The R Bytecode Compiler and VM

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

July 11, 2019
R is a language for data analysis and graphics.

Originally developed by Ross Ihaka and Robert Gentleman, R is now maintained and developed by the R core group.

R is based on the S language developed by John Chambers and others at Bell Labs.

R is widely used in the field of statistics and beyond, especially in university environments.

R has become the primary framework for developing and making available new statistical methodology.

Many (over 13,000) extension packages are available through CRAN or similar repositories.
The standard R evaluation mechanism
- parses code into an *abstract syntax tree (AST)* when the code is read;
- evaluates code by interpreting the ASTs.

Compilation to some form of bytecode reduces interpreter overhead and allows for some other optimizations.
- Bytecode compilation is used in many languages, e.g. Python and Ruby.

The first release of the compiler occurred in R 2.13.0 (2011).

Significant improvements were released in R 3.2.0 (2015).

Just-in-time compilation was made the default in R 3.4.0 (2017).

Further improvements are in development.
The compiler can be called explicitly to compile single functions or files of code:

- `cmpfun` compiles a function;
- `cmpfile` compiles a file to be loaded by `loadcmp`.

It is also possible to have package code compiled when a package is installed; this is now the default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first or second use;
- loops are compiler before they are run.
The current compiler includes a number of optimizations, such as:
- constant folding;
- special instructions for most SPECIALs, many BUILTINs;
- inlining some simple .Internal calls;
- maintaining intermediate scaler results on the stack without boxing.

The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

The current VM design is stack-based; a register-based design may be adopted in the future.
A Simple Example

R Code

```r
f <- function(x) {
  s <- 0.0
  for (y in x)
    s <- s + y
  s
}
```

VM Assembly Code

```
LDCONST 0.0
SETVAR s
POP
GETVAR x
STARTFOR y L2
L1: GETVAR s
GETVAR y
ADD
SETVAR s
POP
L2: STEPFOR L1
ENDFOR
POPC
GETVAR s
RETURN
```
A Simple Example (cont.)

**R Code**

```r
f <- function(x) {
  s <- 0.0
  for (i in seq_along(x))
    s <- s + x[i]
  s
}
```

**VM Assembly Code**

```assembly
... GETVAR x
SEQALONG
STARTFOR.OP i L2
L1: GETVAR s
    GETVAR x
    STARTSUBSET_N <expr> L3
    GETVAR_MISSOK i
    VECSUBSET
L3: ADD
    ...
```

**Register-based loop body**

```assembly
...
L1: GETVAR s R1
    STARTSUBSET_N x <expr> L3
    VECSUBSET x i R2
L3: ADD R1 R2 s
    ...
```
Timings for some simple benchmarks on an x86_64 Ubuntu laptop:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AST</th>
<th>Comp.</th>
<th>Speedup</th>
<th>Exper.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>11.91</td>
<td>2.10</td>
<td>5.7</td>
<td>1.68</td>
<td>7.1</td>
</tr>
<tr>
<td>conv</td>
<td>9.07</td>
<td>1.31</td>
<td>6.9</td>
<td>0.85</td>
<td>10.7</td>
</tr>
<tr>
<td>ddot</td>
<td>34.59</td>
<td>5.75</td>
<td>6.0</td>
<td>4.10</td>
<td>8.4</td>
</tr>
<tr>
<td>rem</td>
<td>8.06</td>
<td>1.14</td>
<td>7.1</td>
<td>1.00</td>
<td>8.1</td>
</tr>
</tbody>
</table>

- **AST, Comp.** are for R 3.6.0
- **Exper.** includes use of unboxed variable bindings.
- Preliminary experiments:
  - a register-based design may provide another 2x speedup.
  - simple C code generation from either stack-based or register-based code may provide another 3-5x speedup.
Notes and Future Directions

- A major goal: minimize semantic changes.
  - Developing the compiler helped clarify some semantics.
  - Testing against all CRAN and BioConductor packages was also very helpful.
  - In the few cases where semantic differences remain, the compiled semantics are probably better.

- Compilation was a major motivation for adding namespaces to R, and for locking bindings in namespaces.
  - At default optimization level only calls to functions found through namespaces are optimized unconditionally.
  - In other cases, guard instructions are used to fall back to the AST interpreter.

- At this point only function bodies are compiled.
  - Default arguments will be interpreted.
  - Function calls use the (slow) interpreter mechanism.
  - This matches up well with (unfortunately) common one big function approach.
Some useful VM strategies:

- caching bindings from the innermost environment frame;
- using a typed stack to allow unboxed scalars;
- allowing unboxed scalar values in variable bindings;
- separate instructions for one and two index subscripting.

Other directions to explore:

- More efficient function calls.
- Reducing/avoiding lazy evaluation overhead when possible.
- Intra-procedural optimizations and inlining.
- Declarations (sealing, scalars, types, strictness).
- Machine code generation using LLVM or other approaches.
- Incorporating ideas from Justin Talbot, Renjin, and pqR on delaying/fusing computations.
- Trace compilation?
Notes and Future Directions

- **Debugging/profiling issues:**
  - Currently turning on debugging for a compiled function switches to the interpreted version.
  - There is some VM level profiling support but it could be a lot better.

- **Maintainability is a major concern**
  - The compiler is written in R as a literate program using noweb.
  - The VM is not nearly as well documented.
  - The VM uses threaded code when *gcc* is used (based on macros from Piumarta and Riccardi, 1998).
  - Generating machine code might complicate it further (or not).
  - The AST interpreter could be simplified to serve as a cleaner language specification.