Abstract

R is a popular dynamic language designed for statistical computing. Despite R’s huge user base, the inefficiency in R’s language implementation becomes a major pain-point in everyday use as well as an obstacle to apply R to solve large scale analytics problems. The two most common approaches to improve the performance of dynamic languages are: implementing more efficient interpretation strategies and extending the interpreter with Just-In-Time (JIT) compiler. However, both approaches require significant changes to the interpreter, and complicate the adoption by development teams as a result.

This paper presents a new approach to improve execution efficiency of R programs by vectorizing the widely used `Apply` class of operations. `Apply` accepts two parameters: a function and a collection of input data elements. The standard implementation of `Apply` iteratively invokes the input function with each element in the data collection. Our approach combines data transformation and function vectorization to convert the looping-over-data execution of the standard `Apply` into a single invocation of a vectorized function that contains a sequence of vector operations over the input data. This conversion can significantly speed-up the execution of `Apply` operations in R by reducing the number of interpretation steps.

We implemented the vectorization transformation as an R package. To enable the optimization, all that is needed is to invoke the package, and the user can use a normal R interpreter without any changes. The evaluation shows that the proposed method delivers significant performance improvements for a collection of data analysis algorithm benchmarks. This is achieved without any native code generation and using only a single-thread of execution.

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

General Terms Language, Transformation, Performance

Keywords Dynamic Language, R, Vectorization

1. Introduction

In the age of big data, the R programing language is a rising star. It is considered today’s lingua franca for data analysis [26, 40]. Its growing popularity is the result of the productivity benefits that R brings to data analysis. This productivity ensues from R’s dynamic language features, its development ecosystem, and its interactive environment.

However, like programs in many other dynamic languages, R programs tend to be slow, which makes R difficult to be used for processing the truly BIG data. Earlier research revealed that R can be hundreds of times slower than C [28]. R programs can contain [42]: looping-over-data computation (Type I), vector code (Type II), and native library invocations where R is used as glue code (Type III). The major performance problem only appears in R code of Type I. In fact, glue code (Type III) that is dominated by the execution of native code library routines typically performs as well as equivalent C code. Also, if the execution of an R code segment or program is dominated by vector operations (Type II), the performance gap between R and C is much smaller than the gap in the Type I case. The reason is that vector operations are mostly implemented using native C code, which avoids much of the overhead of interpretation. Figure 1 shows the performance of two shootout [1] benchmarks written in the Type II vector programming style. As we can see, thanks to the vectorization, these two R programs are much faster than their Python counterparts, and are less than 10 times slower than the equivalent C implementation. This is a good result considering that these two R programs are interpreted while the C programs are compiled. In this case, vectorization makes all the difference since these two Type II R implementations are respectively 167x and 15x times faster than the corresponding scalar, Type I, R versions.

It would be ideal to reduce the interpretation overhead through automatic vectorization. However, automatic vectorization of arbitrary codes is only partially effective today.
We chose instead to apply vectorization to codes written in terms of the Apply class of operations, which is a well known, easy to use, and very popular class of operations in R. Apply operations accept a function and a collection of data items, and iteratively invoke the function with each data item. The popularity of Apply will likely increase due to the growing popularity of the Map-Reduce programming model. Indeed numerous data analytics problems have been implemented or re-implemented in this programming model [13]. Furthermore, Map-Reduce is the foundation of many R based big data analytics frameworks, such as Rabid [23] and SparkR [41].

Although the Apply class of operations is simple, the execution of these operations in R is typically slow due to the overhead resulting from the repetitive invocation of the input function by the interpreter. For example, R code lapply(L, f) maps the input data L to the output data with the 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 Type I codes and subject to significant interpretation overhead.

This paper presents an approach that reduces the interpretation overhead through the vectorization of the Apply class of operations. Our approach combines two transformations:

- **Function Vectorization** which transforms the function passed to Apply into a function that takes vectors as input. The original function operates on a single element at a time, while the transformed version operates on a vector containing all elements of the original input data collection and generates vector results. In the original Apply operation, the function is invoked multiple times, once for each element of the input data collection, while in the transformed Apply, the vector function is only invoked once. The body of the function is transformed by converting all operations that depend on the input parameters into vector operations or higher dimensional array operations if each original input data element was an array to begin with.

- **Data Transformation** which converts the input data (which could be a list of arbitrary complex data structures) into a vector form, so that the vectorized function can directly access the input data items in contiguous memory locations.

After these transformations, the original looping-over-data execution of an Apply operation is replaced by a single vector function invocation. The vector operations in the vectorized function can take advantage of the built-in vector functions of R, and execute with much less interpretation overhead.

We would like to point out that the function vectorization described in this paper differs significantly from the conventional automatic vectorization used to exploit the SIMD parallelism. First, our transformation does not need to determine whether the operation can be parallelized because parallelism is implied in the semantics of the Apply operations. Second, the speedups resulting from the transformations presented in this paper do not come from exploiting parallel hardware resources (such as multi-processors, SIMD devices, or GPUs), but from reducing the interpretation overhead. Therefore, speedups can be achieved even if there is a single-thread of execution. Third, a new challenge appears that needs to vectorize a language that contains vector extensions and does not have static typing. It is not existed in traditional vectorization domains. Fourth, the approach described here complements the use of Apply for parallelism in all levels (vector units, accelerators, multiprocessing, distributed computing). The parallel version of Apply operation can take advantage of multi-thread coarse-grain parallelism, the proposed method can reduce the interpretation overhead in each thread, and the vector operations obtained inside the vectorized function can make use of SIMD devices. We could therefore combine multi-thread parallelization of the Apply loop, reduction of interpretation overhead, and vectorization for SIMD devices together since they are orthogonal approaches.

The rest of this paper is organized as follows. The Apply class of operations and its interpretation overhead are discussed in Section 2. Each of the following sections describes one of the main contributions of this paper:

- **Section 3**, we introduce the transformation framework, which includes the data and the function transformations. Our techniques are applicable not only to R but also to any dynamic language with vector operations.

- **In Section 4**, we describe our implementation of these transformations. All the techniques are implemented in R. As such, our vectorization transformation does not require any modifications to the R interpreter. We also briefly discuss the application of our approach to other dynamic languages.

- **In Section 5**, we evaluate our implementation using a suite of machine learning kernels originally written in a Map-Reduce style. Our strategy can achieve 15x speedup for the iterative method benchmarks and 5x for the direct method benchmarks. Our evaluation does not use hard-
ware parallelism, that is all evaluations are done using a single thread of execution. Also, all improvements are based on high level language transformations. Our strategy does not use any native code generation, either. We then give a detail analysis of the factors that impact the effectiveness of this approach.

Section 6 covers the related work and Section 7 presents the conclusions.

2. Motivation

2.1 The 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 frequently used lapply function accepts a list, \( L = \{e_1, e_2, ..., e_n\} \), and a function \( f \). The value of \( \text{lapply}(L, f) \) is the list \( \{ f(a_1), f(a_2), ..., f(a_n) \} \). Other functions in the Apply family include apply which accepts a matrix or multidimensional array and returns a dense vector instead of a list, eapply which operates on an environment, and by, mapply, rapply, sapply, and tapply whose semantics can be found in any R manual.

Many computations can be naturally written using the Apply class of operations including, for example, all the machine learning kernels in [13]. The use of \text{lapply} is illustrated in Listing 1, which shows an R version of the Gradient Descent Linear Regression.

```
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 1: Linear Regression with lapply

Applications implemented in terms of Apply can take advantage of parallelism using R packages, such as SNOW [38], SNOWFall [22], and Foreach [43], 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 [23] and SparkR [41], both of which provide distributed memory parallel implementations of Apply.

2.2 Performance Issue of Apply Class of Operations

An outline of the implementation of \text{lapply} by GNU R interpreter is shown in Listing 2. Other functions in the Apply class have the similar implementations. This looping-over-data form incurs in a huge interpretation overhead, and as a result is very slow. Because Apply operations are widely used, GNU R implements them as C functions. However, each invocation of \( f \) (Listing 2, Line 4) is interpreted separately for each element of the input list. 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 2: Pseduo code of lapply

2.3 Vector Programming and Apply Operation

The Apply class of operations can be conceived as generalizations of array operations. Assume, a vector \( a \) with one million elements. To add one to each of the elements, we can use either \( \text{lapply} \{ a, \text{function}(x)\{x+1\} \} \), 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 as many times as the length of \( a \). For the second form, the interpreter will only invoke one \text{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 this example.

In the above example, rewriting the Apply operation as an array operation is simple, but rewriting arbitrary operations in terms of Apply can be very complex. For example, rewriting the vector code for \text{grad.func} in Listing 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 if the single input element is a complex data structure, such as a vector, matrix or a structure composed by vector, matrix and lists. The vectorization method we present next makes it possible to take advantage of the performance benefit of vector programming from a scalar programming paradigm.

3. Algorithm

This section describe the transformation framework in a general dynamic vector language context. It can be applied to R as well as other dynamic languages that have vector support.

3.1 Vectorization Transformation

Vectorization of the Apply operation can be represented as

\[
Lout \leftarrow \text{Apply}(L, f) \Rightarrow Lout \leftarrow \bar{f}(L)
\]

\(^1\)The result data representations are different in R
where \( L, f \) and \( L_{out} \) have the same meaning as in Section 2. The function \( f \) accepts a single object. We call it Single Object Function. \( f \) is a Vector Function, that can process a vector containing all elements in \( L \), say \( \{a_1, a_2, \ldots, a_n\} \), and return \( \{f(a_1), f(a_2), \ldots, f(a_n)\} \). In this way, the original loop shown in Listing 2 is transformed into a single function invocation.

\[ \vec{f} \]

The nature of the function, operations in the body of the function, and the type of the input data must all be resolved dynamically. Therefore, vectorization here must dynamically generate correct code that contains a dynamic selection of the vectorized execution and the single object execution. The latter is needed for those cases where vectorization is not possible.

We will first describe a basic algorithm that translates the original Apply operation into a vector function invocation under some simplifications, followed by the full algorithm.

### 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 1.

In order to simplify the discussion, we assume in this section that (a) The input data is an array, so that there is no need to convert the input data; (b) The control flow inside \( f \) does not depend on the formal parameters; (c) \( f \) is normalized so that all assignments are of the form \( v_3 \leftarrow op(v_1, v_2) \). These assumptions will be removed in the next section where the full algorithm is described.

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. We call them EXPANDED variables. Because \( R \) is a vector language, the single object variable 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 variable becomes a vector, a matrix, or a three dimensional array respectively. We call variables still representing a single object UNEXPANDED variables. Algorithm 1 illustrates the process of vectorizing the simple class of functions just described.

If we apply the basic vectorization algorithm to the grad.func function in Listing 1, \( yx \) will be marked as EXPANDED at the beginning, then \( y, x, \text{error}, \) and \( \text{delta} \) will be marked as EXPANDED because they are in the use-def chain of \( yx \). There are some subtleties to take into account. Thus, if the input data \( yx \) is \( \{y_1, x_1, \ldots, y_n, x_n\} \), \( yx[1] \) in the original function would return \( y_1 \). But when expanded to create the vectorized function, it should return a vector containing the first item of each elements in \( yx \), which is \( \{y_1, y_2, \ldots, y_n\} \). The reference \( yx[2] \) should be replaced in a 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 \( +, -, \text{sum} \) should be converted to handle the new aggregates correctly as discussed in the next section. The function can return the result \( \text{delta} \) directly, since it is already an EXPANDED variable.

According to the basic algorithm, the only changes are those shift of the computation to simple operations and accesses, following the structure defined in Figure 2.

If we flatten all operations in the original function \( f \), the vectorization transformation of Apply operations can be seen as loop distribution. Because we have expanded all the variables that depend directly or indirectly on the function parameters, the loop distribution is a legal transformation. If one statement in \( f \) is another loop, a loop interchange is also required to move the Apply level loop to the inner po-
Data: The single-object function \( f \) in \( \text{apply}(L, f) \) Mark formal parameters of \( f \) as EXPANDED;
while There is a change do
  foreach \( v_3 \leftarrow \text{op}(v_1, v_2) \) where either \( v_1 \) or \( v_2 \) is EXPANDED do
    Rewrite \( \text{op} \) to a new \( \tilde{\text{op}} \) that satisfy;
    if \( v_1 \) and \( v_2 \) are both EXPANDED then
      \( \tilde{\text{op}} \) 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
      \( \tilde{\text{op}} \) 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
  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
end

Algorithm 1: Basic function vectorization algorithm

Based on this algorithm, we have two data transformations, \( \text{PERM\_DOWN} \) and \( \text{PERM\_UP} \). \( \text{PERM\_DOWN} \) moves the first dimension of the data to the innermost position, and if this first dimension is stored in list format \( (L) \), it will be converted into dense form \( (D) \) once it is moved to the innermost location. 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 \( \text{PERM\_DOWN} \), the vectorized function \( \tilde{f} \) can access the data without the need for a gather operation. Figure 4 shows three examples, all of which can still use simple forms to access the input data as a vector.

The \( \text{PERM\_UP} \) does the reverse transformation. It moves the innermost dimension to the outermost position. This data transformation is typically used to convert the vector function’s return value back into the original shape.

Table 1: Data Representation of Different Types

<table>
<thead>
<tr>
<th>Data Dimension</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>([___])</td>
</tr>
<tr>
<td>1-D</td>
<td>Vector ([D_1]; \text{List} [L_1])</td>
</tr>
<tr>
<td>2-D</td>
<td>Matrix ([D_1 \times D_2]; \text{List of Vector} [L_1 \times D_2]); List of \text{List} [[L_1 \times L_2]]</td>
</tr>
<tr>
<td>Higher Dimensions</td>
<td>Array ([D_1 \times D_2 \times \ldots \times D_n]; \text{List} [L_1 \times L_2 \times \ldots \times L_n])</td>
</tr>
</tbody>
</table>

Figure 4: Data Access after \( \text{PERM\_DOWN} \) Transformation

Figure 3: Three Tasks in the Full Vectorization Algorithm
With these two transformations, each data object has two representations, the DOWN representation and the Up representation. The original “single object context” always uses the Up representation, and the “vector context” obtained after function vectorization always uses the DOWN representation. These two transformations will be applied repetitively when vectorizing nested Apply. For each vectorization, the strategy is to move the data’s outermost dimension to the innermost place by PERM_DOWN to match the movement of the computation loop from the outermost to the innermost.

We also define a REP_EXPAND operation, which 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, \([ \mathbf{1} ] \Rightarrow [ D ] \). 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 structures behaves similarly. REP_EXPAND is 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. we now have the vector) \(\vec{a}\). If the vector operation \(\oplus\) requires both operands to be EXPANDED, we can use REP_EXPAND to fix this limitation by generating the vector expression \(\vec{a} \oplus \text{REP_EXPAND}(b)\).

In the above notation, the change in signature of the function due to vectorization can be represented as \(f(v : [Q_1 \times \ldots \times Q_n]) \Rightarrow \vec{f}(\vec{v} : [Q_1 \times \ldots \times Q_n \times D_{n+1}])\).

### 3.3.3 Function Vectorization

The goal of function transformation is to convert the single object function \(f\) into an equivalent vector version

\[
f(v : [\ldots]) \text{body} \Rightarrow \vec{f}(\vec{v} : [\ldots] \times D) \text{body} \quad (2)
\]

We define a new operation VEC_FUN to do the function vectorization, where \(\vec{f} \leftarrow \text{VEC_FUN}(f)\). There are three cases to consider in the design of VEC_FUN.

- **Direct replacement** If the function \(f\) is an elemental function, as is the case for most of the math functions in R, the transformation just returns \(f\). Elemental functions can operate on a single object or on a vector of objects. If the vector version function has a different name, as is the case for \(c()\), whose vectorized version is called \(\text{cbind()}\), the function name in Apply must be changed.

- **Generic replacement** If \(f\) is a built-in function that cannot be vectorized, or \(f\) is too complex to be analyzed, we can create a generic vector version of \(f\), which is defined as follows: \(\vec{f}(\vec{v} : [\ldots] \times D) \{\text{PERM_DOWN}(\text{Apply}(f, \text{PERM_UP}(\vec{v})))\}\). 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_UPed so that the Apply operation can iterate over the right data. And the result must be PERM_DOWNed again to comply with the vector function’s output data shape. This fall-back mechanism guarantees that the vectorization algorithm can support arbitrary code in a dynamic language.

- **Body vectorization** If we can analyze the body of \(f\), we apply the algorithm described in the next section to transform \(f\) into \(\vec{f}\).

### 3.3.3 Vectorization of the body of a function

The vectorization of \(f\) requires (1) changing the original parameter’s type into the EXPANDED type by appending one additional dimension, and (2) transforming the function body so that it can 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 of this pass is to determine the shape of each variable, EXPANDED or UNEXPANDED. Assuming a gated SSA representation [30], if a variable is on the left hand side of any assignment (including assignments from gated PHI functions) in the use-def chain that starts at the function’s formal argument input list, it is marked as EXPANDED. Otherwise, the variable is marked as UNEXPANDED, which means the variable will be the same as the variable in the single object function.

#### Loop Transformation

This pass processes the loop statements in the function body that might have control divergences. We can identify these loops by examining the expression that decides the loop termination. If the expression contains a variable marked as EXPANDED, the loop must be processed. One possible approach to transform a loop is rewriting the loop body so that each statement in the body is controlled by a mask expression as discussed in [21]. But this approach would require modifications to the interpreter to support the predicated execution and will introduce wasted computation due to the masked operation. In our implementation, we use the same fall-back mechanism used for the generic replacement of functions. That is, we simply add an additional loop on the outermost position to process each element of the vector.

#### Branch Transformation

This pass transforms control divergent branch statement. Specifically, it transforms if statements with conditional expressions containing EXPANDED variables into a sequence of statements individually “guarded”. Our algorithm follows the approach of [4]. Given an if statement with EXPANDED condition \(c\), we first identify all the EXPANDED variables in the true and false blocks that are written in the blocks but are still live after the if statement. These variables are the operands of the \(\phi\) nodes in the basic block that post dominates the if statement. Then, we generate a flattened basic block, containing (in the order they are listed): the pre-dom block, true block, false block, and post-dom block. Finally, we replace the post-dom’s \(\phi\) nodes with a Select statement (ifelse in R) which se-
lects the first variable if the condition is true and the second variable otherwise.

**Code Rewriting** The final pass replaces all objects that are operated with EXPANDED variables with the appropriate vector form. This pass checks each statement $r \leftarrow op(v_1, \ldots, v_n)$. If any operand on 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 $\phi$ node, which means a $\phi$ node’s operand could be REP\_EXPAND($v_i$). The last step is to remove $\phi$ nodes. If one operand of the $\phi$ node is REP\_EXPAND($v_i$), where $v_i$ is from the basic block $B$, we insert the statement $v_i \leftarrow REP\_EXPAND(v_i)$ at the end of $B$.

### 3.3.4 Caller Site Rewriting

Given the single object function $f(v : [< \ldots >])$ and the corresponding vectorized function $\tilde{f}(\tilde{v} : [< \ldots > \times D])$, the Apply function invocation Apply($f$, data : $[L < \ldots >]$) is equivalent to a direct vector function invocation $\text{PERM}_\text{UP}(f(\text{PERM}_\text{DOWN}(\text{data} : [L < \ldots >])))$. For a dynamic language like R, if we do not know the body of $f$ at compile time, we replace $f$ with $\text{VEC\_FUN}(f)$, which will vectorize function $f$ at run time.

After code rewriting, the vector function invocation typically will have much less interpretation overhead than the original Apply function call. However, the new code also introduces the overhead of $\text{PERM}_\text{DOWN}$ and $\text{PERM}_\text{UP}$. One important task of the caller site rewriting is to do code optimization to reduce the data transformation overhead. There are two cases to consider. First, if the caller site contains $\text{PERM}_\text{UP}(\text{PERM}_\text{DOWN}(v))$ or $\text{PERM}_\text{DOWN}(\text{PERM}_\text{UP}(v))$, the restructurer simplify removes both transformations. Second, as discussed in Section 3.3.1, each data object has two representations, the UP and the DOWN. We can build an internal link between the two representations. The first time either $\text{PERM}_\text{DOWN}$ or $\text{PERM}_\text{UP}$ is invoked, the run-time creates the link. If either UP or the DOWN data object is modified, the link will be broken. When $\text{PERM}_\text{DOWN}$ or $\text{PERM}_\text{UP}$ is called again on the same object, 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 having to do any data permutations. This optimization is very effective to iterative data analysis algorithms.

### 3.4 Limitations of the Transformation

The strategy described above can successfully vectorize most applications written in terms of Apply, and reduce the interpretation overhead. However, it is not universally applicable since it has three main limitations:

**Cannot correctly vectorize Apply operations using functions with side effects** Although no side effects is expected of the functions used in Apply operations, side effects are possible in real programs. Unfortunately, it is hard to identify these side effects in a dynamic language context, and our implementation does not analyze side effects. Therefore, the user must be aware that vectorization as described here would change the outcome of the program in the presence of side effects.

The input collection must be a homogeneous data structure. That is, each element must have the same structure type. The $\text{PERM}_\text{DOWN}$ requires each element of the input list of Apply to be a homogeneous structure, so that the output DOWN shape object can be constructed. In some rare cases, this condition is not satisfied, then the $\text{PERM}_\text{DOWN}$ cannot be performed correctly. In R, this limitation can be solved by adding NAs to make the data homogeneous. But this filling will introduce additional overhead.

The strategy could increase memory usage and footprint. The vectorization strategy described above, could produce codes with less locality than the original. The reason is that in the original looping-over-data execution, each iteration only process one element of the data, and only uses little memory for intermediate results. After one loop, the intermediate result will be cleaned by Garbage Collection. However, the vectorized code consumes several elements at a time, and requires vectorized (and hence enlarged) all intermediate results, which requires more memory. This increased memory requirement affects locality and could have an adverse effect on performance.

**Possible slower execution** If vector execution cannot cover all cases, generic replacement is used as a fall-back mechanism for some function invocations and loop executions. Each generic replacement introduces transition overhead, and therefore the performance gained from vector execution may not be sufficient to obtain speedups.

### 4. Implementation

We have implemented the vectorization algorithm as an R package, named VALOR (Vectorization of AppLy for interpretation Overhead Reduction). Since it is written in R, the package is interpreted and therefore its installation does not require compilation. Section 4.1 to Section 4.4 describe the implementation details. Although this section only explains the realization in R, the technique can be easily extended to vectorize the Map style operations in other interpreted dynamic languages, which is briefly discussed in Section 4.5.

#### 4.1 Runtime Functions

##### 4.1.1 Data Transformation Functions

The three data transformation functions $\text{PERM}_\text{DOWN}$, $\text{PERM}_\text{UP}$, and $\text{REP}_\text{EXPAND}$ are all implemented in a recursive style so that they can support arbitrarily nested data structures. $\text{PERM}_\text{DOWN}$ and $\text{PERM}_\text{UP}$ only take one input argument, the data, and return the DOWN or UP representations respectively. $\text{REP}_\text{EXPAND}$ must know how many times to expand the input data. Therefore $\text{REP}_\text{EXPAND}$
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.

4.1.2 Vectorization Functions

We implemented VEC_FUN as described in Section 3.3.2. When it is invoked at the 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 by PERM_DOWN operation, most data accesses inside the function maintain their original form or are only slightly changed after vectorization. To describe the transformation of memory accesses, consider a variable \( x \) in the original function \( f \), that is marked as EXPANDED in \( \tilde{f} \), there are three basic classes of accesses involving \( x \):

- \( x \): The reference in \( \tilde{f} \) will remain \( x \). In the original function, \( x \) returns one element while in the vectorized form \( x \) returns the whole EXPANDED variable.
- \( x[a] \): In this case \( x \) is a vector in the original function. In the vectorized function, \( x \) will be a matrix, and \( x[a] \) is changed to \( x[a] \) to access the \( a \)th column. \( x[a] \) is chosen because R uses column-major storage.
- \( x[\text{label}] \): In this case, \( x \) is a structure (list) in the original function. In the vectorized function, \( x \) is a structure of a vector, and \( x[\text{label}] \) 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 Many R operations have higher dimensional counterparts that our 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 2.

<table>
<thead>
<tr>
<th>Low Dim</th>
<th>High Dim</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, *, /, ^</td>
<td>+, *, /, ^</td>
<td>They support vector by default</td>
</tr>
<tr>
<td>mean</td>
<td>meanSums</td>
<td>Sum along rows</td>
</tr>
<tr>
<td>length</td>
<td>nrow</td>
<td>Length along rows</td>
</tr>
<tr>
<td>c</td>
<td>cbind</td>
<td>Column binds</td>
</tr>
<tr>
<td>unlist</td>
<td>simplify2array</td>
<td>Transform list of vectors into a matrix</td>
</tr>
<tr>
<td>which.max</td>
<td>col.max</td>
<td>Find the max value’s pos in each row</td>
</tr>
</tbody>
</table>

Generic Replacement If the VEC_FUN routine cannot analyze or transform an operation (a native function that is not in Table 2, or a very complex R function), it generates a generic expressions using Apply to simulate the vector execution as described in Section 3.3.2.

Recursive Transformation In many cases, the operation in a function body will be transformed by applying VEC_FUN recursively. For example, \( op \) will be changed into VEC_FUN(\( op \)), so that the vectorization routine will be invoked at run time to get the vectorized version of \( op \).

4.2 Caller Site Interface

Our package provides two translation APIs, va_compile and va_vecfun. The former accepts R expression, and the later accepts a function. Both of them will go through the input object’s AST, and do caller side code rewriting, which translates Apply calls into vector function invocations. Listing 3 shows the code of Listing 1 after translated by va_compile. Listing 4 is the vectorized \( \text{grad.func} \) function generated at run time from the expression VEC_FUN(\( \text{grad.func} \)).

4.3 Optimizations

The code in List 3 and List 4 has not been optimized. There are many redundant computations. VALOR applies several optimizations to remove these redundancies.

4.3.1 Removing redundant data transformations

This kind of redundancy is mainly from the iterative part in a program. For example, \( \text{PERM_DOWN}(yx) \) in Line 4 of Listing 3 will be invoked by every loop iteration. But \( yx \) is loop invariant. We could have used LICM (Loop Invariant Code Motion) to remove the redundancy, but instead VALOR uses the runtime memorization technique described in Section 3.3.4, which has a simpler implementation and
produces similar results. List 5 is the optimized code. Because R has neither a map data structure nor references, we have to use hidden variables to do the memorization. The variable .va.yx is the DOWN shape representation of yx, and .vasrc.yx records the original variable that was transformed into the DOWN shape. If the function requires a DOWN variable, the code first checks whether a DOWN variable exists (Listing 5, Line 4), then checks whether the source variable has not been changed (Listing 5, Line 5) by the identical function. If both conditions are satisfied, the DOWN variable is returned, otherwise PERM_DOWN is applied, and set the link.

R’s copy-on-write mechanism makes sure that any modifications to the source object breaks the runtime link. In the previous example, both yx and .vasrc.yx point to the same memory object after the runtime link 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 when comparing .vasrc.yx with yx.

The runtime approach is also used in the evaluation of the function vectorization. For example VEC_FUN(grad.func) of Line 4 in Listing 3 is also loop invariant.

Another optimization to remove redundant data transformations is in Listing 5, Line 11 with the use of R’s delayedAssign function. The vector function invocation returns a DOWN shape variable .va.delta. Line 11 creates a promise of delta. If delta is never used, the promise will never be forced, and the PERM_UP function call will be avoided.

4.3.2 Removing redundant data replication

The REP_EXPAND in x * REP_EXPAND(theta, x) from Line 4, Listing 4 could be redundant because “*” supports implicit value replication. We use static and runtime checks to remove the unnecessary data replication. If the operation supports implicit data replication (all R functions implicitly support replication of scalars), 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. We also insert dynamic type checking inside REP_EXPAND. If REP_EXPAND finds out the input UNEXPANDED variable is a scalar, the function will return that variable.

4.3.3 Optimizing Reduce Function Calls

The interpretation of Reduce is similar to the interpretation of Apply in that it involves a looping-over-data strategy with large interpretation overhead. If the reduction’s operator is “+”, we use sum or colSums to replace Reduce. These two functions are implemented in a native library, and have much less interpretation overhead than that of the Reduce function. One limitation is that sum and colSums only operates on dense vector objects. However, in the MapReduce framework, the input data of Reduce often comes from an Apply operation. 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 5 is the optimized code. va_reduceSum checks the type of the input data and calls sum, colSums, or the original Reduce.

4.4 Optimization Opportunities inside GNU R

The strategy described in Sections 4.1 to 4.3 as implemented in VALOR requires no modification to GNU. However, if we were willing to slightly modify GNU R runtime as discussed next, we could get an additional performance boost.

Adding more built-in vector functions. The performance of the result code from vectorization is highly dependent on the coverage of the built-in vector functions. If all the operations in the vectorized function had a vector counterpart like that of the operation in Table 2, a low interpretation overhead could be achieved. However, there are many operations from real applications are not covered by the built-in vector operations of R. One simple approach is to add the vector version of the most frequently used functions to the R runtime to improve the coverage. For example, implementing rowSort() as the vector version of sort() to sort each row of a matrix. The approach is relatively simple, but requires additional engineering effort.

Using SIMD unit to accelerate built-in vector functions. The current built-in vector operations of GNU R, such as those in Table 2, do not use processor SIMD instructions. Performance would likely improve if all vector operations use these instructions. Like the coarse grain parallelism of the multi-thread version of Apply, SIMD is orthogonal to the vectorization transformation proposed in this paper, and we could get another level of performance improvement with it. However, it requires rewriting the R runtime code.

4.5 Extending to Other Dynamic Languages

The strategy discussed in this paper including the algorithm described in Section 3 and the optimization approaches de-

---

Listing 5: Optimized Code from va_compile

```r
yx <- ...
for(i in 1:50) {
  .va.delta <- VEC_FUN(grad.func)
  if(exists(".va.yx", inherits = FALSE) || !identical(.vasrc.yx, yx)) {
    .va.yx <- PERM_DOWN(yx)
    .vasrc.yx <- yx
  }
  .va.yx
}
delayedAssign("delta", PERM_UP(.va.delta))
theta <- theta - alpha * va_reduceSum(.va.delta)
  / length(yx)
```
scribed in this section can be generalized to other interpreted dynamic languages. Map operations are pervasive in dynamic languages. 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. Matlab has a number of functions (arrayfun, bsxfun, cellfun, et al.) that can be used with anonymous functions to perform map operations.

The proposed approach can be applied to other dynamic 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, as was done in NumPy \([2]\) for Python, this approach can be applied with little change.
- **Automatically Vectorizing Loops** Our current vectorization strategy assumes that the computation is written in terms of Apply operations. If the program is not written in this way, it is possible to automatically vectorize loops, transform the body of the loop into a function, and introduce an Apply operation that calls this function. Loop vectorization can be done either by using dependence analysis or by extending the dynamic language with annotations similar to the SPMD of OpenMP.

5. Evaluation

We evaluated the performance of our vectorization package by comparing the running time of the vectorized code with the time of the original code written in terms of Apply class of functions. Our vectorization package\(^4\) and all benchmarks\(^5\) are open-sourced on Github.

5.1 Benchmarks

The benchmarks in the evaluation are kernels of data analytics and machine learning algorithms collected from \([23]\), \([41]\) and \([13]\). Table 3 lists the kernels and the configurations we used. The first seven kernels use iterative algorithms and the last six use direct algorithms. Iterative and direct methods are two approaches in computational mathematics. The former requires repeating the computation several times to generate a sequence of improving approximate solutions until it reaches the converge condition, while the later method attempts to solve the problem by a finite sequence of operations. In the benchmark implementation here, the iterative method will invoke Apply several times, and the direct method will only invoke Apply once.

We fixed the number of iterations for iterative algorithms so that the running time is not dependent on the input data. We included both the single-variable configuration and multi-variable configuration in the benchmarks for some algorithms, for example we included both LR and LR-n, to make a more extensive evaluation. In fact, if we had only included single-variable configurations, the performance number would be better, but the result would be biased. Regarding the implementation, the single-variable configuration and multi-variable configuration are different. We used some special routines to optimize the single variable case, and its base performance is slightly better than that of multi-variable implementations when restricted to a single variable.

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. We turned on R byte-code JIT compiling at its fullest optimization level for all the tests.

The running time for iterative benchmarks is determined by measuring the time of one iteration. It is obtained as the average time per iteration of iterations 6 to 15, which is when the application runs into a steady state\(^6\). 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 below overhead due to data permutation both the iterative benchmarks and direct benchmarks. Table 3 only lists the base input size. We also evaluated the input size 4 and 16 times larger than the base size. For example, 4M and 16 M samples as input for ICA.

5.3 Vectorization Speedup

Figure 5 and 6 show the speedup of the kernels with Apply operation vectorization relative to the unmodified 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 reducing the number of machine instructions executed. 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 29.7x speedup shown in Figure 5 for LR with the base input is the combination of these two effects. Other benchmarks have similar hardware performance counter metrics.

For the codes we used, the vectorization approach is not sensitive to input size. With the increase of the input size,\(^6\) the first few iterations behave different than later iterations because their (mainly memory) behavior is influenced by the system’s state before the first iteration.

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

\(^4\) https://github.com/wanghc78/valor

\(^5\) https://github.com/wanghc78/benchmarks/tree/vecapply

\(^6\) The first few iterations behave different than later iterations because their (mainly memory) behavior is influenced by the system’s state before the first iteration.
Table 3: Benchmarks and configurations

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

Table 4: Data transformation overhead (Iterative benchmarks)

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

Table 5: Data transformation overhead (Direct benchmarks, Base Input)

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

The speedup becomes as a result of lower CPI due to the increase in the number of consecutive operations, and the lack of significant change in cache miss rate.

The speedup number of different benchmarks varies a lot, depending on different factors such 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. The results are better if 1) most scalar operations have corresponding vector implementations in R; 2) the data elements to be processed by the Apply operation are relatively simple and homogeneous; and 3) the code must have relatively few control divergences. We will discuss these factors further with experiment data in the following sections.

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 3.3.1). However, this kind of data permutation is very expensive, especially when the size of the input is large. In general, the data transformation overhead is due to \texttt{PERM\_DOWN} and \texttt{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 \texttt{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 shows overhead of iterative benchmarks, which is very large in most cases. The 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 5 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 \texttt{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 have no corresponding vector version. The vector computation has several fall-backs to generic transformation cases, which lead heavy back-and-force data permutation overhead.

5.5 Vectorization of Nested Apply Functions

Some kernels in Table 3 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 goes over all the points, while the inner Apply loops over all the centers. The outer Apply of Near-
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 our experiments, vectorization happens when the compiler meets the first `Apply` function call. We evaluated the performance of this schema (outer-only) by comparing it with other two schemas (inner-only and both). Figure 7 shows the result for the base input case.

In k-Means, the inner `Apply` only loops over 10 cluster centers, and the inner only schema has limited performance improvement. Vectorization of both `Apply`s 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 we vectorize both levels, the data object will be expanded into a 10k by 10k sized object, which causes large memory consumption, and has less speedup. Based on these observations, the outer only schema seems to be good enough for our benchmarks. A better schema would take the size of the input data into consideration to decide when to vectorize.

### 5.6 Vector Programming in the Applications

Another factor that impacts the speedup of our vectorization is the usage of vector programming in the original single object function. For example, Figure 5 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, which 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 already has relatively low interpretation overhead. Our vectorization algorithm can handle the single object function with vector programming well. Because the base case has less interpretation overhead, our vectorization achieved relatively low additional speedup. Figure 8 shows the speedup by varying the n values of LR-n, LogitReg-n, k-Means-nD and LR-OLS-n. The figure shows that the smaller the n value, the base function is more likely a Type I R code, and has large interpretation overhead. Then, our approach can achieve better speedup. On the other hand, the larger the n value, the smaller speedup that our approach can achieve due to the relatively efficient of the base case.

### 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 a relatively long input data list 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. These sequences of long vector operations hinder data locality. And the long intermediate results also consume more memory.

Tiling can reduce the impact of this issue. Based on our evaluation of different tiled sizes, we can achieve up-to an additional 30% performance improvement with a good choice of tiled size. A better way is to do native level vector code fusion [36]. After our vectorization, the function operations on long vectors, which are very suitable to apply native level code fusion.

5.8 Built-in Vector Function’s Support

One of the most important factors affecting performance is the vector computing support from R’s built-in functions. If all the operations in the vectorized function are in Table 2, we could expect a higher speedup. This is why Linear Regression gets such a good speedup. However, many functions do not have a corresponding higher dimensional versions. For example, R’s order function is used in k-NN to locate the k closest training samples of one testing sample. It accepts a vector, and returns a permutation indexes vector. The vectorized function of order should accept a matrix, and return a matrix. There is no such a built-in function in R. Therefore, the vectorization compiler must use the generic replacement transformation in which Apply traverses each row of the input matrix to simulate the vector execution. Because order is a time consuming part in k-NN, the final speedup of the vectorized k-NN is not high. The crossproduct function has a similar limitation, and it hinders the speedup of LR-OST and PCA.

5.9 Comparing with Coarse Grain Parallelism

Coarse grain parallelism can also be used to accelerate Apply class of operations. The R package parallel provides mclapply, parLapply, and other similar interfaces to run Apply operations on different R processes in the same machine or across machines. We compared the performance of the proposed vectorization approach with the coarse grain parallelism in this section. We used mclapply to replace the original lapply in the benchmarks, and measured the performance at different parallel levels. Figure 9 and 10 show the speedup numbers of the mclapply based coarse grain parallelism relative to the default lapply which only uses a single process.

The speedup number of this coarse grain parallelism is not very high. The coarse grain parallelism suffers from two major sources of overhead: (1) fork process: each mclapply operation requires forking new R processes since they must have the latest data; (2) data transfer: the result from forked R processes requires IPC communication. If the data set is large while the computation time for each item is small, the overhead will overwhelm the gain from parallel computing. On the other hand, the proposed vectorization approach has no such overhead since it uses single thread execution in the original R process, and can achieve larger performance speedup.

Coarse grain parallelism from the parallel package is more suitable to run Apply tasks with small data input and complex single object function, like the NN and k-NN benchmarks. In this situation, we can still combine the coarse grain parallelism with our approach to get better performance since these two approaches are orthogonal.

6. Related Work

6.1 Function Vectorization

Our approach transforms a single object function used by the Apply class of operations into the corresponding vector version. It belongs to the function vectorization domain. In [21], the term Whole-function Vectorization is used to describe the transformation of a single object function to a vector function that accepts vector input and processes the data using vector operations. The paper proposed the algorithm to vectorize OpenCL kernels, and targeting CPU SIMD units or GPUs. Similar approaches were used in Intel ispc [32], River Trail [19], and [24].

The concept of function vectorization in the above work is similar to ours. But these projects all targeted conventional
static languages, where the data types and the operations in
the single object function are relatively simple. Our work
supports all kinds of complex data types in R, which requires
data transformations to ensure that the vectorized function
accesses the data in a vector form, and transforms arbitrary
operations with different schemes.

6.2 Vectorization in Scripting Language

Code vectorization for static languages have been studied
and successfully used in commercial compilers for decades.
In [27], automatic vectorization is used to translate Mat-
lab scalar loops into vector code. But a major challenge of
loop vectorization is data dependence analysis, which can
become quite complex (particularly in the case of symbolic
computations) even for a static language. The dynamic fea-
tures of scripting languages make the problem even harder.
As a result, vectorization of arbitrary code in the dynamic
language domain is not common. Our approach targets one
specific program paradigm to bypass the data dependence is-
sue, and successfully vectorizes the operations of many ap-
plications.

6.3 R Optimization

R optimization work can be classified into two categories,
 Improving GNU R and building new R virtual machines.

Improving GNU R [37] developed the byte-code compiler
 and interpreter for GNU R, which greatly improved R’s
speed compared with the default AST interpreter in scalar
dominated code (Type I). pqR [29] added many small im-
provements to GNU R, such as data sequence data type,
more precise reference counting, parallel execution of vec-
tor computation. Enterprise R [6] replaced GNU R’s built-in
routines with high performance libraries, such as Intel MKL.

Our previous work ORBIT [42] used profiling driven ap-
proach 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.

Building new R virtual machines Renjin [8] implemented
an R AST interpreter on top of JVM and applied code fusion
and lazy evaluation optimizations. FastR [20] used the self
optimizing AST interpreter approach from Truffle [45], and
built a new JVM based R interpreter. It also applied data
and code specialization optimization. Riposte [35] targets
long vector R code (Type II). It achieved high speedup for
long vector code through optimizations including trace JIT,
vector code fusion, and SIMD and parallel execution. The
latest version [36] implemented vector length specialization
for short vector code. Rapydo [18] is another new R VM
based on the PyPy framework.

The approach introduced in this paper belongs to the
first category. It translates Type I R code into Type II R
code to reduce interpretation overhead. There is no similar
approach in related work. Furthermore, our work 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 virtual machines.

6.4 Parallel R Processing Systems

A single R instance has the problem of slow running speedup
and memory size limitation. Coarse grain parallel computing
is widely used with R to help it process large data set.
[15] lists R packages, applications and runtime systems that
leverage parallelism to improve R’s capability.

SNOW [38] 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 [39] of CRAN uses the
similar model in a shared memory machine, but provides
more comprehensive interfaces. They both use the Master-
Slave model. They have been merged together, and become
the official parallel package in R.

ScaleR [7] is a commercial product that also follows the
Master-Slave model. It uses Parallel External Memory Algo-
rithms (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
offloads them to multiple parallel processes.

Different to these coarse grain parallelism that use parallel
hardware resources to run the application, the method of
this paper improves the running speed of R by reducing the
interpretation overhead. It can obtain speedups even if R still
uses a single-thread of execution. This level of interpretation
overhead reduction is orthogonal to the above coarse grain
parallelism, and they can be combined together to get better
performance.

6.5 Optimization in Dynamic Languages

Specialization SELF [11, 12] is an early pioneer of using
specialization optimizations. Many types of specializations
are discussed in the literature, including Type Specialization
MaJIC [5] for Matlab, [17] for JavaScript; Function Argu-
ment Specialization [34] for JavaScript; Interpretation Spe-
cialization [44] for Lua.

JIT Native Code Generation Generating native code by
JIT is a common approach to improve the performance of
dynamic languages. This has been done for Python [10, 33],
JavaScript [16], LuaJIT [31], and PHP HHVM [3].

Ahead-of-time (AOT) Compilation Another approach to
improve the performance of dynamic languages is to do
Ahead-of-time (AOT) compilation of a dynamic language
into a low level static language, for example PHP phpc [9],
HipHop [46], and Matlab FALCON [14].

7. Conclusion and Future Work

We have described an approach to vectorize Apply class of
operations in R. The approach combines data transformation
and function vectorization, and transforms the looping-over-data execution of `apply` into a vector function invocation to reduce interpretation overhead.

There is much room to improve the current work. First, the overall speedup is highly dependent on the underlying built-in vector function support. We plan to implement the vector version of some frequently used R built-in functions. For example, the vector version of `order`. Second, as the code is transformed into long vector computations (Type II R code), it exposes more opportunities to explore the parallelism at different levels. We plan to combine the processor’s SIMD unit and thread level parallelism to get better performance. Lastly, we plan to integrate this work with R cloud computing frameworks, such as Rabid and SparkR. 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.

Acknowledgments

We thank Olivier Tardieu, Luke Tierney, Jan Vittek and Michael Lawrence for their suggestions and help. We also thank anonymous reviewers and attendees of the RIOT2015 workshop for their valuable comments. This material is based upon work supported by Huawei gift funding, IBM Fellowship, and the National Science Foundation under Award CNS 1111407.

References


