R in Java
FastR: an implementation of the R language

Petr Maj
Tomas Kalibera
Jan Vitek
Floréal Morandat
Helena Kotthaus

Purdue University & Oracle Labs
https://github.com/allr
What we do...

- **TimeR** — an instrumentation-based profiler for GNU-R
- **TracR** — a trace analysis framework for GNU-R
- **CoreR** — a formal semantics for a fragment of R
- **TestR** — a testing framework for the R language
- **FastR** — a new R virtual machine written in Java
Why?

... language for data analysis and graphics

... used in statistics, biology, finance ...

... books, conferences, user groups

... 4,338 packages

... 3 millions users
Scripting data

read data into variables
make plots
compute summaries
more intricate modeling
develop simple functions
to automate analysis
R history

• 1976 S  
  John Chambers @ Bell Labs, then S-Plus (closed-source owned by Tibco)

• 1993 R  
  Ross Ihaka and Robert Gentleman, started R as new language at the University of Auckland, NZ

• Today, The R project  
  Core team ~ 20 people, released under GPL license. Continued development of language & libraries: namespaces (‘11), bytecode (‘11), indexing beyond 2GB (‘13)

http://www.r-project.org  
http://cran.r-project.org
What R is…
- vectorized
- functional
- object-oriented
- lazy
- portable
- interactive

What R isn’t…
- fast
- low-footprint
- concurrent
- distributed
- formally specified
- standardized
PARAMOUNT PRESENTS

ALFRED HITCHCOCK'S

THE TROUBLE WITH R

Color by TECHNICOLOR
To understand where the time is spent, we turn to more representative R programs. Fig. 7 shows the breakdown of execution times in the Bioconductor dataset obtained with ProfileR. Each bar represents a Bioconductor vignette. The key observation is that memory management accounts for an average of 29% of execution time. Memory management time was further broken down into time spent in garbage collection (18.7%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. The time spent in built-in functions represents the true computational work performed by R, this is on average 38% of execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and there is a program that spends 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position of variables in a frame are known. Given the nature of R, many of the core numerical functions are written in C or Fortran. This can lead to the perception that execution time is dominated by native libraries. Looking at the amount of time spent in calls to foreign libraries.
Call-by-value semantics are implemented by a copy-on-write (COW) mechanism. Thus, under the covers, arguments are shared and only duplicated if there is actually a need.

Another reason for the large footprint is that all numeric data has to be boxed into a vector. An empty vector is 40 bytes long; 10 cells are 56 byte long, and take on allocating lists is greater than testing to note that the time spent in function call arguments. It is interesting to see that the time spent in foreign functions shows that this is clearly not the case. On average, the time spent in foreign function call arguments is greater than 22% of the run-time.

Not only is R slow, but it also consumes significant amounts of memory. Unlike C, R has to avoid cache pollution. This is largely due to the combination of language features (call-by-value, extreme dynamism, lazy evaluation) and the lack of efficient built-in types.

Lists are created by the R virtual machine. The R allocation is split between vectors (which are mostly used by the interpreter for, e.g., arguments to user-defined functions) and lists (which are mostly used by the interpreter for, e.g., arguments to user-defined functions). The graph clearly shows that R allocates orders of magnitude more data than in C.

The programming language shootout compares heap memory usage in C (calls to malloc) and data allocated by R.

The programming language shootout was run on an Intel X5460, 3.16GHz, Linux 2.6.34. R 2.12.1, GCC v4.4.5.
is dominated by native libraries. Looking at the amount of time spent in calls to foreign functions are written in C or Fortran. This can lead to the perception that execution time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. The maximum spent in garbage collection is 70% and there is a program that spends 29% of execution time. Memory management time was further broken down into time spent in duplication (2.6%), and duplications (3.6%), vec-

To understand where the computational work performed is concentrated, we turn to more detailed examination times in the Bioconductor dataset obtained with ProfileR. The key observation is that memory management time accounts for an average of 50% of execution time. There are some interesting outliers. For example, the Shootout benchmarks show the breakdown of execution time for representative R programs. Fig. 6.

Fig. 6. Evaluation of the Design of R

Name | Input | Time Spent
--- | --- | ---
S-1 | Binary trees 16 | 0.0
S-2 | Fasta 2.5M | 0.1
S-3 | Fasta redux 2.5M | 0.2
S-4 | K-nucleotide 50K | 0.3
S-5 | N-body 500K | 0.4
S-6 | Mandelbrot 4K | 0.5
S-7 | Pidigits 500 | 0.6
S-8 | Regex-dna 2.5K | 0.7
S-9 | Rev. complement 5M | 0.8
S-10 | Spectral norm 640 | 0.9

Fig. 7. Slowdown of Python and R, normalized to C for the Shootout benchmarks.

Slowdown of Python and R, normalized to C for the Shootout benchmarks.

Fig. 7. Time breakdown of Bioconductor vignettes.

Name | Input | Time Spent
--- | --- | ---
S-1 | Binary trees 16 | 0.0
S-2 | Fasta 2.5M | 0.1
S-3 | Fasta redux 2.5M | 0.2
S-4 | K-nucleotide 50K | 0.3
S-5 | N-body 500K | 0.4
S-6 | Mandelbrot 4K | 0.5
S-7 | Pidigits 500 | 0.6
S-8 | Regex-dna 2.5K | 0.7
S-9 | Rev. complement 5M | 0.8
S-10 | Spectral norm 640 | 0.9

Fig. 8. Allocate cons

Allocate cons

Fig. 8. Allocate vector

Allocate vector

Fig. 8. Garbage collection

Garbage collection

Fig. 8. Lookup

Lookup

Fig. 8. Duplicate

Duplicate

Fig. 8. Builtin

Builtin

Fig. 8. External

External

Fig. 8. Special

Special
Fig. 6. Slowdown of Python and R, normalized to C for the Shootout benchmarks.

Fig. 7. Time breakdown of Bioconductor vignettes.

To understand where the time is spent, we turn to more representative R programs. Fig. 7 shows the breakdown of execution times in the Bioconductor dataset obtained with ProfileR. Each bar represents a Bioconductor vignette. The key observation is that memory management accounts for an average of 29% of execution time. Memory management time was further broken down into time spent in garbage collection (18.7%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. The time spent in built-in functions represents the true computational work performed by R, this is on average 38% of execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and there is a program that spends 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position of variables in a frame are known. Given the nature of R, many of the core numerical functions are written in C or Fortran. This can lead to the perception that execution time is dominated by native libraries. Looking at the amount of time spent in calls to foreign libraries, 

Intel X5460, 3.16GHz, Linux 2.6.34, R 2.12.1, GCC v4.4.5
How is R used?

- Extract core semantics by testing
  - *R has no official semantics*
  - *A single reference implementation*

- Observational study based on a large corpus
  - *Many open source programs come with “vignettes”*
  - *Dynamic analysis gives under-approximated behaviors*
  - *Static analysis gives over-approximation*
A requirement of all packages in the Bioconductor repository is the inclusion of vignettes, i.e. programs that demonstrate real-world usage of these libraries. Out of the 630 Bioconductor programs, we focused on the 100 packages with the longest running vignettes. CRAN packages usually do not have vignettes, this is unfortunate as it makes them harder to analyze. We retained 1238 out of 3495 available CRAN packages.

The Shootout benchmarks were not available in R, so we implemented them to the best of our abilities. They provide tasks that are purely algorithmic, deterministic, and computationally focused. Further, they are designed to easily scale in either memory or computation. For a fair comparison, the Shootout benchmarks stick to the original algorithm. Two out of the 14 Shootout benchmarks were not used because they required multi-threading and one because it relied on highly-tuned low level libraries.

We restricted our implementations to standard R features. The only exception is the knucleotide problem, where environments were used as a substitute for hash maps.

## 6 Evaluating the R Implementation

Using ProfileR and TraceR, we get an overview of performance bottlenecks in the current implementation in terms of execution time and memory footprint. To give a relative sense of performance, each diagnostic starts with a comparison between R and C and Python using the shootout benchmarks. Beyond this, we used Bioconductor vignettes to understand the memory and time impacts in R’s typical usage.

All measurements were made on an 8 core Intel X5460 machine, running at 3.16GHz with the GNU/Linux 2.6.34.8-68 (x86_64) kernel. Version 2.12.1 of R compiled with GCC v4.4.5 was used as a baseline R, and as the base for our tools. The same compiler was used for compiling C programs, and finally a Python v2.6.4 was used. During benchmark comparisons and profiling executions, processes were attached to a single core where other processes were denied. Any other machine usage was prohibited.

### 6.1 Time

We used the Shootout benchmarks to compare the performance of C, Python and R. As shown by Fig. 6 the performance of R is slower than C by an average of 501 and Python by 43. Benchmarks where R performs better, like regex-dna (only 1.6 slower than C), are usually cases where R delegates most of its work to internal C functions.
Vectors

```r
x <- c(2, 7, 9, NA, 5)
c(1, 2, 3) + x[1:3]
x[is.na(x)] <- 0
```
Functions

\[ q \leftarrow \textbf{function} \ (x=5) \ x \times x \times x \]

\[ q() \]

\[ q(2) \]

\[ q(x=4) \]

\[ p \leftarrow \textbf{function} \ (x=5, \ldots, y=x+1) \]
Evaluating the Design of R

of the shootout problems are not easily expressed in R. We do not have any statistical analysis code written in Python and R, so a more meaningful comparison is difficult.

Fig. 11 shows the breakdown between code written in R and code in Fortran or C in 100 Bioconductor packages. On average, there is over twice as much R code. This is significant as package developers are surely savvy enough to write native code, and understand the performance penalty of R, yet they would still rather write code in R.

7.1 Functional

Side effects. Assignments can either define or update variables. In Bioconductor, 45% of them are definitions, and only two out of 217 million assignments are definitions in a parent frame by super assignment. In spite of the availability of non-local side effects (i.e., <<-), 99.9% of side effects are local. Assignments done through functions such as <<- need an existing data structure to operate on, thus they are always side effecting. Overall they account for 22% of all side effects and 12% of all assignments.

Fig. 12. Histogram of the number of function arguments in Bioconductor. (Log scale)
Promises

assert<-function(C,P)
  if (C) print(P)

assert( x==42, print(“Oops”))
Laziness. Lazy evaluation is a distinctive feature of R that has the potential for reducing unnecessary work performed by a computation. Our corpus, however, does not bear this out. Fig. 14(a) shows the rate of promise evaluation across all of our data sets. The average rate is 90%. Fig. 14(b) shows that on average 80% of promises are evaluated in the first function they are passed into. In computationally intensive benchmarks the rate of promise evaluation easily reaches 99%. In our own coding, whenever we encountered higher rates of unevaluated promises, finding where this occurred and refactoring the code to avoid those promises led to performance improvements.

Promises have a cost even when not evaluated. Their cost in in memory is the same as a pairlist cell, i.e., 56 bytes on a 64-bit architecture. On average, a program allocates 18GB for them, thus increasing pressure on the garbage collector. The time cost of promises is roughly one allocation, a handful of writes to memory. Moreover, it is a data type which has to be dispatched and tested to know if the content was already evaluated.
Forcing promises

\[ x \leftarrow F \]

\[ x[12] \leftarrow F \]

\[ F ; e \]

\[ \{ e ; F \} \]
Scoping

Lexical scoping with context sensitive name resolution

\[ \text{c} \gets 42 \]
\[ \text{c} \text{ (1, 2, 3)} \]
\[ \text{c} \gets 42 \]
\[ \text{d} \gets \text{c} \]
\[ \text{d} \text{ (1, 2, 3)} \]
less than 0.05% context sensitive function name lookups

only symbols that rely on it are c and file
Referential transparency

\[ \text{assert}(y[[1]] == 5) \]

\[ f(y) \]

\[ \text{assert}(y[[1]] == 5) \]

\[ f <- \text{function}(b) \{ b[[1]] <- 0 \} \]
Assignment

\[
x [ 42 ] <- y
\]

\[
\text{cpy}(H, \nu') = H', \nu'' \quad \Gamma = \iota \ast \Gamma' \quad \iota(H', x) = \nu'''
\]

\[
\text{readn}(\nu, H') = m \quad \text{set}(\nu''', m, \nu'', H') = H''
\]

\[
x[[\nu]] <- \nu' \quad \Gamma; H \to \nu'; H''
\]
Assignment

\[
\begin{align*}
y & \gets \text{c}(...) \\
f & \gets \text{function()} \\
    & \{ \\
    & \quad x \ [\ 42 \ ] \ \gets \ y
\end{align*}
\]
Assignment

\[
x[42] \leftarrow y
\]

\[
\text{cpy}(H, \nu) = H', \nu' \quad \Gamma = \nu * \Gamma' \quad \text{assign}(x, \nu', \Gamma', H') = H''
\]

\[
x \leftarrow \nu \Gamma; H \rightarrow \nu; H''
\]
45% of assignments are definitions

only 2 out of 217 million assignments are definitions in a parent frame

99.9% of side effects are local
who <- function(x) UseMethod("who")

who.man <- function(x) print("Ceasar!"))

who.default <- function(x) print("??")

me <- 42; who(me)
class(me) <- 'man'; who(me)
S3 method redefinitions

Fig. 15. Object usage in the corpus. In our corpus, 1,055 S3 classes, or roughly one fourth of all classes, have no methods defined on them and 1,107 classes, 30%, have only a `print` or `plot` method. Fig. 16 gives the number of redefinitions of S3 methods. Any number of definitions larger than one suggest some polymorphism. Unsurprisingly, `plot` and `print` dominate. While important, does the need for these two functions really justify an object system? Attributes already allow the programmer to tag values, and could easily be used to store closures for a handful of methods like `print` and `plot`. A prototype-based system would be simpler and probably more efficient than the S3 object system. Finally, only 30% of S3 classes are really object-oriented. This translates to one class for every two packages. This is quite low and makes rewriting them as S4 objects seem feasible. Doing so could simplify and improve both R code and the evaluator code.

S4 objects on the other hand, seem to be used in a more traditional way. The class hierarchies are not deep (maximum is 9), however they are not flat either. The number of parent classes is surprisingly low (see [5] for comparison), but reaches a maximum of 50 direct super-classes. In Fig. 15, singleton classes, i.e., classes which are both root and leaf, are ignored. At first glance, the number of method redefinitions seems to be a bit smaller than what we find in other object languages. This is partially explained by the absence of a root class, the use of class unions, and because multi-methods are declared outside of classes. The number of redefinitions, i.e., one method applied to a more specific class, is very low (only 1 in 25 classes). This pattern suggests that the S4 object model is mostly used to overcome an absence of structure declarations rather than to add objects in statistical computing. Even when biased by Bioconductor, which pushes for S4 adoption, the use of S4 classes remains low. Part of the reason may be the perception that S3 classes are less verbose and clumsy to write than S4; it may also come from the fact that the base libraries use S3 classes intensively and this is reflected in our data.
Objects

```r
setClass("P", representation(x="numeric",y="numeric"))
setClass("C", representation(color="character"))
setClass("CP", contains=c("P","C"))

r <- new("CP", x = 0, y = 0, color = "red")
r@color

setGeneric("add", function(a, b) standardGeneric("add"))
setMethod("add", signature("P","P"),
    function(a, b) new("P", x=a@x+b@x, y=a@y+b@y))
setMethod("add",signature("CP", "CP"),
    function(a, b) new("CP", x=a@x+b@x, y=a@y+b@y, color=a@color)
```
### Object usage

<table>
<thead>
<tr>
<th></th>
<th>Bioc</th>
<th>Misc</th>
<th>CRAN</th>
<th>Base</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># classes</td>
<td>1535</td>
<td>0</td>
<td>3351</td>
<td>191</td>
<td>3860</td>
</tr>
<tr>
<td># methods</td>
<td>1008</td>
<td>0</td>
<td>1924</td>
<td>289</td>
<td>2438</td>
</tr>
<tr>
<td>Avg. redef.</td>
<td>6.23</td>
<td>0</td>
<td>7.26</td>
<td>4.25</td>
<td>9.75</td>
</tr>
<tr>
<td>Method calls</td>
<td>13M</td>
<td>58M</td>
<td>-</td>
<td>-</td>
<td>76M</td>
</tr>
<tr>
<td>Super calls</td>
<td>697K</td>
<td>1.2M</td>
<td>-</td>
<td>-</td>
<td>2M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Bioc</th>
<th>Misc</th>
<th>CRAN</th>
<th>Base</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># classes</td>
<td>1915</td>
<td>2</td>
<td>1406</td>
<td>63</td>
<td>2893</td>
</tr>
<tr>
<td># singleton</td>
<td>608</td>
<td>2</td>
<td>370</td>
<td>28</td>
<td>884</td>
</tr>
<tr>
<td># leaves</td>
<td>819</td>
<td>0</td>
<td>621</td>
<td>16</td>
<td>1234</td>
</tr>
<tr>
<td>Hier. depth</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Direct supers</td>
<td>1.09</td>
<td>0</td>
<td>1.13</td>
<td>0.83</td>
<td>1.07</td>
</tr>
<tr>
<td># methods</td>
<td>4136</td>
<td>22</td>
<td>2151</td>
<td>24</td>
<td>5557</td>
</tr>
<tr>
<td>Avg. redef.</td>
<td>3</td>
<td>1</td>
<td>3.9</td>
<td>2.96</td>
<td>3.26</td>
</tr>
<tr>
<td>Redef. depth</td>
<td>1.12</td>
<td>1</td>
<td>1.21</td>
<td>1.08</td>
<td>1.14</td>
</tr>
<tr>
<td># new</td>
<td>668K</td>
<td>64</td>
<td>-</td>
<td>-</td>
<td>668K</td>
</tr>
<tr>
<td>Method calls</td>
<td>15M</td>
<td>266</td>
<td>-</td>
<td>-</td>
<td>15M</td>
</tr>
<tr>
<td>Super calls</td>
<td>94K</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>94K</td>
</tr>
</tbody>
</table>
The trouble with R?

• R is slow because it lacks a JIT

• R is a memory hog because it has large objects, allocates profusely and a non-moving garbage collector

• R has tricky semantics rife with cobwebs and dark corners

R is legacy software that must be maintained, yet it must also evolve to meet new challenges
R

in

Java
Why Java?

• Because JavaScript may not be fast enough
• Leverage a runtime system with zero-maintenance
• Get a good just-in-time compiler
• OpenJDK is an open source platform
Why not Java?

• The R community dislikes Java
• Interaction with C is cumbersome
• Complete break with GNU-R
FastR Architecture

R interpreter written in Java

47 KLOC

Up to 20x faster on shootout

GPL license

Built on OpenJDK, Graal, Truffle

ANTLR parser

Truffle Specializing Interpreter

FastR Runtime

Java Libraries

Hotspot

GRAAL JIT Compiler

Garbage Collector

Thread and locks

Native libraries (C, Fortran…)

Exec

OSR

JNI
FastR Throughput (w/o JIT)

Shoutout: speedup over GNU-R (gmean: 3.34x)

Benchmark25: speedup over GNU-R (gmean: 1.41x)

http://r.research.att.com/benchmarks/
Interpretation

\[ f(12, x+1, a=3) \]

\[
\begin{align*}
\text{gen\_call} & \quad f \\
\text{exp} & \quad \text{exp} \\
12 & \quad x+1 \\
3 & \quad 'a'
\end{align*}
\]

\[ f \leftarrow \text{function}(a,b,c)\{a+c\} \]
Specialization

\[
\text{f} \leftarrow \text{function}(a,b,c)\{a+c\}
\]
Runtime specialization

class If {
    RNode condE, trueB, falseB;

    Object execute(Frame f) {
        try {
            val = condE.executeScalarLogical(frame);
        } catch (UnexpectedResult e) {
            cast = ToLogical.mkNode(condE, e.result());
            replaceChild(condE, cast);
            return execute(frame);
        }
        if (val == TRUE) return trueB.execute(f);
        if (val == FALSE) return falseB.execute(f);
        throw unexpectedNA();
    }
}
Data types

- Data types are specialized to optimize memory usage and speed
- Scalar are unboxed when possible (soon)
- Memory footprint is improved
• …delay construction of large data objects
• …are a data-flow representation of vectors
• …avoid unnecessary work if a subset of the data is required
• …avoid allocation of temporary objects
• …permit fusion of multiple data traversals into one
Inlining

spectralnorm-naive

A <- function(i, j) {
  1 / ((i + j) * (i + j + 1) / 2 + i + 1)  
}

B <- function(u) {
  ...  
  for (j in 0:n1)
    ret[[i]] <- ret[[i]] + u[[j + 1]]  
    * A(i - 1, j)
}

- Inlining is a critical optimization in modern languages
- Replace a function call with it’s body
- Guarded inlining leaves a slow path in the code that performs the normal function
Delite is a compiler framework and runtime for parallel embedded DSLs.

Delite provides:

- Built-in parallel execution patterns
- Optimizers for parallel code
- Code generators for Scala, C++ and CUDA
- A heterogeneous runtime for executing DSLs

Relite is a proof-of-concept R interface to Delite.

https://github.com/TiarkRompf/Relite
```r
sapply(1:50000, function(x){sum(1:x)})
```

```
4.1s  GNU-R
1.5s  FastR
```

```r
Delite(sapply(1:50000,function(x) {sum(1:x)}))
```

```
0.4s  Relite
```

```r
sapply(1:50000,function(x) {sum((1:x)*0.1)})
```

```
9.0s  GNU-R
2.2s  FastR
0.5s  Relite
```

```r
sapply(1:50000, function(x){ sum(sapply(1:x, function(y) y*0.1))})
```

```
2395s  GNU-R
104s  FastR
0.5s  Relite
```
Conclusions

• R is an amazingly successful systems with great mindshare
• The R implementation is hard to maintain and evolve
• The FastR project aims to rethink how to implement R
• We leverage well tested technologies to build a high performance VM
• FastR can be a source of inspiration for GNU R