Just-in-time Length Specialization of Dynamic Vector Code

Justin Talbot
Tableau Software
jtalbot@tableausoftware.com

Zachary DeVito
Stanford University
zdevito@stanford.edu

Pat Hanrahan
Stanford University
hanrahan@cs.stanford.edu

Abstract

Dynamically typed vector languages are popular in data analytics and statistical computing. In these languages, vectors have both *dynamic type* and *dynamic length*, making static generation of efficient machine code difficult. In this paper, we describe a trace-based just-in-time compilation strategy that performs partial length specialization of dynamically typed vector code. This selective specialization is designed to avoid excessive compilation overhead while still enabling the generation of efficient machine code through length-based optimizations such as vector fusion, vector copy elimination, and the use of hardware SIMD units. We have implemented our approach in a virtual machine for a subset of R, a vector-based statistical computing language. In a variety of workloads, containing both scalar and vector code, we show near autovectorized C performance over a large range of vector sizes.

Categories and Subject Descriptors D.3.4 [*Processors*]: incremental compilers, code generation

General Terms Design, Experimentation, Performance

Keywords R, just-in-time compilation, tracing, data parallelism

1. Introduction

The rise of big data has increased the demand for domain specific languages designed for data analytics and statistical computing. The most popular of these—R [17], Matlab® [2], and NumPy (a library for Python) [9] —are dynamically typed, vector- and array-based languages. These languages are productive and user-friendly: dynamic typing supports their use in iterative, exploratory programming, and vectors and vectorized operations permit concise expression of common analytic tasks, such as tabular data processing and linear algebra. Unfortunately, efficient execution of these dynamic vector languages is not straightforward; R, for example, runs two orders of magnitude slower than well written C [14, 18].

In addition to the standard difficulties in dealing with *dynamic types*, compilers for vector languages have to deal with *dynamic lengths*. Without static length information, it is difficult to generate efficient machine code. In particular, short vectors—used in vector languages to represent scalars and small data structures—suffer from considerable runtime overhead. Their dynamic length implies that they cannot be allocated to hardware registers, and extra code, such as length checks, has to be generated to support them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from `Permissions@acm.org`.

ARRAY'14 June 11 2014, Edinburgh, United Kingdom
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2937-8/14/06 $15.00.
`http://dx.doi.org/10.1145/2627373.2627377`

To execute short vector code efficiently, this paper describes a trace-based just-in-time compilation approach that performs partial specialization of vector lengths. As in other trace-based systems, we record and compile frequently executed bytecode sequences ("traces"), but our traces contain both scalar and vector instructions, and lengths are recorded using a dependent type system. In an optimization pass, we selectively specialize the dependent lengths while attempting to balance increased performance due to specialization with the resulting increase in compilation overhead. This pass uses two heuristics—specializing very short vectors and specializing sets of vectors that are likely to share the same length. Each is motivated by an analysis showing that these properties are very stable in R programs. The specialized length information is then used to perform length-dependent optimizations: vector operations are fused, very short vectors are assigned to hardware vector units, and longer vectors are assigned to shared heap locations.

We have implemented this approach in a virtual machine which supports a subset of the R language. We evaluate our approach on a variety of R workloads and find that we can achieve performance near that of autovectorized C across a relatively wide range of dynamic vector lengths. This is one to two orders of magnitude faster than the open source R implementation.

2. Example

To better understand the problems that arise in executing short vector code in R, consider the example shown in Figure 1 that computes the option pricing for an American put. This model has no closed form solution, so iteration over `n` discrete time steps is used to approximate the result.

In R, the lengths of `americanPut`'s parameters are not statically known. This function can be called at runtime with scalars, in which case this function computes the price of a single option. Or called with vectors, in which case this function computes multiple option prices in parallel. Both uses are plausible and both may arise in a single analytic session. For example, the user may first call this function with medium-length vectors and a small `n` to roughly evaluate a number of possible options, and then may call the function again with short vectors and a large value of `n` to more accurately evaluate a few interesting positions. A further complication is that vector languages typically allow users to mix vectors of different lengths by applying automatic rules to make the vectors "conform" in binary or ternary operators. For example, R uses the *recycling rule* which automatically repeats shorter vector operands to the length of longer operands. Thus, the semantics of the language allow the lengths of `americanPut`'s parameters to vary arbitrarily and the run time must be able to handle all possible length combinations. While we could generate generic code that could handle all these combinations, it would necessarily be quite slow. Alternatively, we could statically generate specialized variants for all possible length combinations, but this would be prohibitive in both time and space.

```
americanPut <- function(S, K, up, p0, p1, n)
{
    # compute initial values
    t <- ...

    # iteratively propagate values
    for (j in n:1)
        for (i in 1:j)
            t[[i]] <- pmax(
                p0*t[[i]] + p1*t[[i+1]],
                K - S*up^(2*i-j) )
    t[[0]]
}
```

Figure 1. Vector R code for computing the value of an American put using the binomial option pricing model. If the parameters passed to this function are scalars, it evaluates the price of a single option. But if the parameters are vectors, the code computes the price of multiple options simultaneously.

We address these problems with a trace-based just-in-time compilation strategy combined with *partial* length specialization. Our goal is to specialize just enough to get high performance out of the resulting compiled code, but not so much so that compilation costs overcome the performance gains. The next two sections describe our tracing and partial specialization approaches.

3. Tracing Vector Code

Two principal approaches have been suggested for JIT compilation—function-based and trace-based. A primary question in the design of a new JIT is which approach to use. While trace-based approaches have recently been applied with some success to languages such as Lua [15], Python [5], and Javascript [7], recent developments in Javascript have shifted to function-based approaches [1]. Despite this, we use a trace-based approach, primarily because tracing, combined with guard elimination, creates straight-line code which is ideal for vector fusion. Additionally, there are a couple of reasons to believe that tracing may be more effective on vector languages than on scalar languages. First, tracing typically struggles with executing nested loops well. But vector operations eliminate the innermost loop of computatation, generally reducing the nesting depth of code. Second, while, tracing has a hard time with control flow divergence, most vector languages include vector blend or select operations which can eliminate some control flow in favor of conditional moves or predicated instructions.

3.1 Vector Traces

Our virtual machine begins executing the code from Listing 1 in its interpreter. When a loop becomes hot, we begin recording the executed instructions into a trace. This is done by patching the dispatch table to first record the instruction in the trace before executing it. Traces are recorded in a typed low-level IR (LIR) in Static Single Assignment (SSA) form [6]. In many vector languages, including R, vectors have value semantics, so our LIR also represents vectors as first-class values. The LIR instruction set includes vector-style operations (e.g. maps, reductions, gathers, scatters, and broadcasts) along with more typical instructions such as loads and stores from environments, boxing and unboxing of interpreter values, and guards. Since our trace does not have any branching control flow, ϕ nodes are only inserted at the end of the trace to represent loop carried dependencies.

Listing 2 shows a snippet of the trace of the inner loop in `americanPut` corresponding to the expression `K - S*up^(2*i-j)`. Some bytecodes in our interpreter translate directly to a single instruction in the LIR, but others are expanded into multiple instructions to conform to the SSA format or to permit later optimizations. For example, loading a vector variable, such as `K` or `S` in Listing 2, is split across three instructions. This allows us to represent the vector length (`v11`) and the pointer to the unboxed vector (`v12`) as separate nodes in the SSA form. Similarly, when recording binary and ternary operators we emit additional `recycle` instructions that make R's vector conformance rules explicit (e.g. lines `v29`–`v31`). Our length equality specialization optimizes some of these away (Section 4.1).

```
# load K
v10 :=   1 -> value         load      'K'
v11 :=   1 -> [int x   1]   length    v10
v12 :=   1 -> [dou x v11]   unbox     v10   ->>

# load S
v13 :=   1 -> value         load      'S'
v14 :=   1 -> [int x   1]   length    v13
v15 :=   1 -> [dou x v13]   unbox     v13   ->>

# compute up^(2*i-j)
...
v24 :=   1 -> [int x   1]   length(up^(2*i-j))
v25 := v24 -> [dou x v24]   value(up^(2*i-j))

# compute K - S * up^(2*i - j)
v29 :=   1 -> [int x   1]   max       v14   v24
v30 := v29 -> [dou x v29]   recycle   v15
v31 := v29 -> [dou x v29]   recycle   v25
v32 := v29 -> [dou x v29]   mul       v30   v31
v33 :=   1 -> [int x   1]   max       v11   v29
v34 := v33 -> [dou x v33]   recycle   v12
v35 := v33 -> [dou x v33]   recycle   v32
v36 := v33 -> [dou x v33]   sub       v34   v35
```

Figure 2. Simplified snippet from the trace of the inner loop of `americanPut` corresponding to the expression `K - S*up^(2*i-j)`.

As in other type-specializing JITs, every instruction in the LIR trace is strongly typed. Except for the special `value` type, which represents boxed values from the interpreter, our LIR types take the form: `<ilength> -> [<eltype> x <length>]`, where `<ilength>` is the instruction's iteration count and `[<eltype> x <length>]` gives the element type and length of the resulting vector. For example, in the snippet above, the type of value `v11`: `1 -> [int x 1]` means that `length` is a scalar instruction (iteration count = 1) and it returns a scalar integer.

The length information in the types can be dependent on previous instructions. The type of `v12`: `1 -> [dou x v11]` means that this `unbox` is a scalar instruction that returns a double vector whose length is the result of instruction `v11`. Using dependent lengths allows us to avoid specializing lengths in the trace while recording. Our constant length specialization pass uses information from the entire trace to selectively replace dependent lengths with concrete values (Section 4.2).

To allow dependent lengths on instructions that might change the length of a vector, our LIR includes a paired instruction that computes the new length. For example, since R allows assignment beyond the end of the vector (handled by the LIR `scatter` instruction), the LIR also includes a `scatter_length` instruction which computes the length of the resulting vector as the maximum of the old length and the new indices.

If the trace successfully completes a loop, the trace is handed off to our JIT compiler which performs length specialization, optimization, and then code generation.

3.2 Early Optimizations

While tracing, and immediately afterward, we perform a small set of early optimizations to make the length specialization pass described in Section 4 more effective. Key among these are constant propagation (to propagate constant length information), redundant load elimination, and common subexpression elimination.

To support dynamic creation and deletion of local variables (a feature of languages, like R, that unify function environments and hash tables), we model local variable loads and stores explicitly in the trace. If these opaque memory accesses remain in the trace, they will block later optimizations, including the length specialization pass. To address this, we use a language-specific alias analysis pass to eliminate most redundant loads from the trace.

After recording the trace we peel the first iteration of the loop and perform a common subexpression elimination (CSE) pass. Inspired by LuaJIT and PyPy [3], this combination of optimizations allows us to handle loop invariant code motion as a side effect of CSE. By eliminating redundant code, CSE can prove that two dependent lengths are necessarily equal, which improves the results of the subsequent length specialization pass.

4. Partial Length Specialization

Given the unspecialized trace, the goal of the partial length specialization pass is to to get good performance without overspecializing, which could lead to poor trace reuse and, thus, high recompilation costs.

To understand which specializations would be profitable, we instrumented the open source version of R and gathered vector length statistics for 200 vignettes—sample programs which come with many R packages to demonstrate their functionality—drawn from a broad sample of packages posted to CRAN (http://cran.r-project.org). While not real analysis workloads, the vignettes exercise a wide range of R's use cases including simulation, model fitting, plotting, mapping, etc. Using the abstract syntax tree (AST)-walking interpreter in the open source distribution of R, we gathered statistics on the operand lengths for R's built-in binary math operators. This data led us to use two partial specialization strategies described in the following sections.

4.1 Length equality specialization

As noted earlier, since R supports mixing operands of different lengths, for each binary or ternary vector instruction we have to insert code to handle the "recycling rule" which extends the shorter to the length of the longer. But, in practice, arbitrary mixtures of vector lengths are not that common. In our instrumented vignettes we found that the arguments to binary arithmetic operators were the same length 92% of the time. Further, when an AST node was observed to have operands of equal length, it had equal length operands on the next invocation 99.998% of the time (and conversely, when an AST node was observed to have operands of different lengths, it also had unequal lengths on the next invocation 99.98% of the time). Thus, despite R's support for mixing operands of different lengths, in practice this is extremely rare. Thus, the goal of our length equality specialization pass is to eliminate `recycle` instructions that are unlikely to be necessary.

We first iterate over the `recycle` instructions and consider whether to specialize the input and output lengths to be equal, using the following criteria:

1. The lengths must have been equal while recording the trace.
2. The lengths must both be loop variant or both loop invariant.
3. The lengths must not be constant.

After selecting pairs to specialize and using union-find to build equality sets, our second pass propagates this information through the trace, converting `recycle` instructions to guards. Many of these guards will be redundant and can be eliminated during this or a later optimization pass.

```
# load K
v10 :=   1 -> value         load      'K'
v11 :=   1 -> [int x   1]   length    v10
v12 :=   1 -> [dou x v11]   unbox     v10   ->>

# load S
v13 :=   1 -> value         load      'S'
v15 :=   1 -> [dou x v11]   unbox     v13   ->>

# compute up^(2*i-j)
...
v25 := v11 -> [dou x v11]   value(up^(2*i-j))

# compute K - S * up^(2*i - j)
v32 := v11 -> [dou x v11]   mul       v15   v25
v36 := v11 -> [dou x v11]   sub       v12   v32
```

Figure 3. After specialization under the assumption that all the parameters to `americanPut` are the same length, the code is greatly simplified and opportunities for vector fusion are exposed (e.g. `v32` & `v36`). The generated machine code will be much faster, while the need for recompilation remains reasonable since the trace can be reused if vector lengths change systematically.

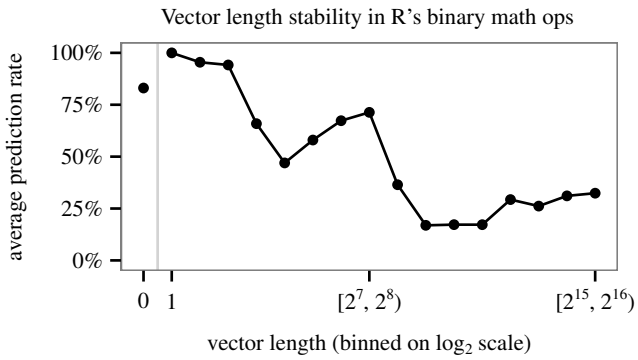

Figure 4. Vector length stability for the operands to binary math operators in R expressed as the percentage of AST node invocations in which the operand length is that same as the previous invocation of the the same node. Lengths have been binned into log_2-sized bins. Scalars and very short vectors are stable in math operators, but stability drops off quickly as lengths increase above 8.

For example, if the input vectors to the snippet shown in Listing 2 are all the same length, then the equality specialization pass and redundant guard elimination will reduce it to the snippet shown in Listing 3. The now unnecessary `recycle` instructions have been eliminated and all vector operations now execute on vectors of the same length, `v11`.

4.2 Constant length specialization

Length equality specialization preserves the ability to reuse the generated machine code if vector lengths change in a structured way. But since the loop lengths remain generic, the generated machine code will include loops over the vectors' dynamic lengths and vectors will have to be stored in dynamically allocated memory rather than in registers. To avoid this overhead, we would like to selectively replace the generic lengths in the trace with constants so we can emit more efficient code. However, such specialization

will cause us to recompile if lengths change in later invocations. We used our instrumented vignettes to better understand how well the operands lengths in one invocation of an AST node predicted operand lengths in later invocations. The results, shown in Figure 4, demonstrate that lengths less than $2^3 = 8$ are quite stable.

Additionally, we expect that this specialization will only provide substantial performance improvements for short vectors where the loop overhead cannot be amortized well. Our performance experiments (Section 7.2) suggest that, on SSE hardware, specializing beyond vectors of length 4 does not provide much performance improvement. Thus, based on both vector length stability and observed performance improvements we only specialize vectors with $1 \leq \text{length} \leq 4$.

4.3 Blacklisting length specializations

Our length specialization choices try to minimize recompilation, but in some cases may still be overly aggressive. For example, if there are multiple input vectors whose lengths vary independently, the total number of traces grows exponentially in the number of inputs. This scenario is unlikely in typical R code; but to avoid degenerate behavior in such scenarios, we track the number of side traces generated in a trace tree that result from length guard failures. Once this reaches a threshold (8) we blacklist the loop and stop compiling additional length variations.

5. Length-based Optimizations

After length specialization, our JIT performs a number of additional optimizations on the trace including constant folding, strength reduction, CSE, and code sinking to move code from the main trace into side exits (inspired by LuaJIT [16]). These optimizations are applied to both scalar and vector operations. We also apply two optimizations that are dependent on the vector length—a vector fusion pass, and a vector register assignment pass.

5.1 Vector Fusion

Given operations with the same `<ilength>`, we can perform *vector fusion*—a transformation similar to loop fusion but starting from vector instructions rather than scalar loops. For short vectors, fusion eliminates loop overhead. For longer vectors, where instruction intermediates exceed the cache size, fusion replaces expensive cache misses with reads from a register. Our fusion pass simply forms fused blocks out of adjacent instructions in the trace that have the same `<ilength>`. We also perform a liveness analysis pass to avoid materializing values not used outside of their fused block. During code generation (Section 6), each fused block is turned into a single loop over the `<ilength>`.

5.2 Vector Register Allocation

A scalar JIT has to perform register allocation to assign virtual registers to the physical registers of the machine. In our vector VM, vector intermediate values may not fit into machine registers, so we must place them in memory instead. While we could place each vector intermediate in its own memory location, this would increase the workload size, make poor use of caches, and introduce completely unnecessary vector copies in ϕ nodes and in, e.g., `scatter` instructions. Instead, we do a linear scan register allocation pass on the SSA trace. Constant length vectors are allocated to stack locations which may be promoted to machine registers during code generation. All other vectors are allocated in heap memory. Vectors which have the same type (including length) and whose live ranges do not overlap can be allocated to the same heap location. These memory locations are managed by the compiled trace. They are allocated when first used, and freed or passed back to the interpreter when the trace exits.

Stage	Time (s)	Percent
Early optimizations	0.003	3.2%
Length specialization	≤ 0.001	$\sim 0.5\%$
Vector optimizations	≤ 0.001	$\sim 0.5\%$
Generating LLVM instructions	0.002	2.2%
LLVM optimization passes	0.012	13.0%
LLVM code emission	0.074	80.4%

Table 1. Compilation time for BLACK-SCHOLES.

If vector lengths are not loop invariant or are used in an operation that may resize a vector (e.g. `scatter`), the size of the heap allocation may need to change during execution of the trace. To handle this, we insert length checking code into ϕ nodes and other length-changing instructions to check the current allocated length and to grow it if necessary. To maintain a constant insertion cost, each resize grows the allocation to the next power of 2.

6. Code Generation

Our code generation pass takes the optimized trace and emits the high-level control flow—the peeled first iteration followed by a loop. We then populate the body of this structure one fusion block at a time. For each fusion block, we emit straight-line vectorized code, if `<ilength>` is constant, or a blocked loop over `<ilength>` if it's a dependent length.

For each side exit we generate code to perform any sunk instructions and then generate code to reconstruct the interpreter state. We then emit a tail call to a stub function that simply returns control to the interpreter. Once the side exit is traced, we replace the stub with a jump to the compiled side trace.

Our JIT currently uses LLVM [12] to generate machine code. There is considerable overhead in using LLVM; its optimization and code emission passes take tens of milliseconds for a few hundred LLVM bytecodes (see Table 1). However, using LLVM has freed us from the need to implement an assembler and we rely on a few of its optimization passes (e.g. `mem2reg`) to simplify our code generation phase.

7. Evaluation

Since we want to demonstrate that we can run a wide range of vector lengths well, including very short vectors, we created a set of ten parameterized workloads in which we can vary the input vector lengths while keeping the total computation constant. We selected a range of benchmarks covering standard scalar iterative workloads, standard data parallel workloads, as well as workloads containing both iterative and data parallel components. Following good practices for vector languages, we wrote these benchmarks in a vectorized style, preferring vector operations such as select to explicit control-flow where possible. Our benchmark implementations are available at: http://github.com/jtalbot/riposte/.

Since LLVM is slow, we run the programs long enough to amortize this JIT compilation cost. In a production JIT, we would need to replace LLVM's code emission phase with a more performant assembler. The system we used for evaluation is a 4-core Intel Core i7 with 8GB of RAM, but all the benchmarks use a single core.

7.1 Overall Performance

The throughput of our runtime on the ten workloads is shown in Figure 5. Along the x-axis of each plot we vary the length of the input vectors from 1 (scalars) to 2^{16}. We compare against R and C. To get the R results we run the same vector workloads in the 64-bit version of R 2.15.1, enabling R's new stack-based interpreter. The

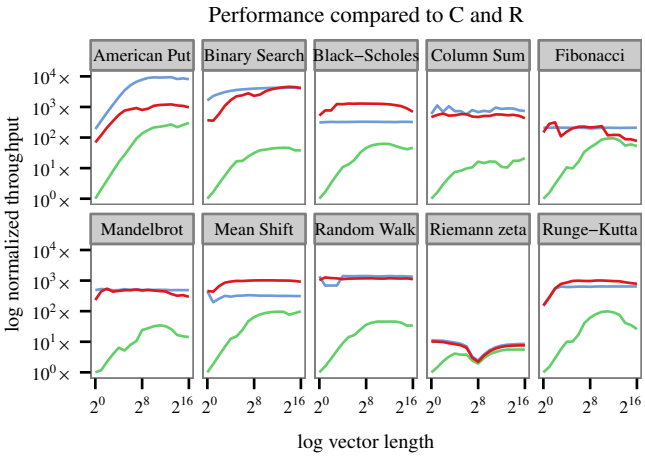

Figure 5. Throughput of our runtime (red), optimized & autovectorized C (blue), and the open source implementation of R (green) as a function of the vector length (on log-log scale). Our performance is near that of C across a wide range of vector sizes.

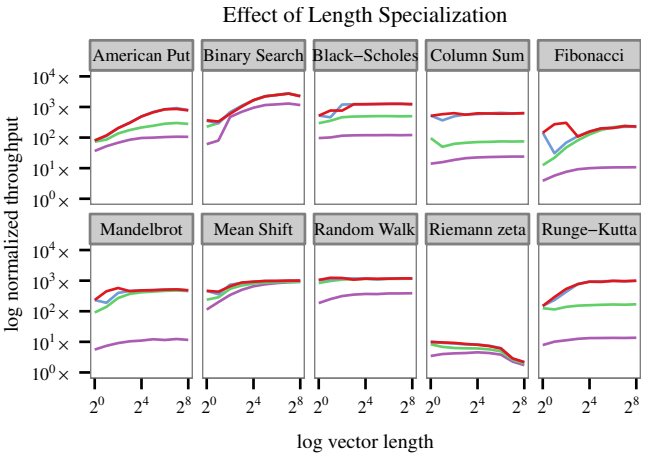

Figure 6. Throughput of our runtime with different length specialization combinations. No specialization (purple) has high overhead and thus low performance. Equality specialization (green) and specialization of scalars (blue) have large performance impacts. Specializing vectors with length ≤ 4 (red) leads to more modest performance improvements.

C code is scalar code produced by hand fusing the workloads. We compiled it using Clang version 3.1 with `-O3` and autovectorization flags on. We recompiled the C code for each vector length (by defining the vector length as a preprocessor symbol) giving the compiler the opportunity to make length optimizations.

Our JIT does not yet support R's missing value feature, so our performance and R's performance are not directly comparable. However, we expect the impact of adding missing value support to our JIT to be relatively small, since it primarily impacts integer and logical operations which are a relatively small percentage of these workloads. Our C implementations also do not implement R's missing value feature.

In Figure 5, first note that R performs very poorly on very short vectors due to considerable interpreter overhead, despite its new stack-based interpreter. Performance improves as the vector lengths increase since the interpreter overhead is better amortized. However, R performance never matches C performance, leveling off when it becomes bound by the materialization of intermediate vector values.

In contrast, our runtime performs well across the entire range of input vector lengths. In most cases our performance is slightly less than C; but in three cases (BLACK-SCHOLES, MEAN SHIFT, and RUNGE-KUTTA) our performance is actually better due to better utilization of the hardware vector units. While Clang's autovectorization failed in these cases, our vector-based compilation strategy was able to take advantage of SIMD units. In the case of FIBONACCI, and to a lesser extent MANDELBROT, performance degrades for large vector sizes due to a missed fusion opportunity since we only fuse adjacent vector operations. The RIEMANN ZETA function computes an exponent which dominates the run time making the performance difference between R and C relatively small. The dip in the middle of the performance curve is related to our call to `pow` to compute the exponent, but the exact reason is not clear.

7.2 Effect of Length Specialization

Next we explore how length specialization affects performance. Figure 6 compares four specialization configurations—no length specialization (purple), only equality specialization (green), and equality specialization plus constant length specialization of just scalars (blue) or of all vectors with length ≤ 4 (red). For most of our benchmarks, the majority of the performance gain comes from equality specialization which permits us to eliminate recycling cost and perform fusion, reducing loop overhead for short vectors and reducing the working set for longer vectors.

Specializing scalars results in large performance improvements in BLACK-SCHOLES, COLUMN SUM, and RUNGE-KUTTA due to replacing a generic `recycle` instruction with a cheaper `broadcast`. Specializing scalars results in minor performance improvements for short vectors in MANDELBROT and MEAN SHIFT due to loop overhead elimination.

Specializing longer vectors (up to 4) results in some performance improvement in workloads with tight inner loops—`Fibonacci` and `Mandelbrot`. The effect on other workloads is minor. Specializing even longer vectors (not shown) results in performance degradations on some workloads due to the resulting instruction mix in the generated code. On SSE hardware, specializing vectors of length ≤ 4 is a good trade off point between improved performance on some workloads and decreased opportunities for code reuse.

7.3 Effect of Register Allocation

Figure 7 shows the effect of the vector register allocation pass. In all workloads vector fusion eliminates most of the vector intermediates so only a small number of intermediates actually have to be allocated on the heap.

The optimization is most effective for the AMERICAN PUT workload which includes vector updates. Without register allocation, each vector update requires a copy of the entire vector to maintain R's value semantics (the well-known aggregate update problem). The register allocation pass discovers that this copy can be safely eliminated. This turns the $O(n)$ vector update into a $O(1)$ in-place update. The decreasing benefit of this optimization for bigger vector sizes is an artifact of our workload design. To keep the total number of operations the same, we decrease n in AMERICAN PUT as we increase the vector length.

In the cases of BLACK-SCHOLES, COLUMN SUM, and MANDELBROT the register allocation pass places both sides of a vector ϕ node into the same heap location eliminating a vector copy, resulting in a roughly constant performance boost. Additionally this decreases the working set size, delaying the performance hit due to exceeding the cache size, visible in the COLUMN SUM and MANDELBROT curves for large vector sizes.

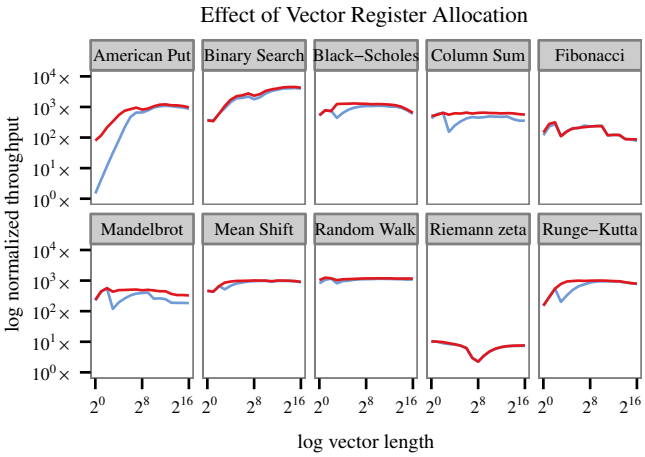

Figure 7. Effect of our register allocation pass (without–blue, with–red). We get large performance improvements in the case of AMERICAN PUT where register allocation eliminates vector copies resulting from the aggregate update problem. In BINARY SEARCH, COLUMN SUM, and MANDELBROT register allocation eliminates a vector copy in a ϕ node resulting in a smaller speed up.

8. Related Work

Recent work on trace-based JIT compilation has focused on optimizing dynamic scalar languages such as Javascript [7], Python [5], and Lua [15]. This paper adapts these techniques to vector code.

Our equality specialization pass is similar to the "shape cliques" technique developed for APL [4], but we use a combination of runtime length information and data-flow analysis to dynamically discover vector structure. Dependent types like ours have also been used for static type checking of array languages [20]. Joisha and Banerjee propose a graph coloring-based approach to combine heap allocations in Matlab programs [8], and Lameed and Hendren present a staged static analysis pass to eliminate array copies in Matlab [11]. Our vector register allocation scheme solves the same problem, but is simpler due to our straight line trace.

Kalibera et al. describe an improved AST interpreter for R [10]. Tierney implements a new bytecode-based interpreter for R [19] and Wang et al. explore techniques for accelerating it [21]. Li et al. perform incremental runtime analysis to parallelize loops in R programs [13]. Morandat et al. have studied the semantics and performance of the current open source R implementation [14].

9. Discussion

We've described a trace-based approach for efficiently executing dynamic vector languages across a wide range of vector sizes from scalars to tens of thousands of elements. It is based on a set of partial length specialization passes that enable length-based optimizations, such as vector fusion and vector register allocation. The resulting virtual machine performs one to two orders of magnitude faster than standard R, comparable to optimized C. As future work, our evaluation needs to be extended to show that our approach can maintain low overhead on real world workloads. This will require supporting more of the R language, which represents a significant technical challenge.

This paper has demonstrated that dynamically typed vector languages can achieve high performance and can effectively utilize data parallel hardware through *just-in-time length specialization and compilation*. For users of such languages, such as data analysts, this means increased productivity and scalability in the languages they prefer to use.

References

[1] Google V8 Javascript engine. `http://code.google.com/p/v8/`.

[2] Matlab. `http://www.mathworks.com/products/matlab/`.

[3] H. Ardö, C. F. Bolz, and M. Fijałkowski. Loop-aware optimizations in PyPy's tracing JIT. In *Proceedings of the 8th symposium on Dynamic languages*, DLS, pages 63–72, New York, NY, USA, 2012. ACM.

[4] R. Bernecky. Shape cliques. *SIGAPL APL Quote Quad*, 35(3):7–17, Sept. 2007.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS, pages 18–25, New York, NY, USA, 2009. ACM.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4), Oct. 1991.

[7] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 465–478, New York, NY, USA, 2009. ACM.

[8] P. G. Joisha and P. Banerjee. Static array storage optimization in MATLAB. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI, pages 258–268, New York, NY, USA, 2003. ACM.

[9] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[10] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A fast abstract syntax tree interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE, pages 89–102, 2014. ISBN 978-1-4503-2764-0.

[11] N. Lameed and L. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 22–41. Springer Berlin / Heidelberg, 2011.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International symposium on Code Generation and Optimization*, CGO, Palo Alto, California, Mar 2004.

[13] J. Li, X. Ma, S. Yoginath, G. Kora, and N. F. Samatova. Transparent runtime parallelization of the R scripting language. *J. Parallel Distrib. Comput.*, 71(2):157–168, Feb. 2011.

[14] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *ECOOP 2012 Object-Oriented Programming*, Lecture Notes in Computer Science, 2012.

[15] M. Pall. The LuaJIT project. `http://http://luajit.org/`, .

[16] M. Pall. LuaJIT Allocation Sinking Optimization. `http://wiki.luajit.org/Allocation-Sinking-Optimization`, .

[17] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011.

[18] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: A trace-driven compiler and parallel VM for vector code in R. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT, pages 43–52, 2012.

[19] L. Tierney. A byte code compiler for R. Technical report, 2012.

[20] K. Trojahner. *QUBE–Array Programming with Dependent Types*. PhD thesis, University of Lübeck, Lübeck, Germany, 2011.

[21] H. Wang, P. Wu, and D. Padua. Optimizing R VM: Allocation removal and path length reduction via interpreter-level specialization. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2014.