreter from the R core team [77] extended GNU R VM with a simple ahead-of-time compiler, a byte-code interpreter, and some compiler time and runtime optimizations, such as constant folding and index based variable lookup. It greatly improved R's speed compared with the default AST interpreter in scalar dominated code (Type I). And this interpreter was officially incorporated into GNU R VM since R-2.14.0.

Revolution Analytics [24] and TIBCO [50] are two commercial offerings of R that target mainly on Type III codes by optimizing native external packages of R. For example, Revolution Analytics R replaced the open source Math libraries in GNU R with Intel MKL library to get high performance in Type III code execution. It also rewrote some internal math routines to boost the performance for some Type II codes.

pqR [62] added many small improvements to GNU R, such as data sequence data type, more precise reference counting, parallel execution of vector computation. These optimizations improved GNU R's performance on some applications.

CXXR [70] is a special project that tries to rewrite GNU R with C++. The initial goal is code cleaning and refactoring to GNU R to provide a good code base for future development. Recently, it expands the effort to implement optimizations for better performance.

The techniques proposed in this thesis also belong to this category. ORBIT VM uses profiling driven approach to translate the R byte-code into a specialized byte-code to improve the running speed, and used data representation specialization to reduce the memory usage. The `Apply` vectorization translates Type I R code into Type II R code to reduce interpretation overhead. There is no similar approach in related work. Furthermore, the approach does not modify GNU R implementation. The implementation is pure R based, and it can be used in not only GNU R, but also other R VMs.

7.2 Optimization in Dynamic Languages

Many techniques have been used in optimizing dynamic languages. Section 2.1.3 has described most of them. Here is a short list of the academic literature in this domain.

7.2.1 Interpreter

The classical way to build an interpreter for dynamic scripting languages is implementing an AST interpreter. But a simple AST interpreter's performance is very poor. Byte-code compiler and interpreter is used in Smalltalk[40] for efficient executing the program logic, and henceforth most scripting languages use byte-code interpreter. Later, many other optimizations have been proposed and applied to improve byte-code interpreter, including Threaded code[26][66], SuperOps[29], etc.. There are also some parallel efforts to improve the performance of AST interpreter, such as Truffle[86], who uses self-optimization approaches to specialize the AST with profiling informa-

tion.

7.2.2 Runtime Optimization

Specialization SELF[33, 32] is an early pioneer of using specialization optimizations. Many types of specializations are widely used now, such as *Type Specialization* MaJIC [23] for Matlab, [43] for JavaScript; *Function Argument Specialization* [71] for JavaScript; *Interpretation Specialization* [85] for Lua; *Vector Length Specialization*[76] for R.

Inline Cache and Hidden Class It was introduced to optimize the polymorphic procedures invocations in Smalltalk[37]. Later this technique has been widely used in optimizing any polymorphic resource accesses that have an indexable resource location, for example, accessing the object's field. Hidden Class was firstly used in SELF implementation[33] to accelerate the dynamic object's field access. Combined with Inline Cache, V8 JavaScript engine [19] successfully accelerated the object property access.

7.2.3 JIT Native Code Generation

Generating native code through JIT is a common approach to improve the performance of dynamic languages. Many research and commercial VMs for dynamic scripting languages uses native code JIT, such as Python[69, 31], JavaScript [39], LuaJIT [64], PHP HHVM[20].

7.2.4 Ahead-of-time (AOT) Compilation

Another approach to improve the performance of dynamic language is to do Ahead-of-time (AOT) compilation of a dynamic language into a low level static language, for example PHP phpc [27], HipHop[87], Matlab FALCON [36].

7.3 Performance Improvement through Vectorization

7.3.1 Function Vectorization

Chapter 4 introduced the approach that transforms a single object function used by the `Apply` class of operations into the corresponding vector version. It belongs to the function vectorization domain. [53] used the term *Whole-function Vectorization* to describe the transformation of a single object function to a vector function that accepts vector input and processes the data in a pure vector way. The paper proposed the algorithm to vectorize Open-CL kernels, and run them on CPU SIMD units or GPU.

Intel ispc compiler [65] used the similar approach to vectorize the innermost ispc parallel functions. [57] vectorized multiple map operations to take advantage of the Xeon Phi's wide SIMD unit. River Trail[47] extends JavaScript by adding new parallel array, and providing map, filter, reduce style operations. It vectorizes the function used by map operations for parallel execution in SIMD and GPU.

The concept of function vectorization in the above work is similar to the approach of Chapter 4. But the above work all target conventional static languages, where the data types and the operations in the single object function are relatively simple. This thesis' work supports all kinds of complex data types in R, which requires the data transformation that ensures the vectorized function accesses the data in a vector form, and transforms arbitrary operations with different schemes.

7.3.2 Vectorization in Scripting Language

Code vectorization for static languages have been studied and successfully used in commercial compilers for decades. [58] gave a detail survey and evaluation of the state-of-art automatic vectorization techniques in commercial compilers. In the dynamic scripting languages domain, [60] used automatic vectorization to translate scalar loop code of Matlab into vector code. But the major challenge of loop vectorization is data dependence analysis, which is very complex even for a

static language. The dynamic features of scripting languages make the problem even harder. As a result, vectorization of arbitrary code in the scripting language domain is not common.

7.4 Parallel and Distributed R Processing System

A single R instance has the problem of slow running speedup and memory limitation. Parallel computing is widely used with R to help it process BIG Data. [38] lists R packages, applications and runtime systems that leverage parallelism to improve R's capability.

7.4.1 Parallel R System

SNOW [79] provides the parallel version of `lapply` that distributes the data and computing to multiple R instances in the same machine or different machines via socket or MPI communications. *multicore* package[80] of CRAN uses the similar model in a shared memory machine, but provides more comprehensive interfaces. These two are all belong to the Master-Slave model described in Section 5.1. They have been merged together, and become the official *parallel* package in R.

ScaleR [25] is a commercial product that also follows the Master-Slave model. It uses Parallel External Memory Algorithms (PEMAs) to provide the feature to process large data set that cannot be fit into one machine's memory one time. It provides parallel algorithms of basic statistics computing and statistical modeling algorithms for the large data set, and offload them to multiple processes and process them in parallel.

7.4.2 Distributed R System

Another type of the parallel R system acts like the distributed runtime system described in Section 5.1. Rabid defines the distributed *List* type and *Table* type, and provides APIs like `lapply` and `aggregate` to work on the distributed objects. It integrates GNU R or Renjin with Apache Spark[12], and can solve very large data analytics problem with the simple interfaces. SparkR[82]

is a similar system that provides a light-weight front-end to use Apache Spark's distributed computation power from R.

RHadoop [68] provides the front-end of Apache Hadoop [9], and offers the Map-Reduce style interfaces to use the distributed R key-value pair object. RHIPE[42] is a similar system to expose the Map-Reduce computing of Hadoop to R. It also defines new operation model like Divide and Recombine.

Presto [83] defines the Distributed Array data type of R to express dense or sparse matrices in a distributed environment, and provides APIs to perform distributed matrix math operations. HP Distributed R [49] extended Presto and provides more comprehensive data types and operations.

Chapter 8

Conclusions

8.1 Summary

This thesis has studied the techniques to improve the performance of dynamic scripting languages. Different to previous work in improving dynamic scripting languages or R language, which may require huge amount of engineering work in the whole virtual machine system, this thesis has proposed some techniques that focus on interpreter level optimization, which are simple and portable approaches.

The strategy taken is a unique point in the space of possible R VM research work. First, it maintains the full compatibility with the GNU R VM. The approach approach is an extension to the GNU R VM, and does not change the R VM's internal data structure. It is worth pointing out that the compatibility requirement is not unique to R. Most scripting languages today, such as Python, Ruby, and Matlab, lack a formal language specification and has one dominant reference implementation that many legacy codes depend on. The performance improvement work must try to be compatible with the dominant implementation. Improving the performance of dynamic scripting language without scarifying compatibility is still an open question and has high research value.

Second the operates entirely within the interpreted execution. In ORBIT, the JIT compiler converts original R byte-code into more optimized forms that will in turn be interpreted. In `Apply` vectorization, the original code containing `Apply` class of operations are transformed to direct vector function invocations, which will be interpreted, too. Different to most of the existing approaches for optimizing dynamic scripting languages, these approaches do not generate native

code, but instead expands it with new specialized operations and transformations.

In summary, contributions of this thesis include:

- ORBIT Virtual Machine that uses profiling driven approach to translate R byte-code into a specialized byte-code to improve the running speed, and used data representation specialization to reduce the memory usage. It improved GNU R's performance without sacrificing the compatibility.

- `Apply` Vectorization applies data permutation and function vectorization to translates R `apply` class of operations from Type I codes to Type II codes. It greatly reduced the interpretation overhead without the need of VM changes or native code generation.

- Integration of `Apply` Vectorization and SparkR effectively improved the performance of SparkR. And this technique can be applied to other distributed R computing frameworks

- R Benchmark Suite provided a collection of R applications, and a extensible benchmarking environment that can analyze all kinds of R VMs to measure the performance and identify the performance bottleneck.

8.2 Future Work

This thesis applies the strategy of interpreter level specialization and practiced it in GNU R. It leaves much room for improvement.

Other specialization techniques could be applied, such as calling convention specialization for GNU R's heavy function call overhead. Furthermore the byte-code interpreter could be improved with techniques like register based byte-code interpreter. More object specialization should be introduced, like hashmap for local variable lookup, stack slots for passing argument lists, etc.

The performance of `Apply` vectorization is limited by the fall-back mechanism from no built-in vector function support and control divergence. Providing more vector functions for commonly used R operations and modifying the interpreter to support predicated execution can improve the

technique's performance. The performance can also be improved with more interpreter modifications, such as SIMD and parallel implementation of the vector operations.

The vectorization in SparkR achieved less performance gain compared with the single node's vectorization. In order to achieve the similar performance boost, system level refactoring in the distributed R system is required, including an efficient cross process communication mechanism, and re-use R instance mechanism for iterative algorithm based applications.

References

[1] R benchmarks, 2008. http://r.research.att.com/benchmarks/.

[2] Airline on-time performance, 2009. http://stat-computing.org/dataexpo/2009/.

[3] Bioconductor: Open source software for Bioinformatics, 2013. http://www.bioconductor.org/.

[4] The computer language benchmarks game (CLBG), 2013. http://benchmarksgame.alioth.debian.org/.

[5] Jsbench, 2013. http://jsbench.cs.purdue.edu/.

[6] Numpy, 2013. http://www.numpy.org/.

[7] The popularity of data analysis software, 2013. http://r4stats.com/articles/popularity/.

[8] Renjin: The R programming language on the JVM, 2013. http://www.renjin.org/.

[9] Apache hadoop, 2014. http://hadoop.apache.org/.

[10] Illinois campus cluster program, 2014. https://campuscluster.illinois.edu/.

[11] R benchmark suite, 2014. https://github.com/rbenchmark/benchmarks.

[12] Apache spark - lighting-fast clustering computing, 2015. https://spark.apache.org/.

[13] An introduction to r, 2015. http://cran.r-project.org/doc/manuals/r-release/R-intro.html.

[14] The r project for statistical computing, 2015. http://www.r-project.org/.

[15] Sparkr - r frontend for spark, 2015. http://amplab-extras.github.io/SparkR-pkg/.

[16] Standard performance evaluation corporation, 2015. http://spec.org/.

[17] Sunspider javascript benchmark, 2015. https://www.webkit.org/perf/sunspider/sunspider.html.

[18] V8 benchmark suite - version 7, 2015. http://v8.googlecode.com/svn/data/benchmarks/v7/run.html.

[19] V8 javascript engine, 2015. https://code.google.com/p/v8/.

[20] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014. ACM.

[21] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[22] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[23] G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI'02, pages 294–303, New York, NY, 2002. ACM.

[24] R. Analytics. Revolution R enterprise, 2013. http://www.revolutionanalytics.com/products/revolution-enterprise.php.

[25] R. Analytics. Revolution r enterprise scaler -transparent parallelism accelerates big data analytics easily, 2015. http://www.revolutionanalytics.com/revolution-r-enterprise-scaler.

[26] J. R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, June 1973.

[27] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009. ACM.

[28] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[29] P. Bonzini. Implementing a high-performance smalltalk interpreter with genbc and genvm, 2004. http://smalltalk.gnu.org/files/vmimpl.pdf.

[30] T. S. BV. Tiobe programming community index, 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[31] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'12, pages 195–212. ACM, 2012.

[32] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI'89, pages 146–160, New York, NY, USA, 1989. ACM.

[33] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA'89, pages 49–70, New York, NY, USA, 1989. ACM.

[34] J. Chambers. The s system, 2001. http://cm.bell-labs.com/cm/ms/departments/sia/S/index.html.

[35] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281–290, 2007.

[36] L. De Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, Mar. 1999.

[37] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[38] D. Eddelbuettel. High-performance and parallel computing with r, 2015. http://cran.r-project.org/web/views/HighPerformanceComputing.html.

[39] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI'09, pages 465–478, New York, NY, 2009. ACM.

[40] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[41] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, Aug. 1993.

[42] S. Guha. *Computing Environment for the Statistical Analysis of Large and Complex Data*. PhD thesis, Purdue University Department of Statistics, 2010.

[43] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI'12, pages 239–250, New York, NY, USA, 2012. ACM.

[44] S. Hager. Implementing the R language using RPython. Master's thesis, Institut Für InformaAtik. Softwaretechnik und Programmiersprachen. Düsseldorft Universitätsstr, 2012.

[45] B. Hamner. Tools used by competitors, 2011. Kagglers' Favorite Tools.

[46] M. Haupt, C. Humer, M. Jordan, P. Joshi, J. Vitek, A. Welc, C. Wirth, A. Wöß, M. Wolczko, and T. Würthinger. Faster fastr through partial evaluation and compilation. In *UseR 2014*, 2014.

[47] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: A path to parallelism in javascript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 729–744, New York, NY, USA, 2013. ACM.

[48] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[49] HP. Hp distributed r - an essential offering of hp haven predictive analytics, 2015. http://www.vertica.com/hp-vertica-products/hp-vertica-distributed-r/.

[50] T. S. Inc. TIBCO enterprise runtime for R, 2013. http://tap.tibco.com/storefront/trialware/tibco-enterprise-runtime-for-r/prod15307.html.

[51] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A fast abstract syntax tree interpreter for r. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 89–102, New York, NY, USA, 2014. ACM.

[52] T. Kalibera, F. Morandat, P. Maj, and J. Vitek. FastR, 2013. https://github.com/allr/fastr.

[53] R. Karrenberg and S. Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 141–150, Washington, DC, USA, 2011. IEEE Computer Society.

[54] J. Knaus. http://cran.r-project.org/web/packages/snowfall/index.html, 2013. http://cran.r-project.org/web/packages/snowfall/index.html.

[55] D. T. Lang. The Rllvm package, 2013. http://www.omegahat.org/Rllvm/.

[56] H. Lin, S. Yang, and S. Midkiff. Rabid – a general distributed r processing framework targeting large data-set problems. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 423–424, June 2013.

[57] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. In *Big Data, 2013 IEEE International Conference on*, pages 125–130, Oct 2013.

[58] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, 2011.

[59] A. McAfee and E. Brynjolfsson. Big data: The management revolution. *Harvard Business Review*, October 2012.

[60] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL'99, pages 53–65, New York, NY, USA, 1999. ACM.

[61] 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, Berlin, Heidelberg, 2012. Springer-Verlag.

[62] R. M. Neal. pqR - a pretty quick version of R, 2013. http://radfordneal.github.io/pqR/.

[63] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.

[64] M. Pall. The LuaJIT project, 2013. http://luajit.org/.

[65] M. Pharr and W. Mark. ISPC: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, 2012.

[66] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 291–300, New York, NY, USA, 1998. ACM.

[67] R. project. CRAN: The comprehensive r archive network, 2013. http://cran.r-project.org/.

[68] RevolutionAnalytics. Rhadoop, 2014. https://github.com/RevolutionAnalytics/RHadoop/wiki.

[69] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA'06, pages 944–953, New York, NY, USA, 2006. ACM.

[70] A. Runnalls. CXXR: Refactorising R into C++, 2011. http://www.cs.kent.ac.uk/projects/cxxr/.

[71] H. N. Santos, P. Alves, I. Costa, and F. M. Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO'13, pages 1–11, Washington, DC, 2013. IEEE Computer Society.

[72] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[73] D. Smith. The ecosystem of R. http://blog.revolution analytics.com/2011/08/the-r-ecosystem.html.

[74] I. Spectrum. Top 10 programming languages, 2014. http://spectrum.ieee.org/computing/software/top-10-programming-languages.

[75] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: a trace-driven compiler and parallel VM for vector code in R. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT'12, pages 43–52, New York, NY, USA, 2012. ACM.

[76] J. Talbot, Z. DeVito, and P. Hanrahan. Just-in-time length specialization of dynamic vector code. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 20:20–20:25, New York, NY, USA, 2014. ACM.

[77] L. Tierney. Compiling R: A preliminary report. In *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, DSC2001, March 2001.

[78] L. Tierney. A byte code compiler for r, 2013. http://homepage.stat.uiowa.edu/ luke/R/compiler/compiler.pdf.

[79] L. Tierney, A. J. Rossini, and N. Li. Snow: A parallel computing framework for the r system. *Int. J. Parallel Program.*, 37(1):78–90, Feb. 2009.

[80] S. Urbanek. Multicore: Parallel processing of r code on machines with multiple cores or cpus, 2009. http://www.numpy.org/.

[81] A. Vance. Data analysts captivated by R's power. *New York Times*, January 2009.

[82] S. Venkataraman. Large scale data analysis made easier with sparkr, 2014. https://amplab.cs.berkeley.edu/large-scale-data-analysis-made-easier-with-sparkr/.

[83] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 197–210, New York, NY, USA, 2013. ACM.

[84] 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.

[85] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO'10, pages 278–287, New York, NY, USA, 2010. ACM.

[86] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM.

[87] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The HipHop compiler for PHP. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'12, pages 575–586, 2012.