Just-in-time Length Specialization of Dynamic Vector Code

Justin Talbot    Zachary DeVito    Pat Hanrahan

Tableau Research    Stanford University

(ARRAY 2014)
Tableau
Tableau + R
Riposte

• Bytecode interpreter and tracing JIT compiler for R

• Focused on
  • executing vector code well
  • using parallel hardware

• Written from scratch
  (how fast can it be? don’t reason from incremental changes!)

• http://github.com/jtalbot/riposte

• http://purl.stanford.edu/ym439jk6562
What makes R’s vectors hard?
They are semantically poor
How is it used?

• dynamically-allocated array?
• tuple?
• scalar?
• dictionary?
• tree?
What does it imply?

(If I know that a variable is a vector of length 4, what else can I figure out?)

• Usually very little!

• Recycling rule means that almost all vectors conform to each other
Riposte

• Project #1: Execute long vectors well (large dynamically-allocated arrays)

  • Deferred evaluation approach

  • Operator fusion/merging to eliminate memory bottlenecks

  • Parallelize execution of fused operators

• But…
Riposte

• Project #2: Execute short vectors well (scalars, tuples, short dynamically-allocated arrays)
  • Hot-loop just-in-time (JIT) compilation
  • (Partial) length specialization
  • Optimize based on lengths
Hot-loop JIT

- Hypothesis: if code has scalars or short vectors, computation time must be dominated by loops.

- Interpreter watches for expensive loops.

- When it finds one, compile machine code for loop, *make assumptions that lead to optimizations (specialization)*

- Guard against changes to assumptions
Hot-loop JIT

- Specialization
  - Assumptions should lead to *big optimization wins* (frequency * performance improvement)
  - Assumptions should be *predictable* (to amortize overhead)
Specialization

- *Type* specialization explored in other dynamic languages (Javascript, etc.)

- *Length* specialization is interesting in R
  - Eliminate recycling overhead
  - Store vector in register/stack instead of heap
  - Length-based optimizations (fusion, etc.)
Which length specializations make sense?

(big win + predictable)
Length specializations?

- Instrumented GNU R
- Recorded operand lengths of binary arithmetic operators
- Ran 200 vignettes, covering wide range of R application areas
Recycling rule?

- In 92% of calls, operands are the same length
  - Recycling overhead is frequently unnecessary
- Recycling is well predicted
  - Same lengths: 99.998%
  - Different lengths: 99.98%
- Specialized code has a high probability of being reused
Predictable lengths?
Predictable lengths?

![Graph showing predicted rate vs. vector length (binned on log2 scale)]
Predictable lengths?

![Graph showing the average prediction rate for different vector lengths. The x-axis represents vector length (binned on log_2 scale), and the y-axis represents the average prediction rate. The graph has a steep decline for vector lengths between $2^7$ and $2^8$.]

<8
Our strategy
Partial length specialization

1. Record loop using recycle instructions + abstract lengths

2. Eliminate *some* recycle instructions + introduce guards
   - Heuristic: Only specialize if the input lengths were equal while tracing and if both are loop carried or if both aren’t

3. Specialize *some* abstract lengths to concrete lengths + introduce guards
   - Heuristic: Only specialize vectors with non-loop carried lengths <= 4
Length-based optimizations

- Operator fusion
  (can’t have intervening recycle operations)

- Vector “register allocation”
  
  - SSE registers
    (needs concrete lengths)

  - Shared stack/heap locations / eliminate copies
    (needs same lengths)
Evaluation
Evaluation

• Can we run vectorized code efficiently across a wide range of vector lengths?

• 10 workloads, written in idiomatic R vectorized style so we can vary length of input vectors

• Compare to GNU R bytecode interpreter & C (clang 3.1 -O3 + autovectorization)

• Measure just execution time
<table>
<thead>
<tr>
<th>American Put</th>
<th>Binary Search</th>
<th>Black–Scholes</th>
<th>Column Sum</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandelbrot</td>
<td>Mean Shift</td>
<td>Random Walk</td>
<td>Riemann zeta</td>
<td>Runge–Kutta</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>normalized throughput (log scale)</th>
<th>vector length (log scale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 ×</td>
<td>$1$, $2^8$, $2^{16}$</td>
</tr>
<tr>
<td>1000 ×</td>
<td>$1$, $2^8$, $2^{16}$</td>
</tr>
<tr>
<td>100 ×</td>
<td>$1$, $2^8$, $2^{16}$</td>
</tr>
<tr>
<td>10 ×</td>
<td>$1$, $2^8$, $2^{16}$</td>
</tr>
<tr>
<td>1 ×</td>
<td>$1$, $2^8$, $2^{16}$</td>
</tr>
</tbody>
</table>

$R$
The diagram shows the normalized throughput for various algorithms and vector lengths. The x-axis represents the vector length on a log scale, while the y-axis represents the normalized throughput also on a log scale. Different algorithms are represented in different columns, and different methods (R, C) and specializations (No Specialization, Recycling) are shown in different colors.

- American Put
- Binary Search
- Black–Scholes
- Column Sum
- Fibonacci
- Mandelbrot
- Mean Shift
- Random Walk
- Riemann zeta
- Runge–Kutta

Each row represents a different algorithm, and each column represents a different vector length. The graphs show how the normalized throughput changes with vector length for each algorithm and method combination.
<table>
<thead>
<tr>
<th>American Put</th>
<th>Binary Search</th>
<th>Black–Scholes</th>
<th>Column Sum</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandelbrot</td>
<td>Mean Shift</td>
<td>Random Walk</td>
<td>Riemann zeta</td>
<td>Runge–Kutta</td>
</tr>
</tbody>
</table>

**normalized throughput (log scale)**

<table>
<thead>
<tr>
<th>vector length (log scale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

**Specialization**
- R
- C
- Recycling
- No Specialization
<table>
<thead>
<tr>
<th>American Put</th>
<th>Binary Search</th>
<th>Black–Scholes</th>
<th>Column Sum</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandelbrot</td>
<td>Mean Shift</td>
<td>Random Walk</td>
<td>Riemann zeta</td>
<td>Runge–Kutta</td>
</tr>
</tbody>
</table>

**Normalized Throughput (log scale)**

- **Vector Length (log scale)**
- **R**
- **C**
- **No Specialization**
- **Recycling**
- **Recycling+Short V**
How far did we get?
How far did we get?

- More stable performance across a wide-range of vector sizes, *but not yet as good as hand-written C on some workloads.*
- Performance on-par with C for some workloads, *but not all.*
  - Faster when we can make better use of SSE
  - Slower when there is scalar control flow
Open issues
Incomplete story

- Instrumentation showed our heuristics will not increase compilation overhead “much”

- Evaluation showed specialization with our heuristics increases performance across a wide range of vector lengths

- Missing: Real-world workloads running in Riposte to demonstrate that our approach works in the wild.
Long vs. short

- Unify long/short vector strategies in a single JIT?
  - Deferred vs hot loop execution?
  - Medium length vectors?
- What can we learn from nested parallel languages?
<table>
<thead>
<tr>
<th>Stage</th>
<th>Time (s)</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early optimizations</td>
<td>0.003</td>
<td>3.2%</td>
</tr>
<tr>
<td>Length specialization</td>
<td>≤ 0.001</td>
<td>~ 0.5%</td>
</tr>
<tr>
<td>Vector optimizations</td>
<td>≤ 0.001</td>
<td>~ 0.5%</td>
</tr>
<tr>
<td>Generating LLVM instructions</td>
<td>0.002</td>
<td>2.2%</td>
</tr>
<tr>
<td>LLVM optimization passes</td>
<td>0.012</td>
<td>13.0%</td>
</tr>
<tr>
<td>LLVM code emission</td>
<td>0.074</td>
<td>80.4%</td>
</tr>
</tbody>
</table>

Table 1. Compilation time for BLACK-SCHOLES.
Current State of Riposte
Towards Completeness

• Much harder than I originally thought…and I was originally pessimistic

• 700 Primitive & Internal functions
  • many not documented at all…what does `.addCondHands` do?
  • Riposte implements most of these in R (including S3 dispatch)
  • Riposte has ~80 primitive functions, most much lower level than R’s

• FFI
  • R header files (Rinternals.h, argh!) expose way too much of the internal implementation details
Vector FFIs?

.Map(ff_name, ...)  
Runtime handles recycling arguments and calls ff_name to get each result.

.Reduce(ff_name, base_case, ...)  
Runtime handles iteration
Vector FFIs?

- Runtime can do vector optimizations such as fusion
- Runtime can parallelize FFI execution
- Many built-in functions could be moved to libraries (e.g. transcendental functions)
Thanks