Optimizing Data Structures in High-Level Programs

New Directions for Extensible Compilers based on Staging

Tiark Rompf [*‡] Arvind K. Sujeeth[†] Nada Amin[*] Kevin J. Brown[†] Vojin Jovanovic[*]
HyoukJoong Lee[†] Manohar Jonnalagedda[*] Kunle Olukotun[†] Martin Odersky[*]

[*]EPFL: {first.last}@epfl.ch [‡]Oracle Labs [†]Stanford University: {asujeeth, kjbrown, hyouklee, kunle}@stanford.edu

Abstract

High level data structures are a cornerstone of modern programming and at the same time stand in the way of compiler optimizations. In order to reason about user or library-defined data structures, compilers need to be extensible. Common mechanisms to extend compilers fall into two categories. Frontend macros, staging or partial evaluation systems can be used to programmatically remove abstraction and specialize programs before they enter the compiler. Alternatively, some compilers allow extending the internal workings by adding new transformation passes at different points in the compile chain or adding new intermediate representation (IR) types. None of these mechanisms alone is sufficient to handle the challenges posed by high level data structures. This paper shows a novel way to combine them to yield benefits that are greater than the sum of the parts.

Instead of using staging merely as a front end, we implement internal compiler passes using staging as well. These internal passes delegate back to program execution to construct the transformed IR. Staging is known to simplify program generation, and in the same way it can simplify program transformation. Defining a transformation as a staged IR interpreter is simpler than implementing a low-level IR to IR transformer. With custom IR nodes, many optimizations that are expressed as rewritings from IR nodes to staged program fragments can be combined into a single pass, mitigating phase ordering problems. Speculative rewriting can preserve optimistic assumptions around loops.

We demonstrate several powerful program optimizations using this architecture that are particularly geared towards data structures: a novel loop fusion and deforestation algorithm, array of struct to struct of array conversion, object flattening and code generation for heterogeneous parallel devices. We validate our approach using several non trivial case studies that exhibit order of magnitude speedups in experiments.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Run-time environments; D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming

General Terms Design, Languages, Performance

Keywords Staging, Code Generation, Data Structures, Extensible Compilers

```
// Vectors
object Vector {
  def fromArray[T:Numeric](a: Array[T]) =
    new Vector { val data = a }
  def zeros[T:Numeric](n: Int) =
    Vector.fromArray(Array.fill(n)(i => zero[T]))
}
abstract class Vector[T:Numeric] {
  val data: Array[T]
  def +(that: Vector[T]) =
    Vector.fromArray(data.zipWith(that.data)(_ + _))     ...
}
// Matrices
abstract class Matrix[T:Numeric] { ... }
// Complex Numbers
case class Complex(re: Double, im: Double) {
  def +(that: Complex) = Complex(re + that.re, im + that.im)
  def *(that: Complex) = ...
}
// (elided Numeric[Complex] type class implementation)
```

Figure 1. Skeleton of a high-level Scala linear algebra package

1. Introduction

Compiling high-level programs to efficient low-level code is hard, particularly because such programs use and define high-level abstractions. The compiler cannot see through abstractions ("abstraction penalty"), and it cannot reason about domain-specific properties ("general purpose bottleneck"). Among the most important abstractions are data structure and collection operations, and those also commonly present the most difficulties to an optimizing compiler.

Let us consider an example of high level programming in Scala. We would like to implement a dense linear algebra package. Figure 1 shows a skeleton implementation of Vectors and Matrices as a thin layer over Arrays, using high-level collection operations (fill, zipWith) internally. Vectors and Matrices contain numeric values (type class Numeric). We also define Complex numbers as a new numeric type.

With these definitions at hand, we can write programs like the following:

```
def diag(k:Int) = k * Matrix.identity(n)
val m1 = (v1+v2).transpose * (v1+v2)
val m2 = diag(l)
if (scale) println(m1*m2) // == m1*(k*id) == k*m1*id == k*m1
else println(m1)          // no need to compute m2
```

This code is elegant and high level, demonstrating how Scala's focus on abstraction and generalization increases development productivity. But unfortunately the code will run very slowly, one or two orders of magnitude slower than a tuned low-level implementation using just arrays and while loops (see Section 5).

What exactly is going wrong? Some of the reasons are:

1. Neither the Scala compiler nor the just-in-time compiler inside the JVM can apply generic optimizations, like common subexpression or dead code elimination (CSE, DCE), to non-trivial matrix or vector computations.
2. The involved compilers have no notion of domain-specific laws like m*id=m (where m is a matrix and id the identity matrix), which could be used for optimization.
3. Programming in a functional programming style creates lots of intermediate objects.
4. The uniform heap-allocated object representation on the JVM is inefficient for complex numbers.

In order to enable a compiler to reason about programs with high-level abstractions, we need mechanisms to extend the compiler so that it is able to resolve those abstractions. There are two common approaches. The first is to translate the abstractions away before the program reaches the compiler proper, so that the compiler does not need to be aware of them. This is the idea behind macro systems and staging. Partial evaluation (PE) can also be used to specialize programs before they reach the actual compiler. The second option is to teach the compiler new domain-specific rules. Usually, compiler extensibility is understood as a means to add new phases. Some extensible compilers also allow adding new IR types but often it is not clear how new nodes interact with existing generic optimizations.

However, neither of these approaches alone is sufficient to handle the challenges posed by high-level data structures and abstractions. Going back to our example, if all we have is staging or macros, then the expression `m*id`, which is equivalent to `m`, will be expanded into while loops before it even reaches the compiler, so no simplification to `m` can take place. In general, limited forms of simplification can be added (see C++ expression templates [51]) but to be fully effective, the full range of generic compiler optimizations (CSE, constant propagation, etc) would need to be duplicated, too. If on the other hand all we have is a facility to add new compiler passes, then we can add an optimization pass that simplifies `m*id` to `m`, but we need another pass that expands matrix multiplications into loops. These passes need to be implemented as low-level IR to IR transformations, which is much more difficult than the macro or staging approach that can use regular (multi-stage) computation to express the desired target code. Implementing optimizations as separate passes also leads to phase ordering problems if multiple optimizations are added independently.

We argue that what we really want is the best of both worlds: On the one hand, we want to treat operations symbolically so that they can be matched by transformation rules. But on the other hand we also want the power of staging to programmatically define the result of a transformation. In addition to that, we need a way to define transformations independently but avoid phase ordering problems when optimizations are applied.

Contributions In this paper, we present an extensible compiler architecture that solves this challenge, while still keeping the programming effort required to express optimizations and transformations low. Our approach achieves large speedups on high-level programs by fusing collection operations, changing data layout and applying further optimizations on high-level objects, enabled by intermediate languages with staging and a facility to combine independently specified optimizations without phase ordering issues.

To illustrate at a high level how our system works, consider again the linear algebra case. We use staging to obtain an intermediate representation from the initial program. In order to avoid the problems of the pure staging or macro approach, we apply optimizations at different levels, and we combine as many optimizations as possible together in a single pass to avoid many of the traditional phase-ordering problems. Linear algebra optimizations are implemented by the library author as rewrite rules that rewrite symbolic expressions into staged program fragments. The system applies rewriting speculatively using optimistic assumptions and rolls back intermediate transformations when they later turn out to be unsound. This strategy eliminates phase ordering problems that are due to one optimization module having to make pessimistic assumptions about the outcome of another one. Once no further simplification is possible on the linear algebra level, we want to switch to a lower level representation which consists of arrays and loops. We implement this kind of 'lowering' transform as a staged interpreter over the intermediate program. Since the interpreter again uses staging, it constructs a new program and thus acts as a program transformer. By default, this interpreter maps each expression to a structurally equivalent one. The library author extends it to map linear algebra operations to their lower level representation. On this lower level, the system applies another set of optimizations (e.g. loop fusion) in a combined pass, re-applying global optimizations to take advantage of new opportunities exposed by changing the representation. This process can be repeated for any desired number of lowering steps.

In particular, we make the following contributions:

- We use staging to build extensible multi-pass compilers that can also profitably combine modular optimizations into single passes: Staged IR interpreters act as IR transformers and speculative rewriting allows combining independently specified optimization while keeping optimistic assumptions.
- We use a graph-based intermediate representation that may contain structured, compound expressions. Splitting and merging compound expressions allows reusing existing optimization on their pieces (e.g. DCE to remove unused parts of a data structure). This approach extends to powerful data format conversions (e.g. array-of-struct to struct-of-array).
- We present a novel data parallel loop fusion algorithm that uniformly handles horizontal and vertical fusion and also includes asymmetric traversals (`flatMap`, `groupBy`).
- We demonstrate how this compiler architecture can solve tough optimization problems related to data structures in a number of nontrivial case studies.

We build on our previous work on Lightweight Modular Staging (LMS) [39] and highlight similarities and differences to previous work as we go along. The speculative rewriting approach is based on earlier work by Lerner, Grove and Chambers [29].

Organization We start out by reviewing partial evaluation and staging (§2). Insights from this section will help understand how we use staging for program transformation (§3), where we first present symbolic optimizations using speculative rewriting (§3.2) before delving into lowering passes as staged IR interpreters (§3.3). A third transformation we explore is the splitting and merging of compound expressions to reuse existing optimizations (§3.4). We then present how these techniques can be used to perform data structure optimizations (§4): our generic staged struct abstraction (§4.1), which extends to unions and inheritance (§4.2), an array of struct to struct of array transform (§4.3), and loop fusion (§4.4). We then present a set of case studies where our transformations are particularly appealing (§5): linear algebra (§5.1), regular expression matching (§5.2), collection and query operations (§5.3), and string templating (§5.4). Finally we discuss our results (§6), review related work (§6.1) and conclude (§6.2).

2. Background

Many computations can naturally be separated into stages distinguished by frequency of execution or availability of information. Multi-stage programming (MSP, *staging* for short) as established by Taha and Sheard [46] make the stages explicit and allows programmers to delay evaluation of a program expression to a later stage (thus, *staging* an expression). The present stage effectively acts as a code generator that, when run, produces the program of the next stage.

Staging is closely related to partial evaluation [20], which specializes programs to statically known parts of their input. For the purpose of this paper, we can treat partial evaluation (and in particular binding-time analysis (BTA)) as automatic staging and staging as programmer-controlled partial evaluation.

A key difference is that partial evaluation strictly specializes programs and usually comes with soundness guarantees whereas adding staging annotations to a program in an MSP language such as MetaOCaml [7] provides more freedom for composing staged fragments but requires some care so as not to change the computed result.

Much of the research on staging and partial evaluation was driven by the desire to simplify compiler development. For example, specializing an interpreter to a particular program yields a compiled program with the interpreter overhead removed (the first Futamura projection [15]). Self applicable partial evaluators can generate compilers from interpreters and compiler generators.

The exposition in this paper uses Scala and Lightweight Modular Staging (LMS) [39], a library-only staging approach. Contrary to dedicated MSP languages based on quasi quotation, LMS uses only types to distinguish the computational stages. Expressions belonging to the second stage have type Rep[T] in the first stage when yielding a computation of type T in the second stage. Expressions of a plain type T will be evaluated in the first stage and become constants in stage two. The plain Scala type system propagates information about which expressions are staged and thus performs a semi-automatic local BTA. Thus, LMS shares some of the benefits of automatic PE and manual staging.

Example: Zero-Overhead Traversal Abstractions Arrays in Scala are bare JVM arrays which need to be traversed using while loops and indices. The Scala standard library provides an enrichment that adds a foreach method:

```
array foreach { i => println(i) }
```

Adding foreach is achieved using an implicit conversion:

```
implicit def enrichArray[T](a: Array[T]) = new {
  def foreach(f: T => Unit): Unit =
    { var i = 0; while (i < a.length) { f(a(i)); i += 1 } }
}
```

This implementation has non-negligible abstraction overhead (closure allocation, interference with JVM inlining, etc). We would like to tell the compiler, whenever it sees a foreach invocation, to just put the while loop there instead. This is a simple case where macros or staging can help.

Using LMS, we just change the method argument types. Figure 2 shows a set of staged array and vector operations. The LMS framework provides overloaded variants of many operations that lift those operations to work on Rep types, i.e. staged expressions rather than actual data.

It is important to note the difference between types `Rep[A=>B]` (a staged function object) and `Rep[A]=>Rep[B]` (a function on staged values). By using the latter, foreach ensures that the function parameter is always evaluated and unfolded at staging time. Macro systems that only allow lifting expression trees support only types `Rep[A=>B]`. This limits expressiveness and there are no guarantees that higher order functions are evaluated at staging time.

In addition to the LMS framework, we use the Scala-Virtualized compiler [32] which redefines several core language features as method calls and thus makes them overloadable as well. For example, the code

```
// Array
implicit def enrichArray[T](a: Rep[Array[T]]) = new {
  def foreach(f: Rep[T] => Rep[Unit]): Rep[Unit] =
    { var i = 0; while (i < a.length) { f(a(i)); i += 1 } }
  def zipWith(b: Rep[Array[T]])(f: (Rep[T],Rep[T]) => Rep[T]) =
    Array.fill(a.length min b.length) { i => f(a(i), b(i)) }
}
// Vector
trait Vector[T] extends Struct { val data: Array[T] }
implicit def enrichVector[T:Numeric](a: Rep[Vector[T]]) = new {
  def +(b: Rep[Vector[T]]) =
    Vector.fromArray(a.data.zipWith(b.data)(_ + _))       ...
}
// (companion objects define Array.fill and Vector.fromArray)
```

Figure 2. Staged Array and Vector Ops.

```
var i = 0; while (i < n) { i = i + 1 }
```

will be desugared as follows:

```
val i = __newVar(0); __while(i < n, { __assign(i, i + 1) })
```

Methods `__newVar`, `__assign`, `__while` are overloaded to work on Rep types. These methods need to be suitably defined and made available in scope. Scala-Virtualized also provides overloaded field access and object construction methods. In the declaration of vectors or complex numbers, extending `Struct` serves as a marker to automatically lift object construction and field accesses to the domain of Rep types. This means that staged field accesses such as `v.data` are transparently available on `Rep[Vector[T]]` values and (in this case) would return `Rep[Array[T]]` values. Similarly,

```
new Vector { val data = /* type Rep[Array[T]] */ }
```

will return a `Rep[Vector[T]]` object.

Generic Programming with Type Classes Figure 2 uses the type class `Numeric` to abstract over particular numeric types. The type class pattern [55], which decouples data objects from generic dispatch fits naturally with a staged programming model. We can define a staged variant of the standard `Numeric` type class and, with addition on numeric vectors defined in Figure 2, make vectors themselves instance of Numeric:

```
class Numeric[T] { def num_plus[T](a:Rep[T],b:Rep[T]): Rep[T] }
implicit def vecIsNumeric[T:Numeric] = new Numeric[Vector[T]] {
  def num_plus(a: Rep[Vector[T], b: Rep[Vector[T]]) = a + b
}
```

This allows us to pass, say, a staged `Vector[Int]` to any function that works over generic types `T:Numeric`, such as vector addition itself. The same holds for `Vector[Vector[Int]]`. Without staging, type classes are implemented by passing an implicit dictionary, the type class instance, to generic functions. Here, type classes are a purely stage-time concept. All generic code is specialized to the concrete types and no type class instances exist (and hence no virtual dispatch occurs) when the staged program is run.

Maintaining Evaluation Order Compared to staging or macro systems based on quasi quotation, LMS preserves program semantics in more cases. In particular, adding Rep types does not change the relative evaluation order of statements within a stage. In

```
compute() foreach { i => println(i) }
```

the staged foreach implementation from Figure 2 will evaluate compute() only once, whereas purely syntactic expansion would produce this target code:

```
while (i < compute().length) { println(compute()(i)); i += 1 }
```

LMS performs on-the fly ANF conversion, similar to earlier work on PE with effects [49]. Systems based on quasi quotation could profitably use the same method to maintain evaluation order but we are unaware of any sustem that does. Instead, most other systems leave it up to the programmer to insert let-bindings in the right places, which can easily lead to subtle errors.

Limitations of Front-End Staging and Macros Despite the given benefits, for top performance it is often not sufficient to use staging (or macros, or partial evaluation) as a front end. Let us consider a simple yet non-trivial example:

```
val v1 = ...
v1 + Vector.zeros(n)
```

Staging will replace the zero vector creation and the subsequent addition with arrays and loops. What we would like instead, however, is to apply a symbolic simplification rule v+zero->v to remove the zero addition before expansion takes place.

Let us imagine that our system would allow us to implement the vector plus operation in such a way that it can inspect its arguments to look for invocations of `Vector.zeros`. This would cover the simple case above but we would still run into problems if we complicate the use case slightly:

```
val v1 = ...
val v2 = Vector.zeros(n)
v1 + v2
```

To handle programs like this, it is not sufficient to just inspect the (syntactic) arguments. We need to integrate the staging expansion with some form of forward data flow propagation, otherwise the argument to plus is just an identifier.

The deeper problem is that we are forced to commit to a single data representation. Even if we combine staging with an extensible compiler we need to make a decision: Should we treat vectors as symbolic entities with algebraic laws, implemented as IR nodes amenable to optimization? Or should we stage them so that the compiler sees just loops and arrays without abstraction overhead?

The following sections will discuss mechanisms to integrate these approaches and for treating data structures in a more abstract way.

3. Program Transformation via Staging

Staging usually is a method for generating programs: A multi-stage program builds an object program which is then compiled normally again. We show that staging is also useful as a tool for transforming programs. Any transformation can be broken down into a traversal and a generation part. Not surprisingly, staging helps with the generation part. In our case, internal compiler passes are IR interpreters that happen to be staged so that they return a new program as the result. This way they can delegate back to program execution to build the result of a program transformation.

Staging for program transformation has a number of benefits. First, building a (staged) IR interpreter is far simpler than building a non-trivial IR to IR transformer. Second, optimizations can be added gradually to a staged program, starting, e.g., with the code from Figure 2. Third, the program (or library) itself is in control of the translation and can influence what kind of code is generated.

One of the key aspects of our approach is to distinguish two kinds of transforms: Optimizations and Lowerings. Lowerings translate programs into a lower-level representation (e.g. linear algebra operations into arrays and loops). Lowerings have a natural ordering so they can be profitably arranged in separate passes. Optimizations, by contrast, have no clear ordering and are prone to phase ordering problems. Thus, they need to be combined for maximum effectiveness. Also, most lowerings are mandatory whereas optimizations are usually optional. Of course the distinction is not always clear cut but many transforms fall into only one of the categories. In any case, it is important that all applicable optimizations are applied exhaustively before lowering takes place. Otherwise, high-level optimization opportunities may be missed. After a lowering step, there may be new opportunities for optimization. Thus, our system performs a sequence of optimization, lowering, optimization steps, until the lowest-level representation is reached. The final representation is unparsed to target code.

We recap the LMS extensible IR (§3.1) and first consider optimizations (§3.2), then lowerings (§3.3). Afterwards we descibe the treatment of compound expressions (§3.4).

3.1 The LMS Extensible Graph IR

We now turn to the level of 'primitive' staged operations. Using LMS we do not directly produce the second stage program in source form but instead as an extensible intermediate representation (IR). We refer the reader to our previous work for details on the IR [37–39, 41] but give a short recap here. The overall structure is that of a "sea of nodes" dependency graph [9].

In Figure 3, which will be the running example for this section, we recast the vector implementation from Figure 2 in terms of custom IR nodes. The user-facing interface is defined in trait `Vectors`, with abstract methods (`vec_zeros`, `vec_plus`) that are implemented in trait `VectorsExp` to create IR nodes of type `VectorZeros` and `VectorPlus`, respectively.

The framework provides IR base classes via trait `BaseExp`, mixed into `VectorsExp` (but not `Vectors`, to keep the IR hidden from user code). Expressions are atomic:

```
abstract class Exp[T]
case class Const[T](x: T) extends Exp[T]
case class Sym[T](n: Int) extends Exp[T]
```

Trait `BaseExp` defines `Rep[T]=Exp[T]`, whereas `Rep[T]` is left as an abstract type in trait Base.

Custom (composite) IR nodes extend `Def[T]`. They refer to other IR nodes only via symbols. There is also a type `Block[T]` to define nested blocks (not used in Figure 3). Taking a closer look at `vec_zero` reveals that its expected return type is `Exp[T]` but the result value is of type `Def[T]`. This conversion is achieved implicitly by `toAtom`:

```
implicit def toAtom[T](d: Def[T]): Exp[T] = reflectPure(d)
```

Method `reflectPure` maintains the correct evaluation order by binding the argument `d` to a fresh symbol (on the fly ANF conversion).

```
def reflectPure[T](d: Def[T]): Sym[T]
def reifyBlock[T](b: =>Exp[T]): Block[T]
```

The counterpart `reifyBlock` (note the by-name argument) captures performed statements into a block object. Additional reflect methods exist to mark IR nodes with various kinds of side effects (see [41] for details).

3.2 Combining Optimizations: Speculative Rewriting

Many optimizations that are traditionally implemented using an iterative dataflow analysis followed by a transformation pass can also be expressed using various flavors of (possibly context dependent) rewriting. Whenever possible we tend to prefer a rewriting variant because rewrite rules are easy to specify separately and do not require programmers to define abstract interpretation lattices.

Rewrites and Smart Constructors LMS performs forward optimizations eagerly while constructing the IR. Hash-consing implements CSE and smart constructors apply pattern rewriting, including various kinds of constant propagation. This can be seen as adding "online" PE to the existing "offline" stage distinction defined by Rep types [42, 45]. In Figure 3, trait `VectorsExpOpt` can be mixed in with `VectorsExp` and overrides `vec_plus` to simplify zero additions. Rewriting is integrated with other forward optimizations: Pattern matches of the form `case Def(VectorZeros(n)) => ...` will look up the available definition of a symbol. Here is a simple example program, before (left) and after constant folding (middle):

```
val x = 3 * y          val x = 3 * y          println(6 * y)
println(2 * x)         println(6 * y)
```

DCE will later remove the multiplication and the binding for `x` (right).

```
// Vector interface
trait Vectors extends Base {
  // elided implicit enrichment boilerplate:
  //   Vector.zeros(n) = vec_zeros(n), v1 + v2 = vec_plus(a,b)
  def vec_zeros[T:Numeric](n: Rep[Int]): Rep[Vector[T]]
  def vec_plus[T:Numeric](a: Rep[Vector[T]], b: Rep[Vector[T]]): Rep[Vector[T]]
}
// low level translation target
trait VectorsLowLevel extends Vectors {
  def vec_zeros_ll[T:Numeric](n: Rep[Int]): Rep[Vector[T]] =
    Vector.fromArray(Array.fill(n) { i => zero[T] })
  def vec_plus_ll[T:Numeric](a: Rep[Vector[T]], b: Rep[Vector[T]]) =
    Vector.fromArray(a.data.zipWith(b.data)(_ + _))
}
// IR level implementation
trait VectorsExp extends BaseExp with Vectors {
  // IR node definitions and constructors
  case class VectorZeros(n: Exp[Int]) extends Def[Vector[T]]
  case class VectorPlus(a: Exp[Vector[T]],b: Exp[Vector[T]]) extends Def[Vector[T]]
  def vec_zeros[T:Numeric](n: Rep[Int]): Rep[Vector[T]] = VectorZeros(n)
  def vec_plus[T:Numeric](a: Rep[Vector[T]], b: Rep[Vector[T]]) = VectorPlus(a,b)
  // mirror: transformation default case
  def mirror[T](d: Def[T])(t: Transformer) = d match {
    case VectorZeros(n) => Vector.zeros(t.transformExp(n))
    case VectorPlus(a,b) => t.transformExp(a) + t.transformExp(b)
    case _ => super.mirror(d)
  }
}
// optimizing rewrites (can be specified separately)
trait VectorsExpOpt extends VectorsExp {
  override def vec_plus[T:Numeric](a:Rep[Vector[T]],b:Rep[Vector[T]])=(a,b)match{
    case (a, Def(VectorZeros(n))) => a
    case (Def(VectorZeros(n)), b) => b
    case _ => super.vec_plus(a,b)
  }
}
// transformer: IR -> low level impl
trait LowerVectors extends ForwardTransformer {
  val IR: VectorsExp with VectorsLowLevel; import IR._
  def transformDef[T](d: Def[T]): Exp[T] = d match {
    case VectorZeros(n) => vec_zeros_ll(transformExp(n))
    case VectorPlus(a,b) => vec_plus_ll(transformExp(a), transformExp(b))
    case _ => super.transformDef(d)
  }
}
```

Figure 3. Vector Implementation with IR and Lowering.

Applying Rewrites Speculatively Many optimizations are mutually beneficial. In the presence of loops, optimizations need to make optimistic assumptions for the supporting analysis to obtain best results. If multiple analyses are run separately, each of them effectively makes pessimistic assumptions about the outcome of all others. Combined analyses avoid the phase ordering problem by solving everything at the same time. The challenge, of course, is to automatically combine analyses and transformations that are implemented independently of one another.

Since the forward optimizations described above are applied at IR construction time where loop information is incomplete, LMS previously allowed rewrites and CSE only for purely functional operations and not in the presence of imperative loops. These pessimistic assumptions, together with monolithic compound expressions (see Section 3.4) prevented effective combinations of separate rewrites.

Lerner, Grove, and Chambers showed a method of composing separately specified optimizations by interleaving analyses and transformations [29]. We use a modified version of their algorithm that works on structured loops instead of CFGs and using dependency information and rewriting instead of explicit data flow lattices. To the best of our knowledge, we are the first to extend this approach to extensible compilers and a purely rewriting based environment. Usually, rewriting is semantics preserving, i.e. pessimistic. The idea is to drop that assumption. As a corollary, we need to rewrite speculatively and be able to rollback to a previous state to get optimistic optimization. The algorithm proceeds as follows: for each encountered loop, apply all possible transforms to the loop body, given empty initial assumptions. Analyze the result of the transformation: if any new information is discovered throw away the transformed loop body and retransform the original with updated assumptions. Repeat until the analysis result has reached a fixpoint and keep the last transformation as result. This process can be costly for deeply nested loops, but compares favorably to the alternative of running independent transformations one after another until a global fixpoint is reached [29].

Here is an example of speculative rewriting, showing the untransformed program (left), the initial optimistic iteration (middle), and the fixpoint (right) reached after the second iteration:

```
var x = 7                 var x = 7          var x = 7 //dead
var c = 0                 var c = 0          var c = 0
while (c < 10) {          while (true) {     while (c < 10) {
  if (x < 10) print("!")    print("!")         print("!")
  else x = c                print(7)           print(7)
  print(x)                  print(0)           print(c)
  print(c)                  c = 1              c += 1
  c += 1                  }                  }
}
```

This algorithm allows us to do all forward data flow analyses and transforms in one uniform, combined pass driven by rewriting. In the example above, during the initial iteration (middle), separately specified rewrites for variables and conditionals work together to determine that x=c is never executed. At the end of the loop body we discover the write to c, which invalidates our initial optimistic assumption c=0. We rewrite the original body again with the new information (right). This time there is no additional knowledge discovered so the last speculative rewrite becomes the final one.

Speculative rewriting as used here is still a fully static transform–not to be confused with speculative dynamic optimizations performed inside virtual machines that may incur multiple profiling, compilation and deoptimization cycles.

3.3 Separate Lowering Passes: Transformers

Previously LMS had a single code generation pass that scheduled the IR graph, performing DCE and code motion. We generalize this facility to allow arbitrary traversals of the program tree which results from scheduling the IR graph. Similar to other optimizations, DCE and code motion work on high-level (possibly composite) IR nodes.

Generic IR Traversal and Transformation Once we have a traversal abstraction, transformation falls out naturally by building a traversal that constructs a new staged program. The implementation is shown in Figure 4. Transformation needs a default case, which we call *mirroring* (see trait `VectorsExp` in Figure 3). Mirroring an IR node will call back to the corresponding smart constructor, after applying a transformer to its arguments. Mirroring loops is slightly more complicated than what is shown in Figure 3 because of the bound variable:

```
case ArrayFill(n,i,y) =>
  Array.fill(transformExp(n), { j =>
    t.withSubst(i -> j) { transformBlock(y) } }
```

Implementing Custom Transformers In our running example, we would like to treat linear algebra operations symbolically first, with individual IR nodes like `VectorZeros` and `VectorPlus`. In Figure 3, the smart constructor `vec_plus` implements a rewrite that simplifies V+Z to V. CSE, DCE, etc. will all be performed on these high level nodes.

After all those optimizations are applied, we want to transform our operations to the low-level array implementation from Figure 2 in a separate lowering pass. Trait `LowerVectors` in Figure 3 implements this transformation by delegating back to user-space code,

```
// traversal
trait ForwardTraversal {
  val IR: Expressions; import IR._
  def traverseBlock[T](b: Block[T]): Unit =
    focusExactScope(block) { stms =>
      stms foreach traverseStm }
  def traverseStm[T](s: Stm[T]): Unit =
    blocks(stm.rhs) foreach traverseBlock
}
// transform
trait ForwardTransformer extends ForwardTraversal {
  val IR: Expressions; import IR._
  var subst: Map[Exp[_],Exp[_]]
  def transformExp[T](s: Exp[T]): Exp[T] = // lookup s in subst
  def transformBlock[T](b: Block[T]): Exp[T] = scopeSubst {
    traverseBlock(b); transformExp(b.res) }
  def transformDef[T](d: Def[T]): Exp[T] = mirror(d, this)
  override def traverseStm(s: Stm[T]) = {
    val e = transformDef(s.rhs); subst += (s.sym -> e); e }
}
```

Figure 4. Traversal and Transformer Interface.

namely method `vec_plus_ll` in trait `VectorsLowLevel`. The result of the transform is a staged program fragment just like in Figure 2.

This setup greatly simplifies the definition of the lowering transform, which would otherwise need to assemble the `fill` or `zipWith` code using low level IR manipulations. Instead we benefit directly from the staged `zipWith` definition from Figure 2. Also, further rewrites will take place automatically. Essentially all simplifications are performed eagerly, after each transform phase. Thus we guarantee that CSE, DCE, etc. have been applied on high-level operations before they are translated into lower-level equivalents, on which optimizations would be much harder to apply. To give a quick example, the initial program

```
val v1 = ...
val v2 = Vector.zeros(n)
val v3 = v1 + v2
v1 + v3
```

will become

```
val v1 = ...
Vector.fromArray(v1.data.zipWith(v1.data)(_ + _))
```

after lowering (modulo unfolding of staged zipWith).

Worklist Transformers and Delayed Rewriting Transformers can be extended with a worklist, which is useful if the result may contain terms that need further transformation. With a worklist transformer, we can register individual rewrites for particular nodes. Running the transformer applies the rewrites, which may register new replacements for the next iteration. The process stops if no further work is required (empty work list).

Delayed rewriting is a simplified interface that allows specifying lowerings together with the operations and the regular immediate rewrites. This helps to reduce the boilerplate needed to define transformations. For example, here is the `VectorZeros` lowering from Figure 3 recast as delayed rewrite:

```
def vec_zeros[T:Numeric](n: Rep[Int]) =
  VectorZeros(n) atPhase(lowering) {
    Vector.fromArray(Array.fill(n) { i => zero[T] })
  }
```

The `atPhase` block is registered with worklist transformer `lowering`. Before lowering, the IR node remains a `VectorZeros` node, which allows other smart constructor rewrites to kick in that expect this pattern.

3.4 Compound Expressions: Split and Merge

Since our IR contains structured expressions like loops and conditionals, optimizations need to reason about compound statements. This is not easy: for example, a simple DCE algorithm will not be able to remove only pieces of a compound expression. Our solution is simple yet effective: We eagerly split many kinds of compound statements, assuming optimistically that only parts will be needed. Splitting is implemented just like any other rewrite, and thus integrates well with other unrelated optimizations (see Section 3.2). We find out which parts are needed through the regular DCE algorithm and afterwards, we reassemble the remaining pieces.

Effectful Statements A good example of statement splitting is effectful conditionals:

```
var a, b, c = ...      var a, b, c = ...      var a, c = ...
if (c) {               if (c)  a = 9          if (c) a = 9
  a = 9; b = 1         if (c)  b = 1          else   c = 3
} else                 if (!c) c = 3          println(a+c)
  c = 3                println(a+c)
println(a+c)
```

From the conditional in the initial program (left), splitting creates three separate expressions, one for each referenced variable (middle). DCE removes the middle one because variable b is not used, and the remaining conditionals are merged back together (right). Of course successful merging requires to keep track of how expressions have been split. An extension of this simple merge facility which attempts to merge expressions that may not have been split before is loop fusion (Section 4.4).

Data Structures Splitting is also very effective for data structures, as often only parts of a data structure are used or modified. This will be discussed in more detail in Section 4 below but here is already a quick example. Assume `c1` and `c2` are complex numbers:

```
val c3 = if (test) c1 else c2
println(c3.re)
```

The conditional will be split for each field of a struct. Internally the above will be represented as:

```
val c3re = if (test) c1re else c2re
val c3im = if (test) c1im else c2im   // dead
val c3 = Complex(c3re, c3im)          // dead
println(c3re)
```

The computation of the imaginary component as well as the struct creation for the result of the conditional are never used and thus they will be removed by DCE.

4. Data Structure Optimizations

High level data structures are a cornerstone of modern programming and at the same time stand in the way of compiler optimizations. We illuminate the main issues compilers have with data structures as used by the two dominant programming paradigms.

OOP Object oriented programming treats every data value as an object. This is a powerful pattern that makes it easy to extend languages with new functionality. In Scala, e.g., it is easy to add a complex number class. But there is a price to be paid: allocating each complex number as a separate object will not perform well. Furthermore, if we are working with arrays of complex numbers we get much better performance if we use a struct of array representation. Staging and embedded compilers allow us to abstract away the object abstraction and reason about individual pieces of data that objects are composed of and possibly rearrange that data in more efficient ways.

FP Functional programs create lots of intermediate results. This is particularly bad for collection operations. Theoretically, compilers can leverage referential transparency. But impure functional languages need sophisticated effects analysis, which is hard if there is a lot of abstraction. Our staged programs are much simpler because abstraction is stripped away. We can do better and simpler effects analysis. A simple liveness analysis can turn copying into in-place modification. A novel loop fusion algorithm (data parallel and asymmetric, includes `flatMap` and `groupBy`) removes many intermediate results (Section 4.4).

```
// generic struct interface
trait StructExp extends BaseExp {
  abstract class StructTag
  case class Struct[T](tag: StructTag, elems: Map[String,Rep[Any]]) extends Def[T]
  case class Field[T](struct: Rep[Any], key: String) extends Def[T]
  def struct[T](tag: StructTag, elems: Map[String,Rep[Any]]) = Struct(tag, elems)
  def field[T](struct: Rep[Any], key: String): Rep[T] = struct match {
    case Def(Struct(tag, elems)) => elems(key).asInstanceOf[Rep[T]]
    case _ => Field[T](struct, key)
  }
}
// splitting array construction
case class ArraySoaTag(base: StructTag, len: Exp[Int]) extends StructTag
override def arrayFill[T](size: Exp[Int], v: Sym[Int], body: Def[T]) = body match {
  case Block(Def(Struct(tag, elems))) =>
    struct[T](ArraySoaTag(tag,size),
      elems.map(p => (p._1, arrayFill(size, v, Block(p._2)))))
  case _ => super.arrayFill(size, v, body)
}
// splitting array access
override def infix_apply[T](a: Rep[Array[T]], i: Rep[Int]) = a match {
  case Def(Struct(ArraySoaTag(tag,len),elems)) =>
    struct[T](tag, elems.map(p => (p._1, infix_apply(p._2, i))))
  case _ => super.infix_apply(a,i)
}
override def infix_length[T](a: Rep[Array[T]]): Rep[Int] = a match {
  case Def(Struct(ArraySoaTag(tag, len), elems)) => len
  case _ => super.infix_length(a)
}
```

Figure 5. Generic Struct interface and SoA Transform.

4.1 A Generic Struct Interface

Product types, i.e. records or structs are one of the core data structure building blocks. It pays off to have a generic implementation that comes with common optimizations (Figure 5). Trait `StructExp` defines two IR node types for struct creation and field access. The struct creation node takes a hash map that relates (static) field identifiers with (dynamic) values and a `tag` that can hold further information about the data representation. The method `field` tries to look up the desired value directly if the argument is a Struct node. With this struct abstraction we can implement the data structure splitting and merging example from Section 3.4 by overriding the if-then-else smart constructor to create a conditional for each field in a struct.

4.2 Unions and Inheritance

The struct abstraction can be extended to sum types and inheritance using a tagged union approach [21, 34]. We add a `clzz` field to each struct that refers to an expression that defines the object's class. Being a regular struct field, it is subject to all common optimizations. We extend the complex number example with two subtraits:

```
trait Complex
trait Cartesian extends Complex with Struct {
  val re: Double; val im: Double }
trait Polar extends Complex with Struct {
  val r: Double; val phi: Double }
```

Splitting transforms work as before: e.g. conditional expressions are forwarded to the fields of the struct. But now the result struct will contain the union of the fields found in the two branches, inserting null values as appropriate. A conditional is created for the `clzz` field only if the exact class is not known at staging time. Given straightforward factory methods `Cartesian` and `Polar`, the expression

```
val a = Cartesian(1.0, 2.0); val b = Polar(3.0, 4.0)
if (x > 0) a else b
```

produces this generated code:

```
val (re, im, r, phi, clzz) =
  if (x > 0) (1.0, 2.0, null, null, classOf[Cartesian])
  else (null, null, 3.0, 4.0, classOf[Polar])
struct("re"->re, "im"->im, "r"->r, "phi"->phi, "clzz"->clzz)
```

The `clzz` fields allows virtual dispatch via type tests and type casting, e.g. to convert any complex number to its cartesian representation:

```
def infix_toCartesian(c: Rep[Complex]) : Rep[Cartesian] =
  if (c.isInstanceOf[Cartesian]) c.asInstanceOf[Cartesian]
  else { val p = c.asInstanceOf[Polar]
    Cartesian(p.r * cos(p.phi), p.r * sin(p.phi)) }
```

Appropriate rewrites ensure that if the argument is known to be a Cartesian, the conversion is a no-op. The type test that inspects the clzz field is only generated if the type cannot be determined statically. If the clzz field is never used it will be removed by DCE.

4.3 Struct of Array and Other Data Format Conversions

A natural extension of the splitting mechanism is a generic array-of-struct to struct-of-array transform (AoS to SoA). The mechanism is analogous to that for conditionals. We override the array constructor `arrayFill` that represents expressions of the form `Array.fill(n) { i => body }` to create a struct with an array for each component of the body if the body itself is a Struct (Figure 5). Note that we tag the result struct with an `ArraySoaTag` to keep track of the transformation. We also override the methods that are used to access array elements and return the length of an array to do the right thing for transformed arrays.

The SoA data layout is beneficial in many cases. Consider for example calculating complex conjugates (i.e. swapping the sign of the imaginary components) over a vector of complex numbers.

```
def conj(c: Rep[Complex]) = if (c.isCartesian) {
    val c2 = c.toCartesian; Cartesian(c.re,-c.im)
  } else { val c2 = c.toPolar; Polar(c.r, -c.phi) }
```

To make the test case more interesting we perform the calculation only in one branch of a conditional:

```
val vector1 = Array.fill(100) { i => Cartesian(...) }
if (test) vector1.map(conj) else vector1
```

All the real parts remain unchanged so the array holding them need not be touched at all. Only the imaginary parts have to be transformed, cutting the total required memory bandwidth in half. Uniform array operations like this are also a much better fit for SIMD execution. The generated intermediate code is:

```
val vector1re,vector1im = ...
val vector1clzz = // array holding classOf[Cartesian] values
val vector2im = if (test) Array.fill(vector1size) {
  i => -vector1im(i) } else vector1im
struct(ArraySoaTag(Complex, vector1size),
  Map("re"->vector1re,"im"->vector2im,"clzz"->vector1clzz))
```

Note how the conditionals for the "re" and "clzz" fields have been eliminated since the fields do not change (the initial array contains cartesian numbers only). If the `struct` expression will not be referenced in the final code DCE removes the "clzz" array.

In the presence of conditionals that produce array elements of different types, a possible optimization would be to use a sparse representation for the arrays that make up the result SoA (similar to DPH [21]). However, all the usual sparse vs dense tradeoffs apply.

One concern with data representation conversions is what happens if an array is returned from a staged code fragment to the enclosing program. In this case, the compiler will generate conversion code to return a plain array-of-struct copy of the data.

4.4 Loop Fusion and Deforestation

The use of independent and freely composable traversal operations such as `v.map(..).sum` is preferable to explicitly coded loops. However, naive implementations of these operations would be expensive and entail lots of intermediate data structures. We present a novel loop fusion algorithm for data parallel loops and traversals. The core loop abstraction is

$$\texttt{loop(s)}\ \overline{\texttt{x} = \mathcal{G}}\ \{\ \texttt{i} \Rightarrow \overline{E[\texttt{x} \leftarrow \texttt{f(i)}]}\ \}$$

Generator kinds: $\mathcal{G}$ ::= Collect | Reduce($\oplus$) | Bucket($\mathcal{G}$)
Yield statement: xs $\leftarrow$ x
Contexts: $E[.]$ ::= loops and conditionals

Horizontal case (for all types of generators):

```
   loop(s) x1=𝒢1 { i1 => E1[ x1 ← f1(i1) ] }
   loop(s) y1=𝒢2 { i2 => E2[ x2 ← f2(i2) ] }
 ---------------------------------------------
 loop(s) x1=𝒢1, x2=𝒢2 { i =>
         E1[ x1 ← f1(i) ]; E2[ x2 ← f2(i) ] }
```

Vertical case (consume collect):

```
   loop(s) x1=Collect { i1 => E1[ x1 ← f1(i1) ] }
 loop(x1.size) x2=𝒢 { i2 => E2[ x2 ← f2(x1(i2)) ] }
 ----------------------------------------------------
 loop(s) x1=Collect, x2=𝒢 { i =>
           E1[ x1 ← f1(i); E2[ x2 ← f2(f1(i)) ]] }
```

Vertical case (consume bucket collect):

```
        loop(s) x1=Bucket(Collect) { i1 =>
           E1[ x1 ← (k1(i1), f1(i1)) ] }
   loop(x1.size) x2=Collect { i2 =>
     loop(x1(i2).size) y=𝒢 { j =>
       E2[ y ← f2(x1(i2)(j)) ] }; x2 ← y }
 ---------------------------------------------------
 loop(s) x1=Bucket(Collect), x2=Bucket(𝒢) { i =>
     E1[ x1 ← (k1(i), f1(i));
         E2[ x2 ← (k1(i), f2(f1(i))) ]] }
```

Figure 6. Loop fusion.

where s is the size of the loop and i the loop variable ranging over $[0, \mathtt{s})$. A loop can compute multiple results $\overline{\mathtt{x}}$, each of which is associated with a generator $\mathcal{G}$. There are three kinds of generators: Collect, which creates a flat array-like data structure, Reduce($\oplus$), which reduces values with the associative operation $\oplus$, or Bucket($\mathcal{G}$), which creates a nested data structure, grouping generated values by key and applying $\mathcal{G}$ to those with matching key. Loop bodies consist of *yield* statements x←f(i) that pass values to generators (of this loop or an outer loop), embedded in some outer context $E[.]$ that might consist of other loops or conditionals. Note that a yield statement x←.. does not introduce a binding for x, but passes a value to the generator identified by x. For Bucket generators, yield takes (key,value) pairs.

This model is expressive enough to represent many common collection operations:

```
x=v.map(f)     loop(v.size) x=Collect   { i => x ← f(v(i)) }
x=v.sum        loop(v.size) x=Reduce(+) { i => x ← v(i) }
x=v.filter(p)  loop(v.size) x=Collect   { i => if (p(v(i)))
                                                 x ← v(i) }
x=v.flatMap(f) loop(v.size) x=Collect   { i => val w = f(v(i))
                                      loop(w.size) { j => x ← w(j) }}
x=v.distinct   loop(v.size) x=Bucket(Reduce(rhs)) { i =>
                                             x ← (v(i), v(i)) }
```

Operation distinct uses a bucket reduction with function rhs, which returns the right-hand side of a tuple element, to return a flat sequence that contains only the rightmost occurence of a duplicate element.

Other operations are accommodated by generalizing slightly. Instead of implementing a groupBy operation that returns a sequence of (Key, Seq[Value]) pairs we can return the keys and values in separate data structures. For a given selector function f that computes a key from a value, the equivalent of (ks,vs)=v.groupBy(f).unzip is:

```
loop(v.size) ks=Bucket(Reduce(rhs)),vs=Bucket(Collect) { i =>
  ks ← (f(v(i)), v(i)); vs ← (f(v(i)), v(i)) }
```

This loop ranges over the size of v and produces two result collections ks, a flat sequence of keys, and vs, a nested collections of values that belong to the key in ks at the same index.

The fusion rules are summarized in Figure 6. Fusion of two loops is only permitted if there are no other dependencies between the loops, for example caused by side effects. Since we are working with a graph-based IR, dependency information is readily available. Vertical fusion, which absorbs a consumer loop into the producer loop, is only permitted if the consumer loop does not have any dependencies on its loop variable other than acessing the consumed collection at this index. In a data parallel setting, where loops are chunked, the producer may not be able to compute the exact index of an element in the collection it is building. Multiple instances of f1(i) are subject to CSE and not evaluated twice. Substituting x1(i2) with f1(i) will remove a reference to x1. If x1 is not used anywhere else, it will also be subject to DCE. Within fused loop bodies, unifying index variable i and substituting references will trigger the usual forward rewriting and simplification. Thus, fusion not only removes intermediate data structures but also provides additional optimization opportunities inside fused loop bodies (including fusion of nested loops).

```
def preferences(ratings: Rep[Matrix[Int]],
                sims: Rep[Matrix[Double]]) = {
  sims.mapRowsToVector { testProfile =>
    val num = sum(0, ratings.numRows) {
      i => testProfile(ratings(i,1))*ratings(i,2) }
    val den = sum(0, ratings.numRows) {
      i => abs(testProfile(ratings(i,1))) }
    num/(den+1)
}}
```

Figure 7. Snippet from collaborative filtering to be optimized

Fixed size array construction Array(a,b,c) can be expressed as

```
loop(3) x=Collect { case 0 => x ← a
                    case 1 => x ← b case 2 => x ← c }
```

and concatenation xs ++ ys as Array(xs,ys).flatMap(i=>i):

```
loop(2) x=Collect { case 0 => loop(xs.size) { i => x ← xs(i) }
                    case 1 => loop(ys.size) { i => x ← ys(i) }}
```

Fusing these patterns with a consumer will duplicate the consumer code into each match case. Therefore, implementations should impose cutoff mechanisms to prevent code explosion. Code generation does not need to emit actual loops for fixed array constructions but can just produce the right sequencing of yield operations.

5. Case Studies

We present several case studies to illuminate how our compiler architecture and in particular the staging aspect enable advanced optimizations related to data structures.

All experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67GHz processors and 96GB of RAM. Scala code was run on the Oracle Java SE Runtime Environment 1.7.0 and the Hotspot 64-bit server VM with default options. We ran each application ten times (to warm up the JIT) and report the average of the last 5 runs. For each run we timed the computational portion of the application.

5.1 Linear Algebra

We first consider two examples using an extended version of the staged linear algebra library presented in Section 1. In the first example, we use staged transformation, partial evaluation, rewrite rules, and fusion together to transparently optimize a logical operation on a sparse matrix into a much more efficient version that only operates on its non-zero values. In the second example, we show how staged transformations can elegantly specialize loops to different computing devices by selecting a dimension to unroll and expressing the unrolled loop as a simple while loop in the source language.

Sparse User-Defined Operators Figure 7 shows an excerpt from a collaborative filtering application. This snippet computes a user's

```
trait SparseMatrix[A] extends Matrix[A] with Struct {
  val numRows: Rep[Double],    val numCols:    Rep[Double],
  val _data:   Rep[Array[A]],  val _nnz:       Rep[Int],
  val _rowPtr: Rep[Array[Int]],val _colIndices:Rep[Array[Int]]
}
trait SparseVector[A] extends Vector[A] with Struct {
  val length:   Rep[Int],       val _data: Rep[Array[A]],
  val _indices: Rep[Array[Int]],val _nnz:  Rep[Int]
}
```

Figure 8. Data structure definition for sparse matrix and vector

preference for other users given the user's previous ratings and a pre-computed NxN similarity matrix. The code is written in a representation-agnostic way – it works whether `sims` is dense or sparse using generic operations. Our goal is to transform the function `mapRowsToVector` into an efficient sparse operation if `sims` is sparse and the user-provided function argument to `mapRowsToVector` returns 0 when the input is a vector (row) containing only 0s. By expressing this as a transformation (rather than during construction), the IR node for `mapRowsToVector` can take part in other analyses and rewritings prior to being lowered to operate on the underlying arrays.

Figure 8 defines new data structures for sparse matrices and vectors in the manner described in Section 4. Using these definitions, we define the transformed version of `mapRowsToVector` as follows:

```
def infix_mapRowsToVector[A,B](x: Rep[Matrix[A]],
                               f: Rep[Vector[A]] => Rep[B]) = {
  def isZeroFunc(f: Rep[Vector[A]] => Rep[B], len: Rep[Int]) =
    // use symbolic evaluation to test the user-defined
    // function f against a zero-valued argument
    f(Vector.zeros[A](len)) match {
      case Const(x) if (x == defaultValue[B]) => true
      case _ => false
    }
  if (x.isInstanceOf[SparseMatrix] && isZeroFunc(f,x.numRows))
    MatrixMapRowsToVec(x,f) atPhase(lowering) {
      // transform to operate only on non-zero values
      new SparseVector[B] {
        val length = x.numRows
        val _indices = asSparseMat(x).nzRowIndices // elided
        val _data = _indices.map(i=>f(x(i)))
        val _nnz = _indices.length
      }
    }
  else MatrixMapRowsToVec(x,f) // default
}
```

Staging is the critical ingredient that allows this transformation to be written expressively and correctly. We first use the programmatic struct support with inheritance from Section 4.1 to operate uniformly on dense and sparse matrices and to obtain access to the underlying representation in order to perform the transformation. If the argument is a sparse matrix, `mapRowsToVector` calls `isZeroFunc`, which symbolically evaluates the user-defined function `f` at staging time to discover if it actually has no effect on empty rows in the argument matrix. This discovery is in turn enabled by further rewrite rules that implement symbolic execution by optimizing operations for constant values. For example, we define `sum` as:

```
override def sum[A:Numeric](start: Exp[Int], end: Exp[Int],
                            block: Exp[Int] => Exp[A]) =
  block(fresh[Int]) match {
    case Const(x) if x == 0 => unit(0.asInstanceOf[A])
    case _ => super.sum(start,end,block)
  }
```

We are able to successfully perform this transformation of `mapRowsToVector` because LMS re-applies all rewrite rules as the

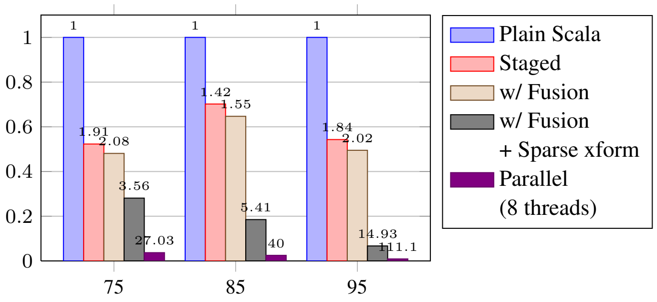

Figure 9. LinAlg Benchmark. The y-axis shows the normalized execution time of the collaborative filtering snippet on a 10kx10k matrix with various optimizations. The x-axis represents the percentage of sparse rows in the test data. Speedup numbers are reported at the top of each bar.

```
// x = input matrix, k = num clusters
// c = distance from each sample to nearest centroid
val newLocations = Matrix.fromArray {
  // returns an Array[Vector[Double]]
  Array.range(0,k) { j =>
    val (points,weightedpoints) = sum(0, x.numRows) {
      i => if (c(i) == j) (1,x(i))
    }
    val d = if (points == 0) 1 else points
    weightedpoints / d
}}
```

Figure 10. Snippet from k-means clustering to be optimized

transformed IR is constructed. After the transformation, the code is further optimized by fusing together the two `sum` statements in Figure 7 (but the fused summation is still only run on non-zero rows). Figure 9 shows the results of running the collaborative filtering snippet with input data of varying sparsity. As expected, the transformation provides greater benefit with increasing sparsity because we switched from a representation-agnostic implementation to a representation-sensitive one. We also implicitly parallelize the transformed map function, resulting in good speedup over the sequential baseline automatically.

Loop Parallelization for Heterogeneous Processors The final linear algebra example we will consider is the parallelization of a loop in the well-known k-means clustering algorithm. Figure 10 shows the snippet, which updates the cluster locations in the current iteration to the mean of all the samples currently assigned to that cluster.

K-means clustering is a statistical algorithm that is amenable to both multicore parallelization and GPU execution [28]. However, the `Array.range` statement in Figure 10 poses a challenge: each of the `k` cluster locations can be computed in parallel, or we can compute the inner `sum` statement in parallel. Since $k \ll$ `x.numRows` usually, on multicore machines with a smaller number of hardware threads (< 16) it is better to parallelize the outer (`fromArray`) dimension, but on massively parallel machines such as GPUs it is much better to parallelize the inner (`sum`) dimension.

If we internally represent parallel operators such as `Array.range` and `sum` as loops, a parallelization framework (such as Delite [5]) can automatically generate parallel and/or GPU code for them. Using the notation from Section 3.3, we can then define a device-specific transformation as follows:

```
trait LoopTransformerExp extends LinAlgExp { self =>
  val t = new LoopTransformer { val IR: self.type = self }
  def hasNestedLoop(l: Loop[_]): Boolean // elided
  def transformCollectToWhile(l: Collect[_]) = {
    // construct transformed representation
    var i = 0
    val size = t.transformExp(l.size)
    val res = alloc(size)
    while (i < size) {
      t.transformBlock(l.update)
        (res,i,t.transformBlock(l.func)(i))
      i += 1
    }
    res
}}
trait LoopTransformer extends ForwardTransformer {
  val IR: LoopTransformerExp
  import IR._
  override def transformStm(stm: Stm): Exp[Any] = stm match {
    case TP(s,l:Loop[_]) if (generateGPU && hasNestedLoop(l)) =>
      l.body match {
        case c:Collect[_] => transformCollectToWhile(c)
        case _ => super.transformStm(stm)
      }
    case _ => super.transformStm(stm)
}}
```

This transformation unwraps the `Array.range` function, which is internally represented by a loop, into an explicit **while** loop to expose any nested parallel operators. Although we elide it here, it is easy to add a heuristic to decide (statically if known or dynamically if not) whether or not to unwrap a loop based on its size. Note that due to staging, the transformed representation can be written as simple source code, which is easier to understand than direct IR manipulations. The transformation is only triggered if we are attempting to generate GPU code; if we are generating CPU code, we parallelize the outer loop to maximize our hardware utilization.

5.2 Regular Expression Matchers

Specializing string matchers and parsers is a popular benchmark in the partial evaluation literature [2, 10, 42]. Usually, specialization produces a set of mutually recursive functions, which is also our starting point. However this code does not achieve top performance compared to fast automata libraries. We show that treating generated functions as data objects and transforming them into a more efficient form yields performance that exceeds that of an optimized library.

We consider “multi-threaded” regular expression matchers, that spawn a new conceptual thread to process alternatives in parallel. Of course, these matchers do not actually spawn OS-level threads, but rather need to be advanced manually by client code. Thus, they are similar to coroutines. Here is an example for the fixed regular expression .*AAB:

```
def findAAB(): NIO = {
  guard(C('A')) {
    guard(C('A')) {
      guard(C('B'), true) {
        stop()
  }}} ++
  guard(W) { findAAB() } // in parallel ...
}
```

The first argument to `guard` is a character classs, `C(.)` defines a singleton character class and `W` denotes the wildcard. We added combinators on top of the core abstractions that can produce matchers from more conventional regular expressions: `many(seq)(star(W), C('A'), C('A'), C('B'))` for the example above. We can easily add a parser for textual regular expressions on top of these combinators.

NFA to DFA Conversion Internally, the given matcher uses an API that models nondeterministic finite automata (NFA):

```
def exploreNFA[A:Manifest](xs: NIO, cin: Rep[Char])(
     k: (Boolean, NIO) => Rep[A]): Rep[A] = xs match {
  case Nil => k(false, Nil)
  case NTrans(W, e, s)::rest =>
    val (xs1, xs2) = xs.partition(_.c != W)
    exploreNFA(xs1,cin)(continueWith(k, x2))
  case NTrans(cset, e, s)::rest =>
    if (cset contains cin) {
      val xs1 = // restrict rest: knowing cset
      exploreNFA(xs1,cin)((flag,acc) => k(
        flag || e(), acc ++ s()))
    } else {
      val xs1 = // restrict rest: knowing_not cset
      exploreNFA(xs1, cin)(k)
    }
}
```

Figure 11. NFA Exploration

```
type NIO = List[NTrans]
case class NTrans(c: CharSet, x: () => Boolean, s: () => NIO)
def guard(c: CharSet, x: => Boolean = false)(s: => NIO): NIO = {
  List(NTrans(c, () => x, () => s))
}
def stop(): NIO = Nil
```

An NFA state consists of a list of possible transitions. Each transition may be guarded by a set and it may have a flag to be signaled if the transition is taken. It also knows how to compute the following state. For simplicity, we use set of characters for the guard and a boolean for the flag, but of course, we could use generic types as well. Note that the API does not mention where input is obtained from (files, streams, etc.).

We will translate NFAs to DFAs using staging. The `Automaton` class is part of the unstaged DFA API. The staged API is just a thin wrapper.

```
case class Automaton[I, O](out: O, next: I => Automaton[I,O])
type DfaState = Automaton[Char,Boolean]
type DIO = Rep[DfaState]
def dfa_trans(e: Boolean)(f: Rep[Char] => DIO): DIO
```

Translating an NFA to a DFA is accomplished by creating a DFA state for each encountered NFA configuration:

```
def convertNFAtoDFA(flag: Boolean, state: NIO): DIO = {
  val cstate = canonicalize(state)
  dfa_trans(flag) { c: Rep[Char] => exploreNFA(cstate, c) {
    convertNFAtoDFA
  }
}}
convertNFAtoDFA(false, findAAB())
```

The LMS framework memoizes functions which, with the state canonicalization (e.g. we need to remove duplicates), ensures termination if the NFA is in fact finite. Indeed, our approach is conceptually similar to generating a DFA using derivatives of regular expressions [6, 35]: both approaches rely on identifying (approximately) equivalent automaton states; furthermore, computing a symbolic derivative is similar to advancing the NFA by a symbolic character, as explained next.

We use a separate function to explore the NFA space (see Figure 11), advancing the automaton by a symbolic character `cin` to invoke its continuation `k` with a new automaton, i.e. the possible set of states after consuming `cin`. The given implementation needs only to treat the wildcard character set (denoted by `W`) specially. The other character sets (single character or range) are treated generically, and it would be straightforward to add more cases without changing this function. The algorithm tries to remove as many redundant checks and impossible branches as possible (it relies on

```
def naiveFindAAB():
Automaton[Char, Boolean] = {
  val x1 = {x2: (Char) =>
    /*matched nothing*/
    val x3 = x2 == 'A'
    val x18 = if (x3) {
      x17
    } else {
      x13
    }

    x18
  }

  val x12 = Automaton(true,x1)
  val x7 = {x8: (Char) =>
    /*matched AA*/
    /* ... */
  }
  val x10 = Automaton(false,x7)
  val x4 = {x5: (Char) =>
    /*matched A*/
    /* ... */
  }
  val x17 = Automaton(false,x4)
  val x13 = Automaton(false,x1)
  x13
}
```

```
def optFindAAB(input: String):
Boolean = {
  val n = input.length
  if (n==0) return false
  var id = 0
  var i = 0
  val n_dec = n-1
  while (i < n_dec) {
    val x18 = input.charAt(i)
     // first case analysis
     // for next state id
     id = id match {
      case 0 =>
       /*matched nothing*/
        val x26 = x18 == 'A'
        val x27 = if (x26) 2
        else 0
        x27
      case 1 =>
        /*matched AA ... */
      case 2 =>
        /*matched A ...*/
    }
    i += 1
  }
  val x18 = input.charAt(i)
  // second case analysis
  // for final boolean flag
  id == 1 && x18 == 'B'
}
```

Figure 12. Generated matcher code for regular expression .*AAB. Optimized code on the right.

the knowing and knowing_not binary functions on character sets). This only works because the guards are staging-time values.

The generated code is shown in Figure 12 on the left. Each function corresponds to one DFA state.

One straightforward optimization is to signal early stopping if the boolean flag remains the same in all possible next states. To do so, we change the DfaState so that the output can encode two boolean values: the original flag and whether we can stop early. This does not help for matching .*AAB, but it does for .*AAB.*, as we can stop as soon as we find an occurrence of AAB.

Transforming Closures Our final implementation generates a complete matcher from a String input to a Boolean output. The generated code does not have functions nor Automaton states. Instead, we generate a case statement for each function. We do this twice: in the middle of matching the input, the value of the case analysis is the next functional state, while for the last character matching, its value is the boolean flag output. Because all the cases are inlined, we achieve early stopping by simply returning from the function with the boolean flag output in the midst of the first case analysis. The optimized code is shown in Figure 12 on the right.

Figure 13 shows benchmarking results. We compare our "naive" implementation (Staged 1), which includes the early stopping optimization, to the JDK and dk.brics.automaton [31] libraries. On average, our "naive" implementation is more than 2.5x faster than JDK's and about 3x slower than dk.brics.automaton. In order to speed up our implementation, we experimented with various staged transformations. Our final implementation (Staged 2) beats dk.brics.automaton by nearly a factor of 2 and JDK by a factor of 15. For comparison, we also tried a completely unstaged plain Scala version, where we directly evaluate the code instead of staging it, and it is 140x slower than JDK.

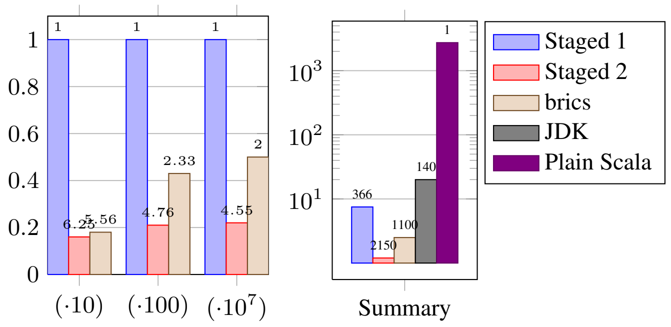

Figure 13. Regexp Benchmark. The first graph shows the relative execution time of matching an 10, 100, or 10^7 long input string of the form A+B on the regular expression .*AAB. The second graph summarizes the relative performance over many different inputs and regular expressions. Speedups are reported on top of each bar.

```
// lineItems: Array[LineItem]
val q = lineItems filter (_.l_shipdate <= Date(''1998-12-01'')).
groupBy (_.l_linestatus) map { case (key,g) => new {
  val lineStatus = key
  val sumQty = g.map(_.l_quantity).sum
  val sumDiscountedPrice =
    g.map(l => l.l_extendedprice*(1.0-l.l_discount)).sum
  val avgPrice = g.map(_.l_extendedprice).sum / g.size
  val countOrder = g.size
}} sortBy(_.lineStatus)
```

Figure 14. Sample query (similar to Q1 of the TPC-H benchmark)

5.3 Collection and Query Operations

Next we consider the case of executing queries on in-memory collections. Figure 14 shows a simplified version (for illustration purposes) of TPC-H Query 1 written using the standard Scala collections API. A straightforward execution of this query as written will be substantially slower than what could be achieved with an optimized low-level imperative implementation. In particular executing this code in Scala will perform a new array allocation for the output of each collection operation. Furthermore, every element of the arrays will be a heap allocated object since both LineItem and the anonymous class for the result are implemented as JVM classes. These classes however contain only fields, no methods, and as such we can apply struct optimizations to them during staging. Since the fields of these structs can all be represented as JVM primitive types, we can eliminate all of the heap allocated objects at runtime by performing array-of-struct (AoS) to struct-of-array (SoA) transformations for each array. Furthermore since LineItem as defined by TPC-H contains 16 fields, while Query 1 only accesses 7 of those fields, the DCE engine eliminates the arrays of the other 9 fields all together once the SoA form is created.

Fusion Optimization In addition, we eliminate the intermediate array and map data structures produced by each of the collection operations by fusing operations (Section 4.4) at two levels. At the inner level, creating the output struct requires multiple reduction operations. Horizontal fusion combines the reductions for each field into a single loop. On the outer level, the query groups the collection according to a key function and then maps each group (sub-collection) down to a single element. We fuse the groupBy and map together by using the rule for consuming a bucket-collect operation. Essentially the fusing algorithm observes that the output element of each group can be computed directly by reducing the entire input collection into the buckets defined by the key function of the groupBy. Furthermore, the vertical consume-collect rule fuses the filter with the groupBy by predicating the addition of each element

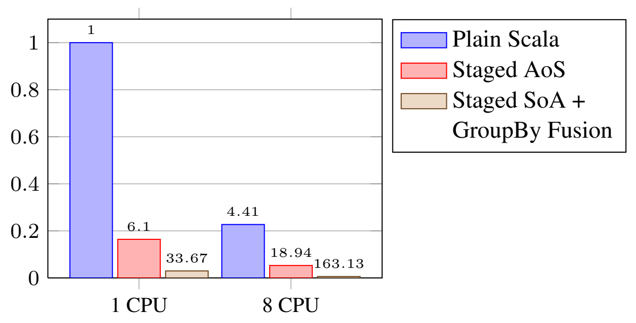

Figure 15. Performance of TPC-H Query 1 with staging optimizations. The y-axis shows the normalized execution time for 30 million records with speedups reported at the top of each bar.

to the appropriate group (and after groupBy-map fusion, predicating the reduction of each element into the appropriate group). Finally, horizontal fusion combines the operations for each field (array in the SoA form) into a single loop. Therefore, the program produces the output collection directly from the input with a single loop and no intermediate arrays. Sorting the output yields the final result.

We perform one final optimization during code generation of the fused loop. In order to implement `Bucket(Reduce)` operations we internally allocate a generic HashMap instance to manage the buckets. However, when n `Bucket(Reduce)` operations with the same key function appear in the same loop, we do not allocate a HashMap for each operation. Instead we use a single HashMap to determine an array index for each key, and then use that index to access the n arrays allocated to hold the result of each operation, eliminating the overhead of computing the same bucket location multiple times.

Performance Results We evaluate these optimizations in Figure 15. The execution times are normalized to the plain Scala library implementation of the query (leftmost bar). The middle bar shows the benefits of staging but without applying SoA transformations or groupBy fusion. Here staging is only applying basic optimizations such as common subexpression elimination and code motion. In particular the staged version contains the following key improvements. Higher-order abstractions such as anonymous functions have been eliminated, construction of the `Date` object within the filter has been lifted out of the loop and converted into an `Int`, pairs of primitive types have been packed into a single primitive type with twice the bits (e.g., a `Pair[Char,Char]` is encoded as an `Int`), and the map-reduce chains in the inner loop have been fused. Since we do not perform SoA transformations in this version, the `LineItem` struct is generated as a custom JVM class and the unused fields remain in the generated code. These optimizations provide approximately 6x speedup over the library implementation.

The rightmost bar shows the results of adding AoS to SoA transformations and all of the fusion optimizations described above, namely fusing the filter, groupBy, and map operations into a single loop. These optimizations provide approximately 5.5x speedup over the basic staging optimizations and 34x speedup over the library implementation. In addition, this version scales better when parallelized across multiple cores as it parallelizes over the size of the input collection rather than over the number of groups. Ultimately, with staging optimizations and parallel code generation, we were able to attain 163x speedup with 8 cores over a single-threaded library implementation.

Heterogeneous Targets AoS to SoA conversion greatly improves the ability to generate code for GPUs. In this example, however, the data transfer cost overwhelms any speedup for end-to-end runs. Another interesting target besides multi-core and GPU devices are "Big Data" cluster frameworks such as Hadoop or Spark. We can use the same collections API to target several different frameworks by adding appropriate framework-specific lowerings. A recent paper [1] contains more details and demonstrates good speedups.

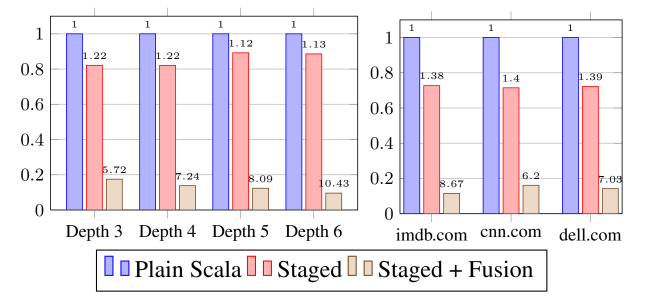

Figure 16. Synthetic file system to HTML (left, x-axis represents the template nesting depth. Depths 1 and 2 are omitted due to very short running times). Commercial websites (right). The y-axis shows normalized running time compared to plain Scala, speedup numbers are shown at the top of each bar.

5.4 String Templates

String template libraries are in wide use today to transform structured data into a flat string representation (e.g. XML, HTML, or JSON). While the semantics are simple and easy to express in a functional way, template engine implementations are often intricate because data copying and intermediate data structures need to be avoided to obtain good performance. We present the core of a template engine based on purely functional collection operations (list construction, concatenation and comprehensions), with loop fusion (Section 4.4) taking care of removing intermediate data structures.

A simple HTML menu can be rendered like this:

```
def link(uri: Rep[String], name: Rep[String]) =
  List("<a href='", uri, "'>", name, "</a")
def renderItem(i: Rep[Item]) =   List("<li><ul>") ++
  i.subitems.flatMap(link(i.name, i.link)) ++ List("</ul></li>")
def usersTable(items: Rep[List[Item]]) = List("<ul>") ++
  items.flatMap(x => renderItem(x)) ++ List("</ul>")
```

When executing this code without fusion optimization, each `flatMap` operation traverses the lists produced by the nested template one more time. Each concat (++) operation incurs an additional traversal. This results in performance that, for fixed output size, decreases linearly with the level of template nesting. The asymptotic worst case complexity is thus quadratic in the output size, whereas an imperative implementation would be linear.

Since all function calls are unfolded at staging time, our fusion transformation (Section 4.4) is able to remove the unnecessary traversals and data structures, provided that only core collection operations are used to express templates. The generated code traverses the input data structure just once, emitting strings directly into a single output buffer.

Benchmark results are reported in Figure 16. We demonstrate the ability of obtaining asymptotic speedups on a synthetic benchmark that generates HTML representations of a file system structure with growing levels of nesting: with fusion enabled, speedups grow with the depth of nesting. Furthermore, we implemented mock templates to resemble the structure of three public web sites, yielding speedups of up to 8.6x over the plain, unstaged Scala implementation.

6. Discussion

As shown in Section 5, our system is able to achieve order of magnitude (and in one case even asymptotic) speedups on a varied range of non-trivial high level programs. While we are not aware of any other system that demonstrably achieves comparable results on a similar range of programs, these results bear the question whether they could be achieved with similar effort using other means.

As suggested by one of the reviewers, a simpler design would be to perform optimizations dynamically. For example, one could implement vectors and matrices as a hierarchy of classes, with a special class for zero vectors, instead of distinguishing zero vectors statically in the IR. This is certainly a valid design, but it comes with a number of drawbacks. First, runtime distinctions introduce interpretive overhead, indirection and virtual calls, which also make it harder for the compiler to generate efficient code. Second, while dynamic information may be more precise than static information at some times, the possibilities for dynamic optimizations are very limited in other key aspects: For example, it is not clear how to integrate DCE or any other optimization that requires non-local knowledge (liveness, horizontal fusion).

Another question posed by a reviewer was whether staging really simplifies transformation compared to state-of-the art rewriting systems such as those included in language workbenches like Spoofax [24] or JetBrains MPS [19]. While these systems are impressive, we believe they serve a different purpose and that the capability to programmatically remove abstraction at all intermediate levels, with strong guarantees about the residual code, is hard to overstate in the context of program optimization. For example, the code in Figure 3 transforms symbolic linear algebra operations to the same zero-overhead traversals shown in Figure 2, with higher order functions and type class instances removed by staging, without dedicated compiler analysis. Staging enables us to express a program (or model) transformer as an interpreter; this is an advantage because writing an interpreter in an expressive language is arguably simpler than implementing a transformer, even with good toolkit support. A key aspect is linguistic reuse of abstractions of the meta language. For example, our recent work on JavaScript generation using LMS [27] re-uses Scala's CPS transform [40] at staging-time to generate asynchronous code that is in continuation passing style (CPS). By contrast, WebDSL [17], which is implemented in Spoofax, has to implement the CPS transform afresh, for each language construct of the object language.

Working with a graph-based IR also makes a qualitative difference compared to expression trees. Most importantly, it becomes much easier to implement transforms that require dependency information. Our fusion algorithm (Section 4.4), for example, does not require loops to be syntactically close or adjacent to be fused:

```
def calcSum() = array.sum
def calcCount() = array.filter(_ > 0).count
println("sum: " + calcSum())
println("avg: " + (calcSum() / calcCount()))
```

Staging will inline the `calcSum` and `calcCount` calls and the fusion algorithm will combine all traversals into a single loop. Any algorithm that only considers dependent traversal expressions will miss the horizontal fusion opportunities across the two `println` statements. This is commonly the case for pure front-end approaches based e.g. on C++ expression templates.

Another key aspect of our design is the different treatment of lowering transforms (handled in separate passes one after the other) and optimizations (combined into joint simplification passes using speculative rewriting). We believe this distinction is essential to make optimizations reliable by avoiding phase ordering problems and making sure that high-level optimizations are applied exhaustively before lowerings take place.

Finally, there is the question of programmer effort required to achieve similar results. From our perspective, the most important aspects are that the optimizations we present can be applied gradually, and that they are reusable for all clients of an optimized library. We expect that end user programmers will not usually write any transformations on their own, but they can add rewrites if they need. Library developers who wish to add optimizations need to implement functionality corresponding to the code we show (e.g. in Figure 3) but they can do so in a gradual way: Start with code similar to Figure 1, add staging (Figure 2), and, if that is not sufficient, add further optimizations (Figure 3). Delayed rewriting (Section 3.3) can further reduce boilerplate. The guarantees provided by the type system (non-`Rep[T]` expressions evaluated at staging time) and smart constructors (no IR node constructed if rewrites match) help ensure that optimizations take place in the intended way. As a debugging aid, transformed code can be emitted at any time.

6.1 Related Work

Extensible compilers have been studied for a long time, recent examples in the Java world are Polyglot [33] and JastAdd [13]. The Glasgow Haskell Compiler also allows custom rewrite rules [22]. There are also elaborate approaches to library specific optimizations inspired by formal methods. ROSE [36] is a framework for building source-to-source translators that derives automated optimizations from semantics annotations on library abstractions. COBALT [30] is a domain-specific language for implementing optimizations that are amenable to automated correctness reasoning. Tate et al. [48] present a method of generating compiler optimizations automatically from program examples before and after a transformation.

Delite [5, 28, 41] is a parallelization framework for DSLs developed on top of LMS. Previously, Delite had a notion of a multi-level IR: some IR nodes could be viewed from several different angles at the same time (say, as a parallel loop and as a matrix operation). This caused problems because "earlier", more high-level views had to be carried along and obsolete or fully exploited information was never discarded, limiting the choices of the compiler. We have used the techniques presented in this work to extend Delite and to implement domain-specific optimizations (similar to those presented in the case studies) for existing Delite DSLs such as OptiML [44]. Delite was also used as the parallel execution engine for the case studies in this paper. Compared to Delite, this paper presents a general extensible compiler architecture, not limited to domain specific languages. Previous publications on LMS [38, 39] presented only the pure front-end, single pass instantiation. The use of `Rep[T]` types to define binding times in LMS is inspired by earlier work on finally tagless or polymorphic language embedding [8, 18].

Tobin-Hochstadt et al. [50] propose languages as libraries in Racket and make pervasive use of macros for translation. Our approach of using staging is similar in spirit. Earlier work on realistic compilation by program transformation in the context of Scheme was presented by Kelsey and Hudak [25].

The ability to compile high-level languages to lower-level programming models has been investigated in several contexts. Elliot et al. [14] pioneered embedded compilation and used a simple image synthesis DSL as an example. Telescoping languages [26] is a strategy to automatically generate optimized domain-specific libraries. In C++, expression templates [51] are popular to implement limited forms of domain specific optimizations [23, 52]. The granularity is restricted to parts of a larger template expression.

Nystrom et al. [34] shows a library based approach to translating Scala programs to OpenCL code. This is largely achieved through Java bytecode translation. A similar approach is used by Lime [3] to compile high-level Java code to a hardware description language such as Verilog.

Our compiler infrastructure implements a set of advanced optimizations in a reusable fashion. Related work includes program transformations using advanced rewrite rules [4], combining analyses and optimizations [9, 29, 53] as well as techniques for eliminating intermediate results [12, 54] and loop fusion [16]. An interesting alternative to speculative rewriting is equality saturation [47], which encodes many ways to express an operation in the IR.

As an example of using partial evaluation in program transformation, Sperber and Thiemann [43] perform closure conversion and tail call introduction by applying offline partial evaluating to a suitable interpreter. More recently, Cook et al. [11] perform model transformation by partial evaluation of model interpreters. Partial evaluation corresponds to constant folding and specialization whereas our approach allows arbitrary compiler optimizations, arbitrary computation at staging/specialization time to remove abstraction overhead and provides strong guarantees about what becomes part of the residual code (type `Rep[T]` residual, vs `T` static).

6.2 Conclusion

We have demonstrated a compiler architecture that achieves order of magnitude speedups on high-level programs by fusing collection operations, changing data layout and applying further optimizations on high-level objects, enabled by intermediate languages with staging and a facility to combine optimizations without phase ordering problems. Our system combines several novel pieces with existing techniques, which together provide optimization power that is greater than the sum of the parts.

Acknowledgments

We thank the POPL reviewers for their valuable feedback and suggestions. This research was sponsored by the European Reserach Council (ERC) under grant 587327 "DOPPLER", DARPA through Oracle order US103282, DARPA contract HR0011-11-C-0007 "SEEC", NSF grant SHF CCF-1111943, the Stanford Pervasive Parallelism Lab affiliates program (supported by Oracle, AMD, Intel, NVIDIA) and additional support from Oracle.

References

[1] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded dsl for high performance big data processing. BigData, 2012. `http://infoscience.epfl.ch/record/181673/files/paper.pdf`.

[2] M. S. Ager, O. Danvy, and H. K. Rohde. Fast partial evaluation of pattern matching in strings. *ACM Trans. Program. Lang. Syst.*, 28(4): 696–714, 2006.

[3] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. OOPSLA, 2010.

[4] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundam. Inf.*, 69: 123–178, July 2005.

[5] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[6] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4): 481–494, 1964.

[7] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *GPCE*, 2003.

[8] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[9] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

[10] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Inf. Process. Lett.*, 30(2):79–86, 1989.

[11] W. R. Cook, B. Delaware, T. Finsterbusch, A. Ibrahim, and B. Wiedermann. Model transformation by partial evaluation of model interpreters. Technical Report TR-09-09, UT Austin Department of Computer Science, 2008.

[12] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, 2007.

[13] T. Ekman and G. Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[14] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–26. Springer Berlin / Heidelberg, 2000.

[15] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12 (4):381–391, 1999.

[16] C. Grelck, K. Hinckfuß, and S.-B. Scholz. With-loop fusion for data locality and parallelism. IFL, 2006.

[17] D. M. Groenewegen, Z. Hemel, L. C. L. Kats, and E. Visser. WebDSL: a domain-specific language for dynamic web applications. In *OOPSLA Companion*, 2008.

[18] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. GPCE, 2008.

[19] JetBrains. Meta Programming System, 2009. URL `http://www.jetbrains.com/mps/`.

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[21] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*, 2008.

[22] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. Haskell, 2001.

[23] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. J. Williams. Array design and expression evaluation in pooma ii. In *ISCOPE*, 1998.

[24] L. C. L. Kats and E. Visser. The Spoofax language workbench. rules for declarative specification of languages and IDEs. In *SPLASH/OOPSLA Companion*, 2010.

[25] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *POPL*, 1989.

[26] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

[27] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. Javascript as an embedded dsl. In *ECOOP*, 2012.

[28] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[29] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002.

[30] S. Lerner, T. D. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.

[31] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[32] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. PEPM, 2012.

[33] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, 2003.

[34] N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for gpus in scala. GPCE, 2011.

[35] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives reexamined. *J. Funct. Program.*, 19(2):173–190, Mar. 2009.

[36] D. J. Quinlan, M. Schordan, Q. Yi, and A. Sæbjørnsen. Classification and utilization of abstractions for optimization. In *ISoLA (Preliminary proceedings)*, 2004.

[37] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[38] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. GPCE, 2010.

[39] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[40] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *ICFP*, 2009.

[41] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. DSL, 2011.

[42] A. Shali and W. R. Cook. Hybrid partial evaluation. OOPSLA, 2011.

[43] M. Sperber and P. Thiemann. Realistic compilation by partial evaluation. In *PLDI*, 1996.

[44] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. ICML, 2011.

[45] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2-3): 101–142, 2001.

[46] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[47] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, 2009.

[48] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In *POPL*, 2010.

[49] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state. Technical report, 1999. URL `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz`.

[50] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. PLDI '11, 2011.

[51] T. L. Veldhuizen. Expression templates, C++ gems. SIGS Publications, Inc., New York, NY, 1996.

[52] T. L. Veldhuizen. Arrays in blitz++. In *ISCOPE*, 1998.

[53] T. L. Veldhuizen and J. G. Siek. Combining optimizations, combining theories. Technical report, Indiana University, 2008.

[54] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[55] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989.