Optimizing R Language Execution via Aggressive Speculation

Lukas Stadler Adam Welc Christian Humer Mick Jordan
Oracle Labs, AT Oracle Labs, US Oracle Labs, CH Oracle Labs, US
{lukas.stadler,adam.welc,christian.humer,mick.jordan}@oracle.com

Abstract

The R language, from the point of view of language design and implementation, is a unique combination of various programming language concepts. It has functional characteristics like lazy evaluation of arguments, but also allows expressions to have arbitrary side effects. Many runtime data structures, for example variable scopes and functions, are accessible and can be modified while a program executes. Several different object models allow for structured programming, but the object models can interact in surprising ways with each other and with the base operations of R.

R works well in practice, but it is complex, and it is a challenge for language developers trying to improve on the current state-of-the-art, which is the reference implementation – GNU R. The goal of this work is to demonstrate that, given the right approach and the right set of tools, it is possible to create an implementation of the R language that provides significantly better performance while keeping compatibility with the original implementation.

In this paper we describe novel optimizations backed up by aggressive speculation techniques and implemented within FastR, an alternative R language implementation, utilizing Truffle – a JVM-based language development framework developed at Oracle Labs. We also provide experimental evidence demonstrating effectiveness of these optimizations in comparison with GNU R, as well as Renjin and TERR implementations of the R language.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization, Run-time environments

Keywords R language, Truffle, Graal, optimization

1. Introduction

The R language is an open-source descendant of the S language, and was created by Ross Ihaka and Robert Gentleman at the University of Auckland in the 1990s [9]. Its predecessor, S, was created by John Chambers et al. at Bell Labs in the 1970s, and began as a set of macros simplifying access to Fortran's statistical libraries [2]. Over the years, the language has become increasingly more popular, with its applications including data discovery and analysis as well as machine learning, over 11,000 packages in the CRAN [1] and Bioconductor [2] repositories, and its user base exceeding 2 million [18]. At the same time, the language has also grown quite complex, becoming a mix of many different programming language paradigms in sometimes unusual and surprising combinations. For example, R's syntax is superficially similar to languages like Java or JavaScript, but at the same time all arguments are lazily evaluated. It is both functional, in that functions in R are first-class data structures and it is, in practice, largely side effect free, but it is also object oriented and those side effects that do occur are largely uncontrolled (in particular, due to lack of support in the type system). R also supports deep introspection and allows for extensive modifications to runtime data structures, for example, iterating and modifying symbol lookup chains at runtime.

For over 20 years R had only one reference implementation, GNU R, and it is only recently that alternative implementations with varying goals and underlying assumptions started to emerge (see Section 7 section for details). GNU R is the most mature implementation and has worked well in practice, but has some limitations (e.g., slow execution of R code due to it being interpreted) that make it less suited to face some of the emerging challenges, particularly in the realm of so called *big data*.

In this paper we present FastR[3], an alternative implementation of R that efficiently implements all important and difficult to optimize features of the language. FastR is built on top of Truffle [27], a framework for developing programming languages utilizing a JVM (Java Virtual Machine).

Our main contribution is to demonstrate that, even when considering a complicated set of language features, aggres-

DLS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4445-6/16/11...$15.00
http://dx.doi.org/10.1145/2989225.2989236

[1] http://cran.r-project.org [2] http://www.bioconductor.org
[3] We share the name and part of the code base with our (mostly spiritual) predecessor, called Purdue-FastR [11, 17]. This earlier prototype is now largely abandoned and it did not implement many important language features, such as the S3 object model that is extensively used in R libraries.

sive speculation expressed in terms of only a small set of fundamental techniques, if properly supported by the underlying infrastructure, can be very effective in delivering a highly performant language runtime, especially when combined with a carefully selected and implemented set of language-level optimizations. In particular:

- We describe how different facets of speculation, that is caching, system-wide assumptions and specialization, help in developing novel optimizations of R's lazy evaluation in presence of largely unrestricted side effects.
- We show how the same techniques are effective in optimizing R's symbol lookups, function and method calls, and vector access operations.
- We demonstrate effectiveness of our lazy evaluation optimizations as well as present overall performance comparison between FastR and other implementations, based on experimental evaluation using a variety of benchmarks. In our set of benchmarks, FastR is on average ∼97x faster than GNU R (with ∼7x geomean speedup).

2. The Challenge

On the surface, R looks similar to other popular programming languages such as Java or JavaScript. For example, consider the following function:

```
function(x) {
  for (i in 1:10000) {
    x[i] <- i
  }
}
```

With `<-` and `[]` being R's assignment operator and indexing operator, respectively, and `1:10000` representing a sequence of numbers between 1 and 10000, execution of this statement will result in populating some indexable data structure `x` with consecutive numbers. Two main indexable data types in R are vectors and lists, the former containing elements of the same primitive type (e.g., a number), and the latter potentially containing elements of different types. Operations on vectors of numbers are very frequent in R and, as such, it is expected that they are performed efficiently. The example above, assuming `x` to represent a vector of numbers, would ideally have similar performance to an array assignment in Java.

Unfortunately, syntax is where the similarities between R and languages like Java or even more dynamic languages like JavaScript end. In particular, function arguments are lazily evaluated in R, so that each argument (e.g., argument `x` in the example above) is represented by a *promise* [4] – an internal data structure storing the expression and the evaluation environment used to compute concrete argument values when needed. This complicates the implementation, not only due to the overhead of creating promises throughout the call chains, but also due to function code being "polluted" by the code needed to obtain concrete argument values (see Section 5.2 for more details).

Another complication is that assignment operator, indexing operator, curly brace and all keywords are implemented as functions[5]. These functions can potentially have side-effects, and their definitions can change at any time during execution of the loop, for example, looking up the assignment function `<-` could lead to different results in different iterations of the loop. Furthermore, because variable scoping in R is dynamic and can be modified at the language level (and these modifications can be triggered as a side-effect of a function execution), it cannot be trivially guaranteed that `x` is going to point to the same data structure throughout the entire execution of the loop. Finally, the function implementing the subset operator is *generic*, so that a specific function to execute is chosen at runtime based on the existence and content of the vector's *attributes*. These attributes, which are meta-data that can be associated with any data structure, can change freely, and the chosen function can be arbitrarily complex (see Section 5.3.1 for additional details).

This paper will demonstrate that using aggressive speculation techniques, as well as the right tool chain (described in the following section), it is possible to remove many of the R language's runtime overheads and improve on the current state-of-the-art.

3. Truffle Overview

Truffle [27] is a framework for implementing language runtimes in Java utilizing JVM services (e.g., memory management). Its API provides many useful building blocks for creating a language runtime, for example:

- base node classes for building an Abstract Syntax Tree (AST) interpreter
- a domain specific language (Truffle DSL) that helps building specialized node classes and simplifies construction of general-purpose inline caches
- support for efficient management of variable scopes (called *frames*)
- source code management and interfaces for tools like debuggers

In a Truffle-based interpreter, the ASTs that represent a program will start out in an uninitialized state and specialize according to the actual behavior at runtime. This means that the trees only contain implementations of operations for data types that have been encountered at runtime. For example, if up to a certain point the "+" operation was only executed with integer arguments, and is then executed on floating-point values, the framework will rewrite the operation in the tree to a more generic implementation that can handle both cases. As long as language developers are careful to provide

[4] In other lazy evaluated languages, such as Haskell, also called *thunk*.

[5] The special syntax is syntactic sugar that is removed by the parser.

progressively more generic operation implementations, the tree will stabilize eventually.

Truffle also provides *assumptions*, which can be used to guard important system-wide conditions. They start out in the “valid” state and can be invalidated explicitly if a condition they are guarding is about to be broken. The main property of assumptions is that checking them is very cheap (potentially free under certain circumstances), while invalidating them is a very expensive operation. This can be used to speculate on conditions that are very likely to be true, but which cannot be statically proven so. As long as the number of failing assumptions is small, the tradeoff between cheap checking and expensive invalidation is beneficial.

Truffle can be enhanced by employing a just-in-time compiler called Graal [27]. Graal can take the ASTs produced by a Truffle language and compile them to efficient machine code. It does so via partial evaluation, also called the Second Futamura Projection [7]. The important property of this process is that it generates a compiler from the interpreter, so that language implementations on top of Truffle do not need a separate custom-built compiler.

Compilation to native code happens only after the AST stabilizes and so called “hot paths”, that is portions of the application code that are executed most frequently, have been identified in the course of interpreted execution. This compilation strategy makes language implementations using Truffle and Graal particularly well suited for executing long-running applications, which aligns well with FastR’s focus on peak performance (as opposed to startup performance).

Truffle provides an additional mechanism to language developers, so called *profiles*. They allow for passing additional information to the compiler to help it generate high-performance native code, at a small additional cost to the interpreted code. Numerous different types of profiles exist, with the most important ones being *condition profiles* and *type profiles*. Condition profiles are used to remember conditional branches that were never taken (to help the compiler generate more concise code). Type profiles remember dynamic type information and help in call devirtualization.

Another integral part of achieving high performance is Graal’s ability to deoptimize and recompile native code. Even if the execution stabilizes on a certain set of discovered types and valid assumptions, it cannot be guaranteed that in the future no new types are introduced and no assumptions are violated. If the framework detects such situation, native code can be invalidated with the execution falling back to the interpreter, and recompiled after execution stabilizes again.

4. FastR Overview

FastR is an open source[6] implementation of R in Java on top of the Truffle framework. As with all Truffle-based language implementations, it is at heart an AST interpreter that specializes the tree at runtime. It reuses code from GNU R where appropriate, either translated into Java or called as C or Fortran code through the R native interface.

The goal of FastR is to be a complete and fully GNU R compatible implementation of the R language, including development tools such as the standard debugger support, as well as support for third-party R packages available in public repositories. Since third-party packages are such a big part of the R ecosystem, it is crucial for FastR to be able to install and load these in exactly the same way as GNU R from the user’s perspective. Installing non-trivial packages exercises most of R’s sophisticated language features, so that FastR had to reach a significant level of compatibility with GNU R in order to be able to do so.

Performance-wise, the main focus of FastR is to accelerate execution of the R code. Currently, a typical work flow for complex long-running R applications is to prototype an algorithm in R and then identify and re-write performance-critical portions of R code in C/C++. This is time consuming, error-prone, and difficult to test for correctness (e.g., due to differences in how languages handle floating point computations), and will hopefully become unnecessary with projects like FastR providing more efficient execution of R code. Nevertheless, since many packages include code written in C/C++ or Fortran, FastR must implement the interface for calling to and receiving callbacks from such code. Unfortunately the native interface is very complex – it exposes the internals of the GNU R implementation and assumes that an implementation is in C/C++ and can freely share memory. This is problematic for a Java implementation as Java has very strict rules on the use of data outside the JVM as established by its Java Native Interface (JNI).

At the time of writing, FastR implements all the major features of the R language, including but not limited to supporting all user-facing R data structures, lazy evaluation, and both S3 and S4 object models. The main remaining missing features are full graphics support as well as selected builtin functions and parts of the native interface. Nevertheless, FastR is complete to the point that it can correctly install over 2000 of the CRAN packages as well as run (in parallel) unmodified selected production applications. The FastR code base also includes approximately 10,000 unit tests which compare individual operations against the GNU R reference implementation. Some of the tests were written by hand and some were generated automatically by the testR [23] project.

To summarize, since FastR implements the majority of important language constructs, its semantic compatibility with GNU R is high, but more work is required on completeness as, for example, many R applications rely on packages with native components that FastR does not yet support.

5. R Language Optimizations

Implementing the R programming language is a challenging task for VM and runtime engineers. It includes many programming language features and paradigms that were as-

[6] https://github.com/graalvm/fastr

sembled with little concern for efficient and compiled execution. In this section we show how, despite R's complexity, the majority of language performance problems can be mitigated via aggressive speculation, where the runtime system optimizes an executing program based on a certain set of predictions (e.g., that the number of different argument types for a give function call is small), but is also prepared to handle the general case, even if not as efficiently. In FastR we distinguish three major different speculation techniques:

Inline Caches In dynamic languages it is often hard or impossible to determine the target of a specific call at compile time, so that every call is in principle polymorphic. In general, the number of call targets is unbound, but we can speculate that the number of targets encountered at runtime is small, and that they can be cached to avoid expensive lookups and call preparation.

Inline caches, though typically associated with function or method call optimizations, do not have to be restricted to call sites – they are a generic concept that can be applied in all places where the number of options is unbound in theory, but in practice only a small set of concrete options will appear at any one place.

The so-called guards, used to determine whether a specific inline cache entry applies in the current situation, are not restricted to simple equality or type checks. However, all inline caches use guards whose runtime cost is negligible compared to the actual operation they are used for.

Assumptions Inexpensive *assumptions* (see also Section 3) can be used to speculate that certain system-wide conditions which need to be checked frequently will hold throughout a program's execution.

The underlying system tries to make checking assumptions as fast as possible, while invalidating an assumption may be an expensive operation.

The use of assumptions needs to be carefully designed so that the overall system stabilizes to a state where no more invalidations occur.

Specialization Specialization refers to the principle of dividing the implementation of an operation into smaller pieces intended to handle specific cases, and speculating that only a limited set of code paths will be explored during program execution, which can greatly simplify the work of an optimizing runtime and compiler. A common use of this is type specialization, with one code path for each data types, but an implementation may also specialize (and have separate code paths) for positive or negative values, values below or above a certain threshold, etc.

These fundamental techniques permeate the entire implementation of FastR, starting with symbol lookup, through lazy evaluation, to other performance-critical implementation components such as function calls and vector accesses.

5.1 Symbol Lookup

The seemingly simple task of looking up a symbol representing a variable or function name is a complex operation in R, due to the way the lookup scopes are chained as user-accessible and user-modifiable first-class objects.

A mapping between symbols and their values in R is represented by an *environment*. If a symbol is not found in an environment, lookup continues automatically in the *enclosing environment*. Certain environments are pre-defined at startup, for example the *base* environment containing definitions of builtin functions and system variables, or the *global* environment representing the outermost scope for user-defined variables and functions. A linked list of (enclosing) environments starts at the global environment, continues with *package environments* containing symbols defined by currently loaded packages, and ends at the *base* environment. This list of environments is called the *search path*, and all lookups for basic operations like `<-` (assignment) or `length` (builtin function returning length of an R data structure) will search through all of them before finding the target function in the base environment.

New environments are created upon each function call to create a scope for the function's local variables. The function environment gets its enclosing environment from the function's enclosing environment. A function's enclosing environment is initially taken from the point where the function was created (this can be an outer function environment, or the *global* environment at the top level), but it can be modified freely.

The main difference between R and other superficially similar languages, such as Java or JavaScript, is that environments can be explicitly added and removed to and from the search path, for example by loading or unloading R packages. Furthermore, R programmers can create new environments and use the builtin functions `attach` and `detach` to manipulate the search path. Modification to variables in most cases only modify the current environment, but R also provides variable access capabilities that can reach out to the outer scope(s), for example, the super assignment operator `<<-` that assigns in the enclosing scope. The combination of all these features makes symbol lookup very dynamic and difficult to optimize – a naïve implementation would perform a full lookup each time irrespective of whether the symbol binding or the search path has changed or not.

FastR uses Truffle facilities so that the actual contents of the environment (*frame* at the Truffle level) and the set of names that may be available in an environment (called *frame descriptor* and consisting of *frame slots*) are separate entities. A function, for example, creates a new environment for each call, but the frame descriptor is shared by all these environments. While new environments are created all the time, only a limited number of frame descriptors is created, and the set of names they may contain will stabilize over time.

Given that functions always use the same frame descriptor, the mapping between a name of a local variable and the actual position in the frame only needs to be determined the first time the function is executed, and can be cached for subsequent execution. The optimizing Truffle runtime can elide allocating the function environment altogether in many cases through escape analysis [20].

In order to accelerate R language's symbol lookup process while remaining faithful to its semantics, for non-local lookups FastR creates Truffle *assumptions* for a number of system-wide conditions:

- For a given frame descriptor, an *assumption* asserts that there is only a single known environment using the frame descriptor. Constructing a second environment with the same layout will invalidate this assumption.
- For a given frame descriptor, an *assumption* asserts that the frame descriptor of enclosing environments does not change. For example, the frame descriptor of the environment of an inner function will always have as an enclosing frame descriptor the frame descriptor of the environment of the outer function. While the actual environments change, the frame descriptors and their relationships usually stay the same.
- For a given frame descriptor, an *assumption* asserts that specific names are *not* available in this frame descriptor.
- For a given frame descriptor (which has a single, known instance – see the first condition), frame slot-specific *assumptions* assert that the symbol-to-value binding does not change, that is, a given symbol always corresponds to the same value.

Inserting new names into an environment, changing the nesting of environments, or changing values inside environments can potentially invalidate these assumptions. These are either very infrequent operations, or the invalidation only happens the first time they are executed, so that overall the time needed for invalidating assumptions is negligible.

Consider the following piece of R code that defines function `f1` at the global scope and `f2` as an inner function in `f1`:

```
f1 <- function(x) {
  f2 <- function(x) {
    length(x)
  }
  f2(x)
}
f1(42)
```

Note that the result of the last statement in a function is the return value of this function. When the call to `f1` is executed, it will in turn call `f2` and return the length of the parameter (in this case – 1). In order to find the actual function that the symbol `length` is bound to in `f2`, FastR will discover and check the following assumptions:

- `length` is not available in `f2`'s frame descriptor
- enclosing frame descriptor of `f2` is `f1`
- `length` is not available in `f1`s frame descriptor.
- enclosing frame descriptor of `f1` is the frame descriptor of the *global* environment
- `length` is not available in the frame descriptor of the *global* environment
- ... (these steps continue through the remainder of search path until reaching the *base* environment)
- `length` is available, with a specific known value (the `length` primitive) in the *base* environment

Almost all non-local lookups in R code can be satisfied by only checking *assumptions*. The first time a lookup is executed, FastR collects the set of *assumptions* needed to guarantee that the lookup result remains valid. For subsequent executions, only the assumptions need to be checked, which does not incur any runtime cost as soon as the code is optimized.

If FastR detects that a symbol lookup is not stable and cannot be fulfilled using assumptions, it will fall back to doing the full lookup every time. This is a rare case in practice.

5.2 Lazy Evaluation

R uses a *call-by-need* lazy argument evaluation strategy – a promise (see Section 2), consisting of a code snippet and its evaluation environment, is used to compute a concrete value of an argument. The actual value of the promise is calculated (*forced*) as late as possible and only if it is needed. A promise is only forced once, and the computed value is cached within the promise.

A straightforward implementation of *call-by-need*, such as that of GNU R, is rather simple, but as other researchers [4] observed, it is typically less efficient than eager argument evaluation strategies, such as *call-by-value*. Creating promises and going through an additional level of indirection to acquire the actual value is a noticeable overhead, but for optimizing runtimes the main problem is that every point in the program that can potentially force a promise becomes a call site that can execute arbitrary code.

There are two basic approaches of bringing the performance of lazy evaluation closer to that of the eager ones but, as we discuss in Section 7, they cannot be used with FastR. They either use static analysis or rely on language features to control side-effects, while FastR uses JIT compilation and has largely uncontrolled, if infrequent, side effects.

The main idea underlying FastR's approach to implementing lazy evaluation is that not all promises are created equal, and different promise categories require different optimization approaches.

5.2.1 Categorizing Promises

We distinguish the following promise categories:

Eager promises represent arguments where a local variable is used as a parameter:

```
x <- 17; foo(x)
```

Indirect ("promised") promises represent arguments that were not forced yet and are passed as parameters to subsequent function calls:

```
bar <- function(x) { foo(x) }
```

Default ("complex") promises represent all other promises, which can contain arbitrary code that needs to be evaluated to obtain the concrete argument value:

```
foo(x + bar(y))
```

Even though there are no systematic studies on what kinds of promises dominate in R code, the authors' experience working with R applications and standard libraries suggests that even such a simple classification creates opportunities for improving lazy evaluation performance, and this paper seeks to back up this hypothesis with the experimental results.

5.2.2 Implementing Promises

Given that they provide the largest potential for optimizations, as the overhead related to these promises can be removed almost completely, FastR focuses its efforts on eager promises. When FastR determines that the argument to a function call is an eager promise, that is, it passes along a local variable, it evaluates the local variable and stores the value of the argument with an eager promise. While this evaluation clearly has no side-effects, the eagerly evaluated value is correct at the point when the promise is forced only if the input variable involved in the argument value computation has not changed in the meantime (e.g., through use of the `<<-` super assignment operator). The eager value also cannot be trusted any more if functions that hand out references to the environment, like `environment()`, are used. We monitor these invariants by associating a Truffle *assumption* with each eager promise – if an input variable changes or the environment escapes, the *assumption* is invalidated, thus invalidating the pre-computed value. In this rare case, the value must be re-computed from the complete promise.

Whenever we need to actually evaluate an eager promise, we need its evaluation environment. Since storing the actual environment in eager promises would break many optimizations (e.g., Truffle *frames* can no longer be "virtualized" and turned into native stack frames), and thus make eager promises less efficient, FastR stores a *marker* that can be used to locate (by traversing the execution frame stack) the correct environment. If the eager promise *assumption* holds, the performance of eager promises is close to call-by-value, because all that is needed is a small amount of additional context for each parameter. On the other hand, on the slow path (after re-computation is triggered), evaluation of eager promises degenerates to that of default promises, and the call site that originally generated the eager promise is instructed to generate default promises.

The implementation of default promises is straightforward and conceptually similar to that of GNU R. A default promise contains a Truffle node representing the argument expression along with the associated (caller) environment. In this case FastR pays the cost associated with passing along the environment, but it can still accelerate the actual computation of the value at the point where the promise is forced. FastR creates inline caches for points in the code that repeatedly evaluate promises with similar expressions. This allows the optimizing Truffle runtime to incorporate the code for the specific expressions, thereby exposing more opportunities for further optimizations.

Indirect promises, while technically being an instance of eager promises (in that their creation does not involve any expensive operations, such as passing along the environment), in reality are simply wrappers around default and eager promises and their evaluation performance is that of the promise they are wrapping.

5.3 Function and Method Calls

From a certain point of view, R is a functional language in that functions in R are first-class citizens of the language – they can be stored in variables, passed as arguments, returned from other functions, etc. As described in Section 2, many language constructs that in other languages are purely syntactical (e.g., curly braces enclosing a code block), are expressed as function calls in R. Fortunately, since functions are stored in variables, function lookup proceeds similarly to the "generic" symbol lookup described in Section 5.1, with the difference beeing that non-function data structures are skipped during lookup. Consequently, if neither the variable storing the function nor the search path changes, the function definition to be used at a given call site remains constant.

The function calling procedure is non-trivial as not only there are many different function types but also R uses a complicated algorithm for matching function parameters to its arguments. Therefore we specialize function calling procedure based on the type of function being called and utilize caching of *argument signatures* [7] to accelerate the argument matching procedure.

R language function types include "regular" functions implemented in R, builtin functions implemented in the host language of the R language implementation, and native functions implemented externally in one of the "native" languages (Fortran, C or C++). Different function types are called differently, and the code handling their invocations is carefully tailored to handle different cases. For example, invocations of native functions and of most of the builtin func-

[7] As opposed to the static notion of *function signature*, we use the term *argument signature* to describe a dynamic ordered list of arguments used for a given function invocation.

tions require their arguments to be evaluated (i.e., promises representing arguments to be forced), and certain types of builtin functions use a different simplified version of the argument matching algorithm.

This is additionally complicated by the fact that, in addition to being functional, R is also object-oriented, with the most popular object model, that is S3, being tightly integrated with the function execution machinery [8].

5.3.1 S3 Object Model

As mentioned in Section 2, any R data structure, for example a vector, can have a list of *attributes*, that is name-value pairs, associated with it. S3 method dispatch is based on the content of one such attribute, called `class`. To illustrate how method dispatch interleaves with regular function invocations, consider the following example, where we first create a one-element vector and then define a "regular" function returning the content of this vector increased by 10:

```
v<-structure(42)
f<-function(x) { x + 10 }
```

Unsurprisingly, calling `f` on vector `v`, `f(v)` results in generating the value 52. However, after we redefine `f` to be a generic function:

```
f<-function(x) { UseMethod('f') }
```

we can create a vector with a single `class` attribute and an S3 method (which is, for all intents and purposes, also an R function) that will be chosen by the runtime (via the `UseMethod` builtin) to execute if its argument has the same class attribute as the method's suffix (after the dot):

```
v<-structure(42, class='foo')
f.foo<-function(x) { x - 10 }
```

In this case, the same invocation uses the generic function, calls the defined method, and generates the value 32. This is called *generic dispatch*, and not only "regular" R functions can serve as generic functions, but also builtin functions, for example those representing vector access and arithmetic operations. This category of builtins implicitly performs a dispatch largely similar to `UseMethod`.

R also supports *group generic dispatch*, where programmers can define methods that will be executed for groups of (builtin) functions instead of for just a single function as in the example above. The function groups are predefined (e.g., functions representing arithmetic operators) and can be useful when defining operations for custom data structures.

When the `UseMethod` builtin function is called, or a primitive operation with generic dispatch is encountered, FastR needs to perform S3 lookup. If the argument does not have a class attribute, then in most cases no additional action is necessary, and function located via symbol lookup (often by analyzing *assumptions* only) can be used directly. In case a class attribute was found, the lookup progresses along a well-defined sequence of function names that need to be considered. For example, if the + operation is executed with arguments of class `foo`, then `+.foo` (generic dispatch) and `Ops.foo` (group generic dispatch since + belongs to the `Ops` group) need to be looked up, and only if none of these functions exists, the actual primitive operation will be executed.

FastR builds an inline cache based on the classes that were seen before, so that the list of function names to be looked up can be precomputed. The lookups themselves can usually all be fulfilled using assumptions, so that the S3 dispatch can subsequently be executed very quickly.

5.3.2 Argument Matching

The argument matching algorithm in R is unusually complicated - it involves positional arguments, named arguments (with partial name matching), default arguments, and variadic arguments. The algorithm, which creates a plan for reordering the arguments from the argument signature at the call site to the formal signature of the target function, consists of four phases:

- Match named arguments that have an exact match in the formal signature.
- Match named arguments with a partial (but unambiguous) match in the formal signature.
- Fill the remaining formal arguments with the remaining actual arguments (positional matches).
- Put all remaining arguments into the variadic argument (expressed by `...`) if present.

For example, considering the following function definition (`..1` and `..2` retrieve first and second argument from the list of variadic arguments):

```
f<-function(a, b, carg, ..., d=6) {
  list(a, b, carg, ..1, ..2, d)
}
```

The function invocation `f(b=2, 1, c=3, 4, 5)` will result in a list consisting of consecutive numbers between 1 and 6: `b` and `carg` arguments are matched first (with `c` being a partial, but unambiguous, match), followed by value 1 being matched to the first unmatched argument `a`, then values 4 and 5 are matched to the variadic argument, and finally the default value is used for argument `d`.

In GNU R, the argument matching algorithm is run for every function invocation. FastR optimizes argument matching in several ways. Argument signatures, which consist of an ordered list of argument names, are *interned* in a global data structure, so that comparing for equality of signatures is simply an identity comparison between signature objects. In places where FastR performs the argument matching algorithm, it also creates an inline cache of argument and formal signatures. If a subsequent function invocations uses

[8] Another object model available in R, that is S4, is a much later and less tightly integrated addition. Even though we implement it in FastR, we will not discuss it further as its performance is considered less critical to the point of some companies discouraging its use in their code [8].

the same signatures, the same argument permutation can be reused. An inline cache size of four elements is enough to elide falling back to the complex matching logic in all but an insignificant number of cases.

5.4 Vector Accesses

R's main data types are vectors, there are no scalar numeric data types, that is, even the number 42 is a 1-element numeric vector. In order to make these vector data types as useful as possible, vector access operations are very powerful and support many different filtering and selection use cases. Elements of a vector can be extracted and replaced using dynamic single and multi-dimensional (for accesses to matrices and multi-dimensional arrays) indexes. Indexes can consist of multiple (for multi-element accesses) logical values, positive or negative integer values, or string values. Vector accesses are very common in R, which is why FastR goes to great lengths to optimize these operations using inline caching and specialization.

As a first level of optimization we split the operation using an inline cache for patterns that are usually stable for a particular vector access operation. For example, consider the following two-dimensional *target* vector (i.e., a matrix) `vec` with named columns and rows:

```
     col1 col2
row1    1    3
row2    2    4
```

and the following two-dimensional *value* vector `val`:

```
     col1 col2
row1    7   42
```

In order to replace both values of the target vector in "row1" with the content of the value vector, we would use the `vec["row1", 1:2] <- val` replacement operation.

For these kinds of vector replace operation, we cache for each combination of:

- type of the target vector (in this case – `double`)
- number of dimensions of the value vector (in this case – 2)
- types of values used for indexing (in this case – `String` for the first index and `double` for the second)
- type of the value vector (in this case – `double`)
- type of the replacement operation (R uses two different replacement operators, with slightly different semantics: `[<-` and `[[<-`)

Please note that the example is only one of many different vector access variants, and in order to make the code for all these combinations maintainable, we develop these operations against a set of generic interfaces. By injecting a concrete type from the inline cache, the Truffle partial evaluator is able to remove all expensive interface calls reliably in the optimized code.

The guards for the vector inline cache are based on how vector accesses are used in R, so that most vector operations have just a single entry in the inline cache (see Section 6.4 for detailed numbers). However, in cases where the operation is extremely polymorphic, changing types and dimensions continuously, we fall back to using the least recently used (LRU) caching strategy. The LRU cache does not become stable, so the operation will run in a more generic and less efficient mode in such cases.

FastR tries to keep the code for vector access operations as small as possible at runtime, by hiding expensive pieces of functionality behind small and efficient checks. If one of these checks fails, FastR will deoptimize and recompile to accommodate the new situation, but will not insert a new line into the inline cache. For example, FastR speculates on the vector length to always be constant if the vector length is below a certain threshold [9]. This exposes additional optimization opportunities, for example unrolling loops iterating over the content of a vector. If FastR encounters multiple target vector length values or values which are greater than the threshold, it assumes a dynamic value for the target vector length. We found the following list of aggressive specializations beneficial for the vector replace operation:

- target vector is not shared and can be reused
- constant length of target, index and value vectors if below a certain threshold
- constant number of positions selected if below or equal a certain threshold
- dimension indices contain NA ("not available") values
- has only positive or only negative vector indices
- has no zero vector indices
- for string based indexing: string indices mapped to the same integer indices
- has not seen any error condition (e.g., out of bounds)

In addition, many special checks are needed in order to provide exactly the same behavior as GNU R in corner cases. While these corner cases might seem unimportant, they are exercised by existing R code, and can lead to hard-to-debug problems if ignored.

6. Experimental Results

In our performance evaluation we demonstrate the effects of our lazy evaluation optimizations in Section 6.3. We also pitch FastR against the current state-of-the-art – in Section 6.5 we compare the following R implementations and configurations:

- **GNU R "base"** – a default configuration of GNU R using AST interpretation

[9] Currently the threshold is experimentally set to 4

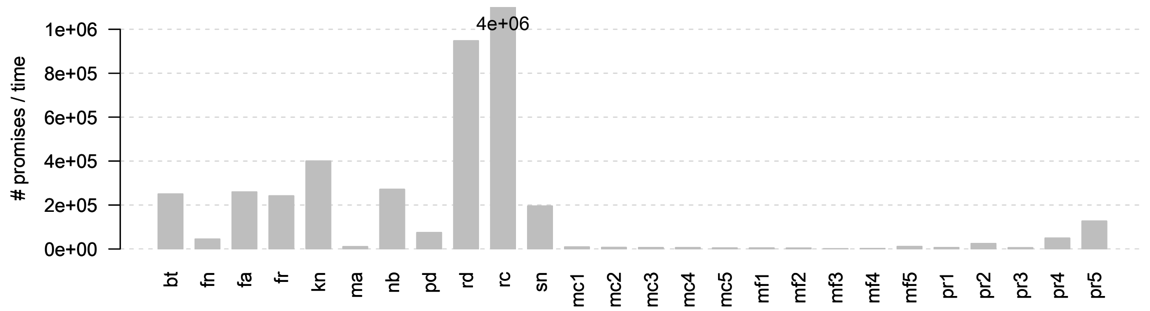

Figure 1. Promise statistics

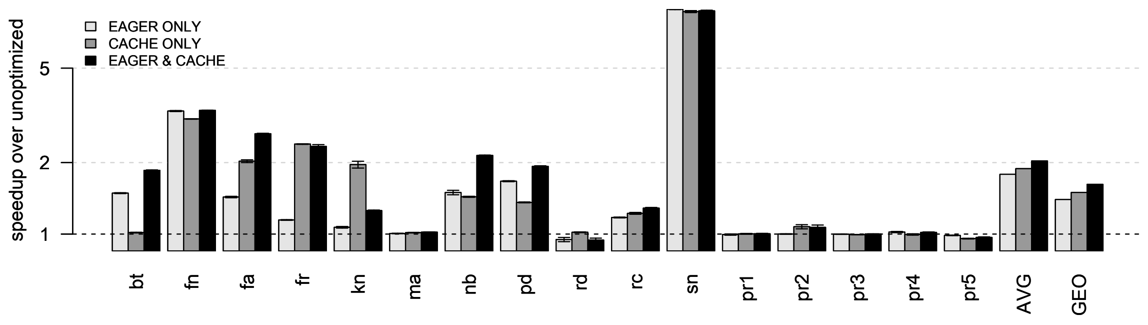

Figure 2. Impact of lazy evaluation optimizations

- **GNU R "BC"** – a configuration of GNU R using byte-code interpreter
- **Renjin** [3] – alternative implementation of R executing R programs on top of a JVM
- **TERR (TIBCO Enterprise Runtime for R)** [22] – alternative implementation of R language runtime in C/C++
- **FastR**

The reason for choosing Renjin and TERR for our performance comparison, is that they are the most complete "production-ready" non-GNU R-based alternative R implementations, with all three platforms (including FastR) offering a similar level or R language semantics compatibility.

6.1 Benchmarks

To the best of our knowledge, no official R benchmark suite exists and, additionally, many existing R applications rely on packages with native components that FastR cannot yet load and run. We then follow the route taken by other researchers [11] that used *shootout* [6] and *b25* [24] benchmark suites, and also modified the latter so that it aggregates final computation results[10]. The shootout suite benchmarks are R implementations of problems created for Computer Language Benchmarks Game [6] and consist of small applications (such as binary tree manipulation or n-body simulation – see benchmark description [6] for details), consisting mostly of R code and stressing different aspects of the language implementation. The *b25* benchmark suite is divided into 3 subgroups consisting of 5 benchmarks each. The first two (*matcal* and *matfunc*) involve matrix calculations and spend majority of time in builtin functions and in native code and are, as such, not particularly interesting from the perspective of improving performance of R code. We include the performance numbers for these two groups merely to indicate that they perform similarly in FastR and GNU R, even though the FastR implementation in some cases involves crossing the Java-to-native boundary. The last *b25* benchmark subgroup (*prog*) represents simple R computation tasks (such as matrix transposition or finding grand common divisors – see the benchmark description [24] for details).

In the plots, we abbreviate the names of the *shootout* benchmarks as follows: **bt** (*binary-trees*), **fn** (*fannkuch-redux*), **fa** (*fasta*), **fr** (*fasta-redux*), **kn** (*k-nucleotide*), **ma** (*mandelbrot*), **nb** (*n-body*), **pd** (*pidigits*), **rd** (*regex-dna*), **rc** (*reverse-complement*), and **sn** (*spectral-norm*). For *b25* benchmarks we use **mc** for *matcal*, **mf** for *matfunc* and **pr** for *prog* subgroups.

6.2 Configuration

The machine we used for running our experiments is an Intel Xeon with 24 2.70GHz E5-2697 v2 CPU cores (Ivy Bridge) between two sockets, featuring 264G for RAM and running Oracle Linux Server 6.5 (a derivative of Red Hat Enterprise Linux Server 6.5), with Linux kernel version 2.6.32. Please

[10] Otherwise, due to lazy evaluation, actual computation may be skipped.

note that while the machine is parallel, both R language and benchmarks in both our suites are inherently sequential, and the benefits of parallel architecture can only be observed if the language implementation supports implicit parallelism. Currently only Renjin claims existence of such support, with both GNU R and FastR focusing on sequential performance.

All benchmarks, except for the runs used to gather statistics, are executed within a common harness, with each benchmark application being executed in a separate process. For each benchmark application, the harness executes a certain number of "warmup" iterations followed by a "measurement" run. The intention here is to present the peak-performance numbers since FastR is targeting long-running applications where warmup-time effects (such as compilation cost) are negligible when compared to total execution time. For all performance runs (Figure 2, Figure 3, and Figure 4) we ran each benchmark five times and plot an average over all runs with 95% confidence intervals. Since the amount of jitter was small, and since all these figures are plotted on a logarithmic scale, confidence intervals are only really visible in Figure 2. We use GNU R version 3.2.4, a version of FastR compatible with GNU R 3.2.4, Renjin version 0.8.2050, and TERR version 4.1.1. We use jdk1.8.0_60 for FastR and Renjin executions. We evaluate different R implementations in their default configurations, much like those that would be most likely used by end users, with the only exception being setting 6G max heap size for Renjin and FastR runs. We did, however, also ran a configuration where GNU R, TERR, and FastR use the same version of native LAPACK and BLAS libraries [11] and the results were virtually the same as with the default configuration runs.

6.3 Lazy Evaluation Optimizations

Lazy evaluation overheads are meaningful with respect to the total execution time of an application only if said application creates a significant number of promises that need to be maintained and evaluated. In particular, applications that spent most of the time in native code and/or executing builtin functions, such as those that belong *b25*'s *matcal* and *matfunc* benchmark groups, are unlikely to suffer from lazy evaluation performance problems and, consequently, benefit from lazy evaluation optimizations. In order to estimate optimization potential, we ran each benchmark only once, measured the number of promises created throughout its execution, and normalized this number with respect to the benchmark's execution time. As we can see in Figure 1, the *b25*'s *matcal* (**mc**) and *matfunc* (**mf**) benchmark groups indeed provide almost no opportunity for performance improvement. Consequently, we report lazy evaluation optimization numbers only for the *shootout* benchmarks and the *prog* category of the *b25* benchmarks.

In Figure 2 we present speedup of different promise optimization configurations over the unoptimized case plotted on a log scale (values greater than 1 indicate speedups and smaller than 1 indicate slowdowns). The right-most bar (*EAGER & CACHE*) represent the impact of both inline caching and of utilizing eager promises. We observe only few minor slowdowns resulting from applying the optimizations, with speedups over the unoptimized implementation reaching over ∼8x and average performance doubled (with geometric mean at over ∼1.6x). The left-most (*EAGER ONLY*) bar represents a configuration that only enables eager promises and the middle bar (*CACHE ONLY*) represents a configuration that only enables inline caching – their analysis indicates that one size does not fit all. Sometimes the benefit of caching is more pronounced, as evidenced by *shootout*'s *k-nucleotide* (**kn**) benchmark, and sometimes it is eager promises that provide majority of the speedup, as demonstrated by the execution times of *shootout's binary-trees* **bt** benchmark.

6.4 Vector Access Inline Cache

For our vector access optimizations to be effective we rely on inline caches to remain stable. Therefore we counted the number of inline cache entries for replace and extract operations for the *b25* and *shootout* benchmarks. In total we found 96 replace operations and 1182 extract operations. 74 (∼77%) of the replace and 1127 (∼95%) of the extract operations required only a single entry in the inline cache. We counted two entries for 16 (∼17%) replace and 43(∼4%) extract operations, three entries for two (∼2%) replace and 6 (∼0.5%) extract operations and four entries for two (∼2%) replace and three (∼0.3%) extract operations. The main observation here is that when running the benchmarks, the inline cache has never overflowed (in fact, the fifth and last entry in the cache was unused for all operations), and none of the operations used in the benchmarks required the runtime to fall back to the LRU caching technique.

6.5 FastR vs the World

While understanding the impact of specific optimizations can be very useful, the real test of our system is its performance against the current state-of-the-art. Before we proceed to describing the performance numbers, it is important to re-iterate that the current main goal of FastR project is to accelerate execution of R code – in particular, we defer further research on improving performance of native code to related work. In Figure 3 and Figure 4 we demonstrate speedup of GNU R's bytecode interpreter, Renjin, TERR and FastR, over "base" GNU R, and plotted on a log scale (again, values greater than 1 indicate speedups and smaller than 1 indicate slowdowns). As we can see, GNU R's bytecode interpreter brings modest performance gains, with ∼1.7 average (∼1.5 geomean) speedup over "base" GNU R for *shootout* benchmarks and ∼1.2 average (∼1.1 geomean) speedup for *b25* benchmarks. Performance of Renjin is unstable, with significant speedups for some benchmarks, such as *b25*'s *prog-1* (**pr1**) and *prog-2* (**pr2**) benchmarks in Figure 4), but also dramatic slowdowns, such as for *b25*'s *prog-4* (**pr2**) bench-

[11] Despite our best efforts we could not make Renjin use the native libraries.

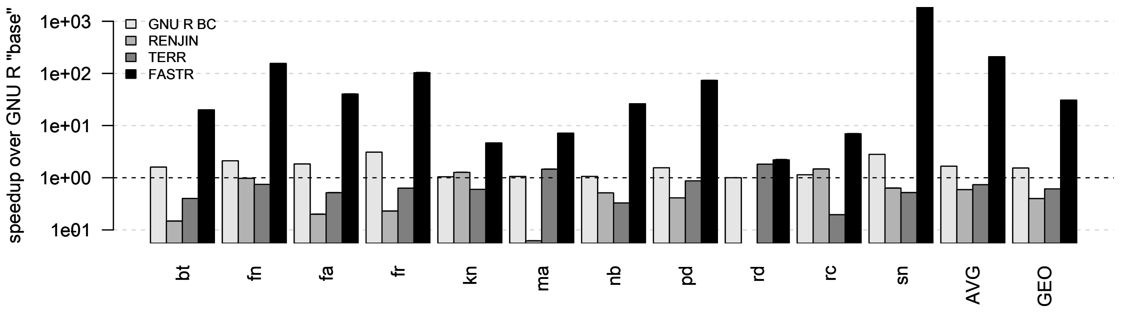

Figure 3. Performance numbers for *shootout* benchmark suite

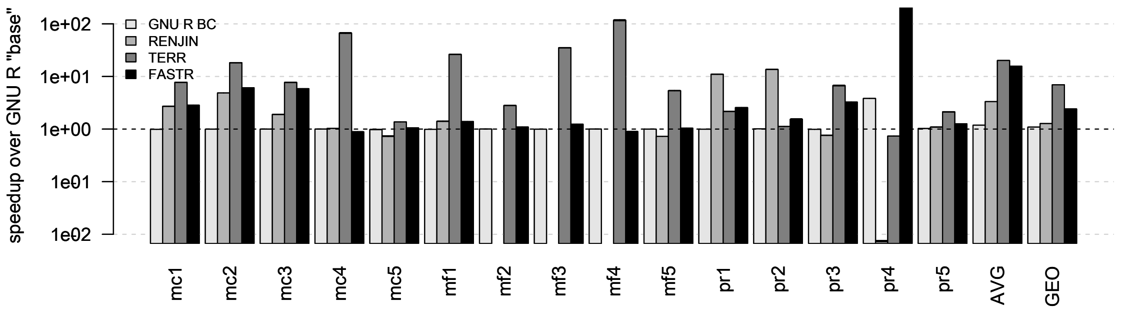

Figure 4. Performance numbers for *b25* benchmark suite

mark in Figure 4. Renjin is also unable to run some benchmarks: *shootout*'s *regex-dna* (**rd**) benchmark and 3 out of 5 *b25*'s *matfunc* (**mf**) benchmarks. Overall, Renjin performs worse than "base" GNU R on *shootout* benchmarks (∼0.6 average score with ∼0.4 geomean) and its average speedup on *b25* benchmarks is ∼3.3 (geomean – ∼1.3). Judging from results of *b25* benchmark executions, TERR appears to be the main competitor of FastR in terms of performance, with TERR's ∼20.1x average speedup (geomean at ∼7.0) and FastR's average speedup of ∼15.7x (and geomean at ∼2.4). What works in TERR's favor here are *b25*'s *matcal* (**mc**) and *matfunc* (**mf**) benchmarks that contain mostly calls to builtin and native functions. TERR, as we suspect (it is closed source), utilizes heavily optimized custom implementations of builtins used by *b25's matcal* and *matfunc* benchmarks (e.g., `matrix`) and it clearly currently has advantage over FastR in terms of crossing the native boundary. However, when considering the remaining benchmarks, not only does FastR perform better then TERR on average, but on *shootout* benchmarks TERR exhibits slowdowns with respect to "base" GNU R (∼0.7 average and ∼0.6 geomean). At the same time, FastR suffers no significant slowdowns, and some of its performance gains are considerable reaching into thousands times speedup over "base" GNU R. The outliers clearly inflate FastR's average performance gain (∼208.7x speedup on *shootout* and ∼15.7x speedup on *b25* benchmarks), but even geometric means across the board (∼30.8x and ∼2.4 speedups, respectively) are arguably impressive considering that we support full R language semantics with all its idiosyncrasies.

In terms of larger R programs, our focus so far was on the internal production applications which we cannot describe and evaluate in detail at the moment, but which FastR executes up to 3x faster than GNU R.

7. Related Work

In addition to FastR, Oracle Labs is developing Truffle-based implementations for JavaScript and Ruby [27]. Truffle-based implementations of Python [26] and SOM [14], a Smalltalk-like language, have been developed by third-parties.

There are several alternate implementations of R in addition to FastR. TERR [22] is a closed source clean-room implementation of R (in C/C++) aimed at enterprise-grade applications. Renjin [3] is a JVM-based interpreter for R, and follows the path taken by several other languages, for example JRuby [10], in targeting the JVM bytecode interface. However, it is challenging to map semantics of languages that significantly differ from Java to the JVM bytecode interface, as noted by the JRuby project which is now experimenting successfully with a Truffle-based solution.

In addition to "pure" GNU R, a few derivatives of this project exist. One example is CXXR [19] – an effort started in 2008 to clean up the implementation by refactoring it into C++. After lying dormant for several years CXXR has recently been picked up by Google and renamed rho [15], with a focus on improving memory management and performance using LLVM [12]. pqR [16] ('pretty quick R') is another GNU R variant that tries to improve on performance by various interpreter modifications. The last category of alternative R implementations are research prototypes that implement only a portion of the full R language semantics. These include Purdue-FastR [17], a fast simplified R interpreter, and Riposte [21], an experimental R implementation that focuses on optimizing vector access operations.

Other frameworks aim to provide similar benefits to Truffle/Graal, the most notable being RPython [1], a statically typed subset of Python that uses trace-based JIT compilation to eliminate most of the interpreter overhead. RPython has been used to implement Python and and other dynamic languages, for example JavaScript and Ruby.

There is also a large body of work on optimizing performance of lazy evaluations. Arguably the most relevant work (in particular, approaches relying on static analysis [5, 25] are ill-suited for FastR), was done in Haskell and attempts to bridge the gap between lazy evaluation and eager evaluation by optimistically trying eager evaluation [4, 13]. These approaches involve complicated and often expensive methods of restarting computations upon eager computation failure, but their main drawback from FastR's perspective is that they rely on language mechanisms to control side-effects.

8. Conclusions

The R language, with its rich set of features and highly dynamic nature, presents unique challenges to language developers, and few traditional compiler techniques can be applied if compatibility with the complete set of dynamic language features is needed. In this paper, we showed how FastR uses the concepts of inline caching, assumptions and specialization extensively to extract the stability needed for efficient execution of such a dynamic language. Experimental results show that FastR greatly benefits from this – it can accelerate R code execution by orders of magnitude while keeping the original language semantics.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *DLS 2007*, 2007.

[2] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The new S language: a programming environment for data analysis and graphics*. Chapman and Hall/CRC, 1988.

[3] BeDataDriven. Renjin, 2016. URL `http://www.renjin.org/`.

[4] R. Ennals and S. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP 2003*, 2003.

[5] K.-F. Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In *ICFP 2000*, 2000.

[6] B. Fulgham and D. Bagley. The computer language benchmarks game, 2013. URL `http://benchmarksgame.alioth.debian.org`.

[7] Y. Futamura. Partial evaluation of computation process&mdash;anapproach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4), 1999.

[8] Google. Google's r style guide. URL `https://google.github.io/styleguide/Rguide.xml`.

[9] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3), 1996.

[10] JRuby. JRuby: The Ruby programming language on the JVM, 2016. URL `http://jruby.org`.

[11] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A fast abstract syntax tree interpreter for r. In *VEE 2015*, 2015.

[12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*, 2004.

[13] J.-W. Maessen. Eager Haskell: resource-bounded execution yields efficient iteration. In *ICFP 2002*, 2002.

[14] S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *OOPSLA 2015*, 2015.

[15] K. Millar. rho, 2016. URL `https://github.com/kmillar/rho`.

[16] R. Neal. pqr, 2015. URL `http://www.pqr-project.org/`.

[17] Purdue. purdue-fastr, 2015. URL `https://github.com/allr/purdue-fastr`.

[18] Revolution Analytics. Revolution Analytics: a 5-minute history. In *UseR!*, 2014.

[19] A. Runnalls. CXXR, 2008. URL `https://www.cs.kent.ac.uk/projects/cxxr/`.

[20] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *CGO '14*, 2014.

[21] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: A trace-driven compiler and parallel vm for vector code in r. In *PACT 2012*, 2012.

[22] TIBCO. TERR, 2016. URL `https://docs.tibco.com/products/tibco-enterprise-runtime-for-r`.

[23] R. Tsegelskyi and J. Vitek. Testr: generating unit tests for r internals. In *UseR!*, 2014. URL `http://user2014.stat.ucla.edu/abstracts/talks/129_Tsegelskyi.pdf`.

[24] S. Urbanek. R benchmark 2.5, 2008. URL `http://r.research.att.com/benchmarks`.

[25] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the Functional Programming Languages and Computer Architecture*, 1987.

[26] C. Wimmer and S. Brunthaler. Zippy on truffle: A fast and simple implementation of python. In *SPLASH Demo 2013*, 2013.

[27] 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 *SPLASH Onward! 2013*, 2013.