

Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R

Justin Talbot
Stanford University
jtalbot@stanford.edu

Zachary DeVito
Stanford University
zdevito@stanford.edu

Pat Hanrahan
Stanford University
hanrahan@cs.stanford.edu

ABSTRACT

There is a growing utilization gap between modern hardware and modern programming languages for data analysis. Due to power and other constraints, recent processor design has sought improved performance through increased SIMD and multi-core parallelism. At the same time, high-level, dynamically typed languages for data analysis have become popular. These languages emphasize ease of use and high productivity, but have, in general, low performance and limited support for exploiting hardware parallelism.

In this paper, we describe Riposte, a new runtime for the R language, which bridges this gap. Riposte uses *tracing*, a technique commonly used to accelerate scalar code, to dynamically discover and extract sequences of vector operations from arbitrary R code. Once extracted, we can fuse traces to eliminate unnecessary memory traffic, compile them to use hardware SIMD units, and schedule them to run across multiple cores, allowing us to fully utilize the available parallelism on modern shared-memory machines. Our evaluation shows that Riposte can run vector R code near the speed of hand-optimized C, 5–50x faster than the open source implementation of R, and can also linearly scale to 32 cores for some tasks. Across 12 different workloads we achieve an overall average speed-up of over 150x without explicit programmer parallelization.

Categories and Subject Descriptors

D.3.4 [Processors]: incremental compilers, code generation, interpreters

Keywords

R language, just-in-time compilation, tracing, data parallel

1. INTRODUCTION

Recent trends in hardware have increased available parallelism by widening SIMD units, increasing core counts, and adding specialized hardware accelerators, such as GPUs. These trends challenge traditional programming models, which focus on scalar code, leading to interest in developing more effective parallel programming models.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
Copyright 2012 ACM 978-1-4503-1182-3/12/09 ...\$15.00.

We approach this challenge from the domain of data analysis where there is increasing interest in programmer-friendly, high productivity languages, such as R, Matlab, and Python. These languages are now widely used in statistics, data mining, and machine learning, as well as in application areas such as social sciences, biology, and finance. Analytic languages commonly include arbitrary-length homogeneous arrays (“vectors”) as basic types in addition to standard scalar types. Vector types and their associated operations are a good fit to data analysis workloads which often contain linear algebra-style arithmetic and substantial processing of tabular data sets. More importantly, these vector types and operations provide regions of implicit data parallelism that should allow users to transparently exploit emerging parallel hardware.

However, the dynamically typed semantics of these languages make it difficult to extract this data parallelism. Necessary type checks and other dynamic control flow lead to virtual machine implementations that execute a single vector operation at a time, resulting in large intermediate values that consume memory and bandwidth. In such implementations, vector operations are memory-bound and fail to take advantage of data parallel hardware.

Work on addressing this problem has focused on two approaches: (1) supplementing or replacing dynamically typed languages with statically-typed ones that are easier to analyze and compile [9, 23, 25], or (2) exposing parallelism by adding coarse-grained *explicit* task parallel constructs to the language [37, 29]. Both of these approaches give up productivity and familiarity. The second approach additionally sacrifices fine-grained parallelism, including the use of SIMD units and GPUs.

A more promising direction is suggested by the recent work on accelerating languages such as Javascript [1, 13] and Lua [2]. This work maintains the original semantics of the dynamic language, but achieves good performance through runtime type specialization and just-in-time compilation. However, this work has focused primarily on scalar code, making little use of parallel hardware.

In this paper, we tackle the challenge of efficiently executing a dynamically-typed data analytic language with vector types. Our solution uses a *trace-based* approach to dynamically separate the complex scalar code, including dynamic type checks and other difficult semantics, from the simpler vector operations. Once separated, we can aggressively optimize the vector operations for modern parallel hardware—performing vector fusion to eliminate memory traffic, JIT compiling to SIMD machine code, and scheduling across multiple cores on a shared memory machine.

We have implemented our tracing approach in Riposte, a prototype virtual machine for the R programming language. For code using R’s vector operations, Riposte achieves a 5–50x speed up over the open source R interpreter [27] on a single core and demonstrates near linear scalability for some workloads out to 32 cores.

This paper makes the following contributions:

- We demonstrate that a *vector-based, dynamically typed* language can be an effective and efficient parallel programming model for data analysis tasks.
- We present a tracing approach inspired by deferred evaluation that can extract simple, compilable vector sequences from dynamic R code. This is augmented with a dynamic liveness analysis pass that can eliminate unnecessary intermediate outputs recorded in the trace, decreasing our memory and bandwidth usage.
- We describe a simple vector virtual machine designed to support the straight-line vector code produced by tracing and show that the semantics of the virtual machine make vector fusion easy. We also describe our VM’s simple just-in-time compiler and its parallel runtime for shared-memory machines.
- We validate our approach by comparing Riposte to the existing open-source R interpreter, showing large speed-ups, and to hand-written, vectorized, C code, showing performance within a small factor of optimal. We also show that Riposte can scale well to 32 cores.

In Section 2, we give a high-level overview of our design through a worked example. Then in Section 3, we describe our tracing approach in more detail. Section 4 explains how we perform vector fusion on the traces. In Section 5 we explain how we optimize, JIT, and execute the fused code. We evaluate Riposte in Section 6 and then end with related work and discussion.

2. DESIGN OVERVIEW

We provide an overview of our system by describing how our trace-based approach will execute the simple program in Figure 1 that computes the average income of males over 40 from a dataset. This example code, while simple, uses some high-level dynamic features of R. The `incomes_of` function takes an arbitrary first-class filtering function as a parameter; the function `males_over_40` is generic and can operate on entire vectors; and the semantics of R dictate that function arguments be lazily evaluated. As Riposte executes, it separates this dynamic language functionality from the vector operations, so that the latter can be executed efficiently.

Riposte interprets the example normally until it reaches line 6. At this point the interpreter must evaluate the subexpression `a >= 40`. Since `a` is the vector `age`, R’s semantics require that this operation actually perform 20 million greater-than or equal comparisons. We could execute this immediately (this is what the open source R interpreter does), but that commits us to materializing the 20 million result values, consuming substantial memory and bandwidth. Instead, the Riposte interpreter checks the length of the operands dynamically; if `a` is a long vector, we delay the `>=` instruction by inserting it into a *vector trace* (lines 1–3 of Figure 2) and inserting a *future* into the interpreter’s result slot. Continuing execution, the operators `==` and `&` (line 6) are similarly recorded and delayed. The interpreter then returns a future from the function `males_over_40` instead of an actual value. In addition to simple operations, like `>=`, which operate element-wise over the array, we also delay and record instruction, like filters (`[]`, line 11) and reductions (`mean`, line 14), which change the shape of the vector.

Tracing ends when the interpreter encounters `print`. The final vector trace, shown in Figure 2, is handed to the JIT to be compiled. Note that unlike the interpreter’s bytecodes, the instructions in the

```

1  age <- read.table("ages") #20 million elements
2  gender <- read.table("genders")
3  income <- read.table("incomes")
4
5  males_over_40 <- function(a,g) {
6    a >= 40 & g == 1
7  }
8
9  incomes_of <- function(filter_function) {
10   mask <- filter_function(age,gender)
11   income[mask]
12 }
13
14 print(mean(incomes_of(males_over_40)))

```

Figure 1: R code that computes the mean income for a subset of a sample, a simple data analysis task. Dynamic types and control flow (such as the dynamic function call used in `incomes_of`) make static analysis of this R code difficult. Tracing permits us to recover blocks of vector code as a first step to compilation.

```

n0 : double[1]      = constant 40
n1 : double[20M]   = load age
n2 : logical[20M]  = ge  n1 n0
n3 : double[1]     = constant 1
n4 : double[20M]   = load gender
n5 : logical[20M]  = eq  n4 n3
n6 : logical[20M]  = and n2 n5
n7 : double[20M]   = load income
n8 : logical[20M[n6]] = filter n7 n6
n9 : double[1]     = mean n8

```

Figure 2: Vector trace generated from the example code in Figure 1. Dynamic types and control flow have been eliminated. Each instruction has both a concrete type (e.g. `double`) and shape (e.g. `20M` elements).

vector trace contain both the type (e.g. `double`), and shape of arrays (e.g. 20 million elements), making it possible to generate efficient code.

At this point, multiple futures exist in the interpreter’s state, but some of these futures have become unreachable. The results associated with them do not have to be written out, saving memory and bandwidth. For instance, the variable `mask` holds a future referencing the output of value `n6` in the trace. But `mask` is no longer reachable because the `incomes_of` function has returned, so the output of `n6` does not have to be materialized.

After eliminating dead futures, we fuse together the vector trace and JIT it to hardware vector instructions (e.g. SSE), shown as pseudocode in Figure 3. The compiled code is inserted into a task queue where it is collaboratively completed by worker threads through task stealing.

Finally, the result replaces the future in the argument to `print` and the interpreter resumes computation, printing out the average income of males over 40.

We can compare Riposte’s trace-based execution of this program to the standard behavior of R’s interpreter. R’s interpreter will execute each vector operation completely before continuing, materializing intermediates that are as large as the original arrays. In this case, intermediates are created to hold the temporaries for the results of the `>=`, `==`, and `&` operators. Further, R implements the filter

```

double2 sum = {0,0};
for(int i = 0; i < 20000000; i += 2) {
  bool2 a = age[i] >= 40;
  bool2 b = gender[i] == 1;
  bool2 c = a & b;
  double2 new_sum = sum + income[i];
  sum = blend(c,new_sum,sum);
}
double mean = (sum[0] + sum[1]) / 20000000;

```

Figure 3: Pseudocode of the generated machine code from the trace in Figure 2. It is vectorized and can be executed in parallel by multiple threads. (blend is the vectorized ternary operator ?:)

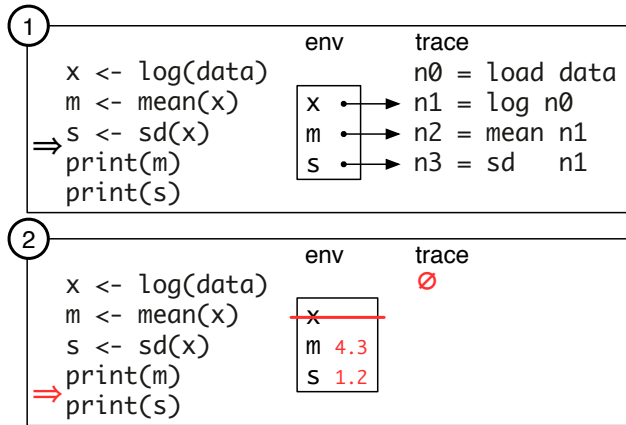


Figure 4: The state of the interpreter (1) before and (2) after executing a trace. The statement print(m) requires the value for m. The value for s will be used later, so it should also be materialized to avoid recomputation. The value x is dead, so it should not be materialized.

operator (`[]`) by first creating an array of indexes where the filter is true, and then using it to gather the list of valid incomes, creating two additional temporaries. Assuming the worst case (where the filter removes no values), the R version will need to read 9 arrays of 20 million elements, and allocate/write 5 arrays. In contrast, Riposte will only need to read the 3 input arrays. The result is that the Riposte trace-based JIT runs this example 6.5 times faster than standard R.

3. EXTRACTING VECTOR TRACES FROM DYNAMIC CODE

The following sections describe our vector tracing approach in more detail. First, we explain the deferred evaluation tracing strategy. Then we describe our liveness analysis which determines the set of trace outputs that should be materialized.

3.1 Vector Tracing via Deferred Evaluation

Tracing is typically done on scalar code [14]. In this case, one observes the execution of an early iteration of a loop, and then generates type-specialized machine code to run the remainder of the iterations. Scalar operators execute relatively quickly, so interpreting the first few iterations of the loop is feasible. In contrast, the cost of vector operations is dependent on the size of the vector. For large vectors, each individual vector operation is itself an expensive

loop. If we waited until the second time we saw such vector operations before optimizing their execution, we would miss a large number of critical loops.

This leads us to use a tracing approach based on deferred evaluation. Rather than executing vector instructions immediately, we delay their execution through the creation of futures that reference entries in the vector trace. This allows us to record and compile vector operators the first time they are seen. Figure 4 (1) illustrates this process. After executing the first three instructions, the values of `x`, `m`, and `s` are futures pointing into the trace rather than concrete values. Since recording and compilation have a cost, we only delay evaluation if the lengths of the operands are long enough to amortize the overhead introduced. As we discuss in Section 5.2, this currently occurs at 512 elements.

After creating futures, the interpreter can propagate them through later instructions. Vector operations that have futures as operands can be recorded and new futures are created for their output (as with `mean` and `sd` in Figure 4). Futures used in many other operations can be propagated through unchanged. For example, futures can be assigned to variables in environments and passed to and returned from functions. Since the trace entry referenced by the future is typed, futures can also be used in dynamic dispatch without being evaluated. Vector operations in loops will be recorded multiple times as interpreter execution unrolls the loop.

Modifying the interpreter to support tracing may add overhead. A naive approach would introduce a future check and a length check into each vector operation in the interpreter. We found that this approach can add 20% or more to the runtime of tight scalar loops. We eliminate the overhead of the future check by including futures as a built-in interpreter type and folding the check for futures into the already existing type check in each operation. We avoid the cost of the length check by placing it after special-case dispatch logic for scalar operands. Since we treat the futures as a type in the interpreter, futures can also be adapted to work with more sophisticated approaches based on inline caching [8].

The infrastructure to store and execute traced instructions requires additional memory allocations and bookkeeping. To minimize this cost, we limit the total number of traced instructions. If a trace grows too large, we simply execute it even if a future has not been requested. This allows us to use fixed-size buffers to store the trace and all compilation intermediates.

3.2 Selective Materialization through Dynamic Liveness Analysis

Our vector tracing approach produces blocks of vector code that we can easily fuse and compile, but it adds an additional complexity. Each trace can contain multiple outputs in the form of unevaluated futures in the interpreter. These outputs may be (1) required by the interpreter to continue execution, (2) required at a later point, or (3) dead (i.e. the future represents an intermediate value that is never used again). Consider Figure 4 (1). The value `m` is required by the interpreter in order to execute the next `print` statement, `s` will be needed in the next line, and `x` is never used again. There are two problems that arise in handling these outputs.

First, for outputs in group (2) there is a classic time vs. space trade off. Since they are not yet needed by the interpreter, we could delay materializing them at the cost of potentially executing the trace multiple times. Or we could materialize them the first time the trace runs at the cost of increased storage. We use the latter approach since it will never materialize more outputs than the standard R execution strategy. Thus, in Figure 4 (2), after evaluating the first `print` statement, the futures for both `m` and `s` have been materialized. In contrast, recomputing values can perform considerably

worse than the current R interpreter in certain cases. In Figure 4, had we not evaluated `s` at the same time as `m`, we would have had to evaluate `x` twice, duplicating the expensive `log` operation.

Second, if we were to materialize dead outputs, we would generate unnecessary memory traffic. For example, in Figure 4, the value of `x` is never used again after the `print` statement, therefore there is no need to materialize it at all. If we fail to detect this, we will allocate and write to an unnecessary result vector.

Since static analysis of R code is difficult, we use a dynamic liveness analysis pass that determines the set of futures that are reachable from the interpreter’s state. Unreachable futures are dead and their outputs can be safely eliminated. This pass is conceptually very similar to garbage collection. The most straightforward approach is to give each future a reference count, but we found that this introduces an unacceptable cost on non-vector code since each stack-stack or stack-environment move must check whether the moved value is a future before updating its reference count.

Instead, we use a strategy similar to a tracing garbage collector. While recording instructions we maintain a conservative root set containing environments and stack frames that may contain futures. This can be done entirely off the fast path since this set only changes when recording an instruction in the vector trace, along function call boundaries, and in some infrequently-used environment manipulation operations. Then, before executing the trace, we traverse the root set looking for futures. Any futures not found are known to be dead and their outputs from the trace are not materialized.

4. FUSING VECTOR TRACES

Vector fusion—combining the individual vector operations into a single loop—is important in getting high performance. In fused code, intermediate values can be stored in registers. In contrast, unfused operators store intermediates in memory. For long vectors, where the intermediates exceed the cache size, fusion will replace expensive cache misses with reads from a register.

Similar to the Stream Fusion approach [11], Riposte considers only the subset of potential fusion opportunities that will result in a single fused loop; we never generate nested loops. We have designed our vector virtual machine to make this fusion easy to derive from the vector’s shape.

We first describe the design of the vector virtual machine and then show how we can use the vector shapes to group operations into fused loops.

4.1 Vector Virtual Machine

Our vector virtual machine supports immutable 1D vectors of uniform type (currently `double`, 64-bit `integer`, and `logical`, each of which also include R’s special missing value `NA`). Each vector has a shape consisting of an integer length, and optionally, a *filter* defined by a logical vector and/or a *group by* defined by an integer vector.

It supports three classes of vector operations:

1. generators, e.g. `gather` (read from an already materialized vector) or `seq` (generate an arithmetic sequence),
2. unary, binary, and ternary operations mapped over vector(s), e.g. `sqrt`, `add`, or `blend`, and
3. reductions, e.g. `sum`, `min`, or `scatter` (write a new vector out to memory).

Two special operators, `filter` and `groupby` semantically change the shape of their input, but do no actual computation.

Map operators are defined to apply element-wise to all vectors regardless of shape. Thus, on grouped vectors, map operators apply to each element of each group, not to the groups themselves. In contrast, the behavior of reductions depends upon their input shape. If the shape has a *filter*, filtered elements are not included in the reduction. If the shape has a *group by*, a grouped reduction is computed. (A grouped `scatter` produces a list of vectors.)

4.2 Shape-based Fusion

The semantics of the vector virtual machine make it easy to determine that operators on vectors of the same length can always be fused. Generators, maps, and reductions on simple vectors of the same length are trivially fuseable since they have the same implicit loop iteration count and the loops are implicitly aligned. Operations on filtered vectors can be fused with operations on the unfiltered length since this translates to a simple conditional on the filter inside the fused loop. Similarly, operations on grouped vectors can also be fused with operations on the ungrouped length. This is true since maps apply element-wise regardless of the grouping and reductions can handle the grouping by generating additional code inside the loop to scatter the output to the correct location for the group.

Binary and ternary map operations can have operands with different lengths in R. R semantically handles this case by repeating the shorter operand(s) to the length of the longest one. While it might be profitable in some cases to fuse such operations, in general, repeating the shorter vector in a fused fashion can cause unnecessary repeated computation. Instead, we handle this case by always materializing the shorter operand(s) and then fusing with the longest operand. Note that handling scalar-vector operations is a special case of this heuristic—the scalar operand is materialized by the interpreter and fusion continues on the vector.

Given this heuristic, we can implement fusion by grouping vector operations by the length of their largest input. Since vector length is known as we record instructions, we implement this grouping by recording multiple length-specific traces. If we encounter an instruction with operands of different lengths, we immediately execute the trace associated with the shorter operand and then delay and record the instruction into the longer operand’s trace.

5. COMPILING AND PARALLELIZING VECTOR TRACES

Riposte’s just-in-time compiler transforms vector traces into executable code. Creating a compiler is typically very involved. However, an advantage of a trace-based approach is that vector traces have no control flow, simplifying the compilation process. In the following two sections, we describe our simple vector JIT. Then in the final section, we discuss scheduling the compiled traces on multiple cores.

5.1 Optimization

Since vector traces contain no control flow, standard optimizations such as constant propagation, algebraic simplification, common subexpression elimination, and dead code elimination can be implemented as single passes over the vector trace.

In addition to improving user-written expressions, these passes also improve less-than-optimal code generated by our tracing approach. For example, R vectors use 1-based indexing, but our vector language uses 0-based indexing for easier compilation. This results in traces littered with code to add and subtract 1. The algebraic simplification pass eliminates most of these. Similarly, our tracing based approach unrolls loops, which can generate many replicates

of the body of the loop. Common subexpression elimination removes copies of loop invariant code.

5.2 JIT

Code generation for straight-line vector traces is simple enough that we emit x86-64 machine code directly. Our fusion approach guarantees that each trace will only contain operands of a single length, and we only have to generate code for a single loop over the vector length. We can generate the body of this loop by replacing the each variable-length vector operation in the trace with a fixed-length hardware equivalent. For the map operations this is often a single SSE instruction. But for operations not supported by the hardware instruction set, we either inline short instruction sequences or emit calls to vectorized functions. Generators are also relatively straightforward, though they must update loop-carried dependencies.

The code generation for reductions is more complicated due to filters and groupbys. A standard way to handle filters in vector code is to use predicated instructions that are disabled when the filter is false. However, x86-64 has poor support for predicated instructions. Instead, we execute all map operations regardless of the filter state, and only predicate the reductions, via an SSE blend operation or a branch. When the cardinality of a reduction’s groupby is low, or there is no groupby, we maintain multiple aggregates, one per vector lane per group, allowing the reductions to be vectorized. For large cardinality reductions, this increase in the working set size can negatively impact performance; so, in these cases, we serialize the reductions and only store a single aggregate per group.

Register allocation is accomplished using the linear scan algorithm [26]. For operations whose output needs to be stored into a live future, we generate store instructions as needed. Additionally, we JIT a loop header that initializes loop-carried state, such as reduction variables or sequence iterators, in the thread-local stack space.

As a further simplification, we do not attempt to reuse vector traces. This means that the JIT does not have to generate code to guard dynamic control flow decisions or to restore the interpreter to the correct state when a guard fails. For sufficiently long vectors, the overhead of compilation is still low because it is amortized over the vector length. For the current implementation, the break-even point occurs at 512 elements. We see only incremental benefits in optimizing for vectors shorter than this. Such vectors will fit in the L1 cache, so they benefit less from fusion; they may be better addressed using standard scalar compilation techniques.

As an alternative to JIT compiling vector traces, we considered strip-mining [7, 12, 35]. However, in a preliminary experiment, we found that strip-mining performed 33% worse than JIT compilation on our BLACK-SCHOLES benchmark. Similarly, other work has reported that strip-mining performs 2 times slower than hand-written compiled code [7].

5.3 Scheduling to Multicore

The compiled code consists of a single fully fused loop which is nearly data-parallel. To partition the iterations of this loop across multiple threads there are two cases we have to handle specially: generators and reductions. Vector generation functions (e.g. seq) may have loop-carried dependencies. When we partition the loop iterations we have to initialize the loop-carried dependencies to their proper state. This is straightforward for the set of generators we support. We only support reductions that are commutative and associative, making them easy to parallelize. Each thread allocates separate space in different cache lines to store partial reductions so that the threads operate independently and avoid false sharing. Af-

Table 1: Workloads

BLACK-SCHOLES	Compute 10M options
DATA CLEANING	On 20M element vector: filter invalid values, compute z-scores, and count outliers
FILTER 1-D	7-element 1-D convolution applied to a 10M element vector
HISTOGRAM	100M element vector with 100 distinct values
K-MEANS	1M 2D data points, 5 means
LOGISTIC REGRESSION	Compute via gradient descent on 30 continuous predictors with 1M data points
MANDELBROT	2048 × 1538 resolution in the domain $(-2, -1) \times (1, 1)$
COVARIANCE MATRIX	50 variables with 1M data points
RAY-SPHERE INTERSECTION	Distance to the closest intersection of a ray with 10M spheres
SAMPLING MIXTURE OF NORMALS	Draw 10M samples from mixture of 3 normals using CDF inversion
SPARSE MATRIX-VECTOR MULTIPLICATION	500K × 500K matrix with 20M non-zero entries
TPC-H QUERY 1	Database query with filter and aggregations (Scale Factor 10 = ~60M rows)

ter the computation has finished, the separate partial reductions are then aggregated together.

We don’t yet support the output of filtered or grouped vectors without an aggregate when executing in multiple threads. Doing so correctly requires a parallel prefix scan which is not yet included in our vector language.

The fused loop is scheduled on multiple cores using Lazy Binary Splitting [33], a recent work stealing-based scheduling strategy that attempts to dynamically discover an appropriate task block size. In our experience, this has been quite robust.

6. EVALUATION

In this section we evaluate our tracing-based approach. First, we demonstrate that our trace-based vector fusion and liveness analysis can eliminate a substantial fraction of memory accesses by comparing Riposte with the open source implementation of R, which does no fusion. We show that eliminating these memory accesses achieves a substantial speed-up. Next, we compare Riposte to vectorized C code executing the same benchmarks to show that we can effectively utilize vector hardware and that vector traces can achieve very high absolute performance. Finally, we examine the scalability of our trace scheduling and show that leveraging the implicit parallelism in R vector code results in a substantial performance improvement.

We use a set of twelve workloads, shown in Table 1, covering a range of problem domains. Riposte and the open source R implementation execute nearly the same high-level R code for each workload. (In a couple cases, we had to write non-idiomatic R in order to work around severe performance limitations of its blend operation (ifelse). And in one case, we worked around the fact that

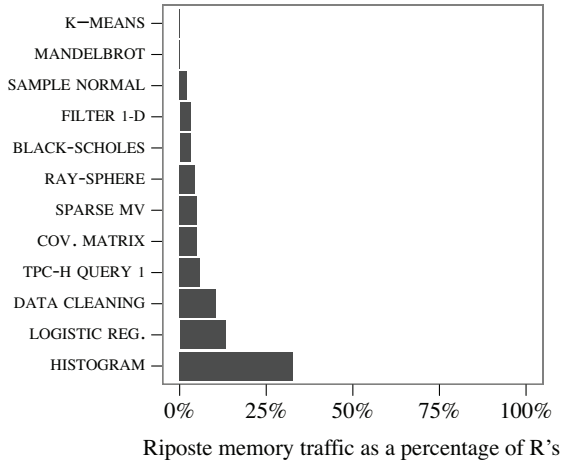


Figure 5: Vector fusion substantially reduces Riposte’s main memory traffic (total number of requests to main memory, measured using hardware counters) compared to R.

Riposte does not yet store futures in objects.) Since we are evaluating the use of vector primitives, we restricted the set of workloads to ones that can be written in vectorized R code. Thus, none of these benchmarks are bottlenecked on interpreter performance. Additionally, to clearly evaluate the benefits of runtime fusion of vector code, we also avoided calling R standard library functions which implement hand-fused sequences in C code.

To make the comparisons as fair as possible, we compiled R 2.14.2, Riposte, and our C implementations of the workloads using ICC 12.0.0. We used the flags `-O3` and `-fp-model fast` to enable ICC’s floating-point optimizations and loop autovectorization. ICC failed to autovectorize half of our C implementations. We hand vectorized these cases using SSE intrinsics or the Intel SPMD Program Compiler (ISPC) [24]. We also spot-checked the generated assembly of R’s vector math operations to ensure that they were correctly autovectorized. The system we used for evaluation is a shared memory machine with 4 8-core Nehalem-EX X7560 processors running at 2.2Ghz and 128GB of RAM. All measurements of speed-up are reported using the minimum execution time of 3 runs of the reference code and the maximum execution time of 3 runs of Riposte.

6.1 Effectiveness of Fusion

The goal of vector fusion is to reduce memory traffic by eliminating intermediates and by rearranging data access patterns to be more cache friendly. Figure 5 shows the main memory traffic of Riposte as a fraction of the main memory traffic of R for all our benchmarks, measured using the `offcore_response_0` counter on Intel Nehalem chips. Fusion succeeds in eliminating a substantial portion of the memory traffic. Both K-MEANS and MANDELBROT are iterative workloads and R has very high memory traffic as it repeatedly cycles the working set in and out of cache. In Riposte, fusion succeeds in reducing the working set below the L3 cache size. Thus, Riposte only touches main memory once, on the first iteration, for a substantially memory traffic reduction. In contrast, the HISTOGRAM workload is a single pass. Riposte is able to eliminate one intermediate (1 store + 1 read), but still must do the initial read, resulting in a comparatively high 33% traffic rate.

Figure 6 shows the speed of Riposte normalized to R. Speed-ups range from 5x to more than 50x. The harmonic mean speed up

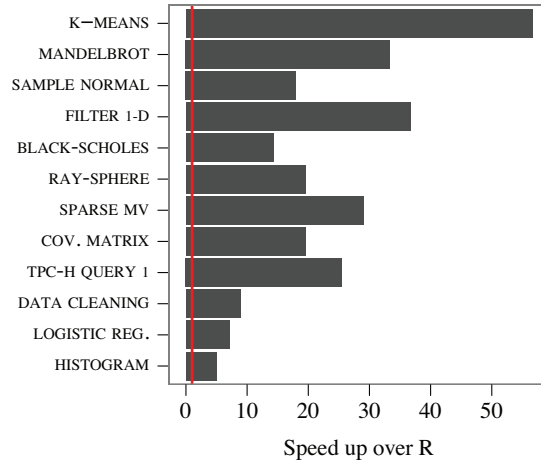


Figure 6: Performance of Riposte (including all interpreter and JIT overhead) normalized to R. The red line is at 1x (no speed-up). Riposte outperforms R on all benchmarks.

across all 12 workloads is 14.4x. The speed-up is roughly a function of the percent of memory accesses eliminated by fusion (note the inverse relationship between Figures 5 and 6). However, other factors such as the memory access pattern resulting from fusion and the task-specific instruction mix also affect the speed up.

For example, for workloads such as FILTER 1-D, K-MEANS, and MANDELBROT, which have a large number of simple math operators and a small set of input vectors, Riposte achieves a 35–50x speed-up. When R runs these workloads, it is extremely memory bound and there is lots of room for improvement. For other workloads that have a much larger input set, like COVARIANCE MATRIX and LOGISTIC REGRESSION, the improvements are smaller (5–20x) since the fraction of memory traffic that fusion can eliminate is lower.

6.2 Absolute Performance

We next compare Riposte to vectorized C implementations of the same tasks to evaluate absolute performance. Figure 7 shows the performance of Riposte normalized to vectorized C. The harmonic mean speed-up across all workloads is 0.74x. Despite the simplicity of our JIT compiler, Riposte performs almost as well as well-written vectorized C code, which indicates that Riposte is getting most of the speed-up possible over R.

The two slowest workloads, MANDELBROT and SAMPLE NORMAL, suffer from our lack of support for true conditional operations. We currently always generate branches as blend operations which require evaluating both sides of the branch. This is costly in these workloads because the amount of work performed in the conditional is not trivial. The lower performance of the TPC-H workload is caused by missed opportunities for common sub-expression elimination because we don’t decompose grouped reduction instructions.

To better understand the pay off of vectorization, we also compared Riposte to unvectorized C implementations (Figure 8). Since Riposte primarily operates on doubles and 64-bit integers, on SSE the maximum speed up possible is 2x. We achieve a more moderate 10%–50% improvement on some workloads compared to C. As vector lengths increase, we expect the payoff of vectorization to grow.

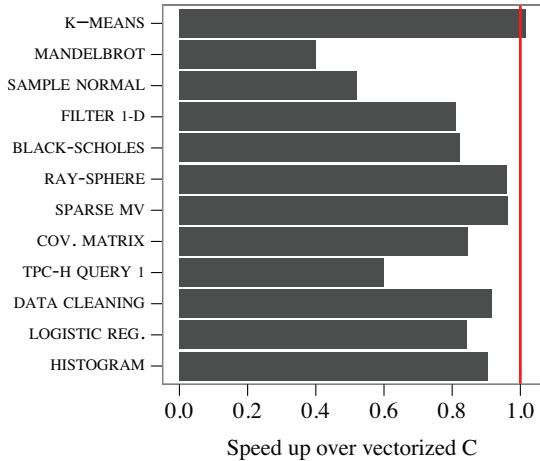


Figure 7: Performance of Riposte (including all interpreter and JIT overhead) normalized to vectorized C implementations. On average, Riposte performs within a small factor of C.

6.3 Scalability

Finally, we examine the scalability of Riposte, shown in Figure 9. Most workloads demonstrate reasonable scalability out to 16 or 32 cores; some even benefit from hyper-threading.

The operating system limits scaling in DATA CLEANING and FILTER 1-D. Both read in a large vector and output a vector of similar size. Creating the new vector requires allocating fresh pages—both examples spend 30–50% of their time allocating pages. Scaling is limited by the performance of the page allocator in Linux, which has been shown to only scale to 8 cores [34].

SPARSE MATRIX-VECTOR MULTIPLICATION scales very poorly due to our simple parallel reduction strategy. We allocate separate output vectors for each thread which are merged at the end. As the number of threads grows, the amount of memory allocated for the reduction exceeds the L3 cache and performance degrades. High-cardinality reductions are a well-known challenge in parallel runtime design [36].

Riposte does not always find the maximum amount of fusion possible. For instance, the loops in COVARIANCE MATRIX are too long to unroll completely meaning that Riposte must execute the loops in multiple traces. While this introduces only minor overhead into single-threaded code, it creates a synchronization point between each trace, which limits scaling. In the standard R implementation, parallelizing individual operations yields only limited improvements because of this problem [31].

Excluding the outlying SPARSE MATRIX-VECTOR MULTIPLICATION benchmark, Riposte’s overall speed up on 32 threads compared to R’s scalar performance ranges from 55–670x with a harmonic mean of 155x.

7. RELATED WORK

Previous work has applied trace-based just-in-time compilation to optimize scalar code. The Dynamo optimization system traced native machine code to perform runtime optimizations [6]. Gal et al. used tracing to optimize Java bytecodes [14]. Tracing has also been used to accelerate dynamic languages such as Javascript [1, 13] or Lua [2]. Riposte adapts these techniques to vector code.

Delaying array operations in a dynamic language in order to expose opportunities for later optimization was a feature of the APL

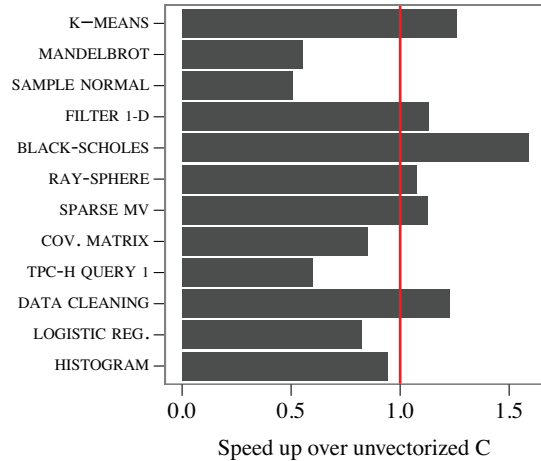


Figure 8: Performance of Riposte normalized to unvectorized C. Riposte’s support for 2-wide SSE operations provides moderate performance gains on some, but not all, workloads.

Machine [4, 19, 16], a proposed (but never built) physical machine with an APL-like instruction set. Riposte demonstrates that a similar approach, designed for modern hardware, can be effectively implemented in software.

Other work has applied vector optimizations in the context of statically compiled languages. Intel’s Array Building Blocks [21] and the PeakStream platform [22] add support for array-based programming to C++ through a JIT compiler. Data-parallel Haskell, a compiled functional language, implements nested array operations using a vector machine that supports segmented operations [23]. Coutts et al. use Haskell’s algebraic data-types and user-defined rewrite rules to implement static fusion of a number of “streamable” operations [11]. Keller et al. apply loop fusion on a functional representation of delayed arrays written in Haskell, and transparently parallelize the code [17].

Morandat et al. have studied the semantics and performance of the current open source R implementation [20]. Schmidberger et al. provide a good analysis of existing parallel libraries for R [29]. There are a number of other projects that have tried to improve the implementation of the R language. Tierney has developed a new bytecode interpreter for R that is progressively replacing the existing AST-walking interpreter in the standard distribution [32]. It improves the performance of scalar code. Tierney has also explored hand-fusion and parallelization of R code [31]. Milborrow has developed an experimental just-in-time compiler for R that can accelerate scalar arithmetic loops [3]. Garvin has developed an R-to-C cross compiler that also focuses on scalar performance. The CXXR project is refactoring the open source R implementation to ease future development, but it has not yet focused on performance improvements [28]. While this paper was in submission, NVIDIA released an experimental R-to-PTX compiler based on an LLVM backend [15]. Like Riposte, it does aggressive vector fusion, but is currently limited to compiling simple sequences of map operations.

The growth of big data analysis problems has created interest in making other higher-level data analytic languages more scalable. McVM [10] is a function-based JIT for Matlab. It has been used to explore static [18] and profile-based [5] optimization techniques to Matlab. Copperhead [9] compiles a statically analyzable subset of Python to GPUs.

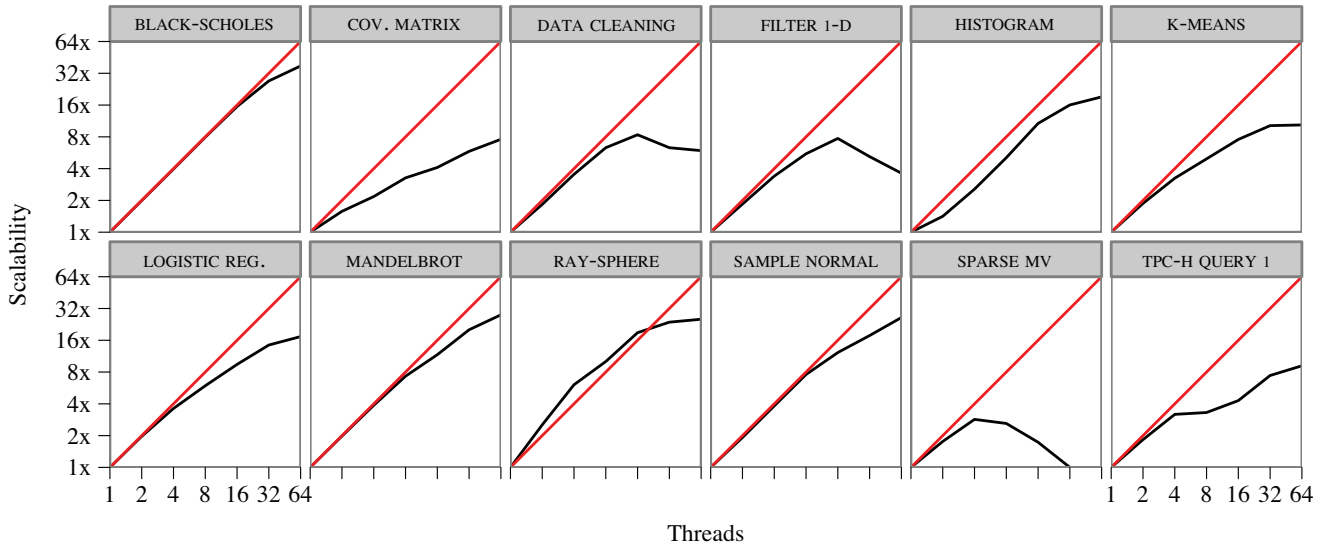


Figure 9: Scalability of Riposte on a 32-core Nehalem machine with hyperthreading. The red line is linear scalability. Riposte scales well on most workloads. The poor scalability in the SPARSE MATRIX-VECTOR MULTIPLICATION workload is due to poor support for high-cardinality multi-threaded reductions in our current implementation.

8. CONCLUSION

We have presented a trace-based approach for efficiently executing dynamically typed data analytic languages on modern hardware. The resulting virtual machine performs 5–50x faster than standard R—nearly as well as vectorized C code—and can scale well on multicore, shared memory machines. Thus, without changing to a statically-typed programming language or having to explicitly parallelize, R programmers using Riposte can see an overall speed-up of two orders of magnitude on interesting data analytic workloads.

Substantial work remains to be done to improve our Riposte implementation. We want to address the limitations of Riposte’s current parallel runtime highlighted in our evaluation by adding support for prefix scans and a better high-cardinality reduction implementation [36]. We also want to increase the set of operators that can be delayed. This should increase the average length of traces and thus the benefits from fusion, as well as increase the likelihood that we will be able to identify futures as dead, decreasing unnecessary memory usage.

Even though Riposte currently uses an interpreter to run scalar code, we want to integrate our vector virtual machine with modern techniques for executing dynamic scalar code such as function- or trace-based JITs. We will also continue to improve the performance of our JIT compiler (currently 90% of its time is spent in the optimization passes), allowing us to compile shorter vectors profitably. These two directions will permit us to experiment with efficiently supporting short vector code and mixed scalar/vector code.

Finally, we plan to expand the design of our vector virtual machine to support compilation to newer vector architectures such as GPUs and Intel’s AVX or MIC [30] architectures, and to consider the issues of scheduling vector code on distributed memory clusters.

In summary, Riposte allows data analysts to leverage emerging hardware trends to improve the performance and scale of their analyses while continuing to code easily and productively.

9. ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1111943.

10. REFERENCES

- [1] Google V8 Javascript engine. <http://code.google.com/p/v8/>.
- [2] The LuaJIT project. <http://lua-jit.org/>.
- [3] The Ra extension to R. <http://www.milbo.users.sonic.net/ra/>.
- [4] P. S. Abrams. *An APL Machine*. PhD thesis, Stanford Linear Accelerator Center, Stanford University, Stanford, CA, USA, 1970.
- [5] A. Aslam and L. Hendren. McFLAT: a profile-based framework for Matlab loop analysis and transformations. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing, LCPC’10*, pages 1–15, Berlin, Heidelberg, 2011. Springer-Verlag.
- [6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI ’00*, pages 1–12, New York, NY, USA, 2000. ACM.
- [7] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.
- [8] S. Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European conference on Object-oriented programming, ECOOP’10*, pages 429–451, Berlin, Heidelberg, 2010. Springer-Verlag.
- [9] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPOPP ’11*, pages 47–56, New York, NY, USA, 2011. ACM.

- [10] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing Matlab through just-in-time specialization. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 46–65. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11970-5_4.
- [11] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.
- [12] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 33–42, New York, NY, USA, 2006. ACM.
- [13] 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 '09, pages 465–478, New York, NY, USA, 2009. ACM.
- [14] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM.
- [15] V. Grover and Y. Lin. Compiling CUDA and other languages for GPUs. In *GPU Technology Conference (GTC)*, 2012.
- [16] L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 1–8, New York, NY, USA, 1978. ACM.
- [17] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.
- [18] N. Lameed and L. Hendren. Staged static techniques to efficiently implement array copy semantics in a Matlab JIT compiler. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 22–41, Berlin, Heidelberg, 2011. Springer-Verlag.
- [19] T. C. Miller. Tentative compilation: A design for an APL compiler. *SIGAPL APL Quote Quad*, 9:88–95, May 1979.
- [20] 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.
- [21] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization (CGO) 2011*, pages 224–235, April 2011.
- [22] M. Papakipos. The PeakStream platform: High productivity software development for multi-core processors. Technical report, 2006.
- [23] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.
- [24] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of the 2012 Innovative Parallel Computing: Foundations & Applications of GPU, Manycore, and Heterogeneous Systems*, InPar '12, 2012.
- [25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, Oct. 2005.
- [26] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, Sept. 1999.
- [27] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0.
- [28] A. R. Runnalls and C. A. Silles. CXXR: An ideas hatchery for future R development. In *Proceedings of the 2011 Joint Statistical Meetings (JSM)*, 2011.
- [29] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the art in parallel computing with R. *Journal of Statistical Software*, 31(1):1–27, 8 2009.
- [30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.
- [31] L. Tierney. Code analysis and parallelizing vector operations in R. *Computational Statistics*, 24:217–223, 2009. 10.1007/s00180-008-0117-9.
- [32] L. Tierney. A byte code compiler for R. Technical report, 2012.
- [33] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '10, pages 179–190, New York, NY, USA, 2010. ACM.
- [34] D. Wentzlaff and A. Agarwal. Factored operating systems (FOS): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.
- [35] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.
- [36] Y. Ye, K. A. Ross, and N. Vespapunt. Scalable aggregation on multicore processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 1–9, New York, NY, USA, 2011. ACM.
- [37] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206, July 2007.