Language Virtualization for Heterogeneous Parallel Computing

Hassan Chafi* Zach DeVito* Adriaan Moors† Tiark Rompf† Arvind Sujeeth*
Pat Hanrahan* Martin Odersky† Kunle Olukotun*

*Stanford University: {hchafi, zdevito, asujeeth, hanrahan, kunle}@stanford.edu
†EPFL: {adriaan.moors, tiark.rompf, martin.odersky}@epfl.ch

EPFL-REPORT-148814

Abstract

As heterogeneous parallel systems become dominant, application developers are being forced to turn to an incompatible mix of low level programming models (e.g. OpenMP, MPI, CUDA, OpenCL). However, these models do little to shield developers from the difficult problems of parallelization, data decomposition and machine-specific details. Ordinary programmers are having a difficult time using these programming models effectively. To provide a programming model that addresses the productivity and performance requirements for the average programmer, we explore a domain-specific approach to heterogeneous parallel programming.

We propose language virtualization as a new principle that enables the construction of highly efficient parallel domain specific languages that are embedded in a common host language. We define criteria for language virtualization and present techniques to achieve them. We present two concrete case studies of domain-specific languages that are implemented using our virtualization approach.

1. Introduction

Until the early 2000s, advances in out-of-order (OOO) superscalar processors provided applications with improved performance by increasing the CPU core clock frequency and the number of instructions per clock cycle (IPC). With each new generation of processors, software developers were able to leverage this performance increase to create more compelling applications without changing their programming model. Furthermore, existing applications also benefited from this performance increase with no extra effort. The inability of processor vendors to deliver higher performance from a single core without exceeding a reasonable power envelope has brought this so called "free lunch" era to an end [45].

Power efficiency is now the most dominant design driver for microprocessors. Power efficient microprocessor design favors chip-multiprocessors consisting of simpler cores [23, 31] and heterogeneous systems consisting of general purpose processors, SIMD units and special purpose accelerator devices such as graphics processing units (GPUs) [2,42] and cryptographic units. Existing applications can no longer take advantage of the additional compute power available in these new and emerging systems without a significant parallel programming and program specialization effort. However, writing parallel programs is not straightforward because in contrast to the familiar and standard *von Neumann* model for sequential programming, a variety of incompatible parallel programming models are required, each with their own set of trade-offs. The situation is even worse for programming heterogeneous systems where each accelerator vendor usually provides a distinct driver API and programming model to interface with the device. Writing an application that directly targets the variety of existing systems, let alone emerging ones, becomes a very complicated affair.

While one can hope that a few parallel programming experts will be able to tackle the complexity of developing parallel heterogeneous software, expecting the average programmer to deal with all this complexity is simply not realistic. Moreover, exposing the programmer directly to the various programming models supported by each compute device will impact application portability as well as forward parallel scalability; as new computer platforms emerge, applications will constantly need to be rewritten to take advantage of any new capabilities and increased parallelism. Furthermore, the most efficient mapping of an application to a heterogeneous parallel architecture occurs when the characteristics of the application are matched to the different capabilities of the ar-

chitecture. This represents a significant disadvantage of this approach: for each application and for each computing platform a specialized mapping solution must be created by a programmer that is an expert in the specific domain as well as in the targeted parallel architecture. This creates an explosion and fragmentation of mapping solutions and makes it difficult to reuse good mapping solutions created by experts.

1.1 The Need for DSLs

One way to capture application-specific knowledge for a whole class of applications and simplify application development at the same time is to use a domain specific language (DSL). A DSL is a concise programming language with a syntax that is designed to naturally express the semantics of a narrow problem domain [47]. DSLs, sometimes called "little languages" [4] or "telescoping languages" [28], have been in use for quite some time. In fact, it is likely that most application developers have already used at least one DSL. Examples of commonly used DSLs are TeX and LaTeX for typesetting academic papers, Matlab for testing numerical linear algebra algorithms, and SQL for querying relational databases. One can also view OpenGL as a DSL. By exposing an interface for specifying polygons and the rules to shade them, OpenGL created a high-level programming model for real-time graphics decoupled from the hardware or software used to render it, allowing for aggressive performance gains as graphics hardware evolves. Even the Unix shell can be considered a DSL that provides a command-line interface to underlying operating system functions such as file and process management. The use of DSLs can provide significant gains in the productivity and creativity of application developers, the portability of applications, and application performance.

The key benefits of using a domain-specific approach for enhancing application performance are using domain knowledge for static and dynamic optimizations of a program written using a DSL and the ability to reuse expert knowledge for mapping applications efficiently to a specialized architecture. Most domain specific optimizations would not be possible if the program was written in a general purpose language. General-purpose languages are limited when it comes to optimization for at least two reasons. First, they must produce correct code across a very wide range of applications. This makes it difficult to apply aggressive optimizations– compiler developers must err on the side of correctness. Second, because of the general-purpose nature needed to support a wide range of applications (e.g. financial, gaming, image processing, etc.), compilers can usually infer little about the structure of the data or the nature of the algorithms the code is using. In contrast, DSL compilers can use aggressive optimization techniques using knowledge of the data structures and algorithms derived from the DSL. This makes it possible to deliver good performance on heterogeneous architectures using DSLs.

Using DSLs naturally divides the application development process into two phases. First, a DSL developer designs a DSL for a specific domain. Then, a much larger number of domain experts make use of the DSL to develop applications. The ideal DSL developer would be a domain expert, a parallelism expert, and a language and compiler expert. Such developers are rare and so there is a need to simplify the process of developing DSLs for parallelism.

Traditionally, DSLs have been developed from the ground-up using custom compiler infrastructure. This is called the "external DSL approach". There are two problems with this approach to DSLs. First, developing these new languages to a sufficient degree of maturity is an enormous effort. This investment would have to include not just language specifications and construction of their optimizing compilers and libraries, but also all the other aspects of modern tooling including IDEs, debuggers, profilers, build tools as well as documentation and training. It is hard to see how such an investment can be made repeatedly for each specialized domain. Second, DSLs do not exist in a vacuum but have to interface to other parts of a system. For instance, a climate simulation program could have a visualization component that is based on a graphics DSL. It is not clear how multiple separate DSLs would inter-operate without creating a new DSL that integrates the desired combination of the others.

1.2 Embedded DSLs and Language Virtualization

Embedded DSLs [22] overcome the problems with external DSLs and make DSL development more tractable. An embedded DSL lives inside of a host language. It is quite like a framework or a library, consisting of a set of classes and operations on objects of those types. There are several advantages to using embedded DSLs for application writers. First, programmers do not have to learn a completely new syntax. Second, multiple DSLs can be combined in the same application. Third, the DSL infrastructure can be shared between different DSLs and DSL developers will all use the same infrastructure for building their DSLs.

The main problem with embedded DSLs is that they cannot exploit domain knowledge to efficiently map programs to specialized architectures because they live inside a general purpose language. There is no obvious solution to this problem apart from replicating much of the effort of building a stand-alone DSL implementation. To overcome this problem, we need embedding languages that are particularly suited to the task of serving as a host language to DSL implementations. A language with this capability supports what we call *language virtualization*. A language is *virtualizable* if and only if it can provide an environment to embedded languages that makes them essentially identical to corresponding stand-alone language implementations in terms of
(1) *expressiveness* — being able to express a DSL in a way which is natural to domain specialists,
(2) *performance* — leveraging domain knowledge to produce optimal code, and

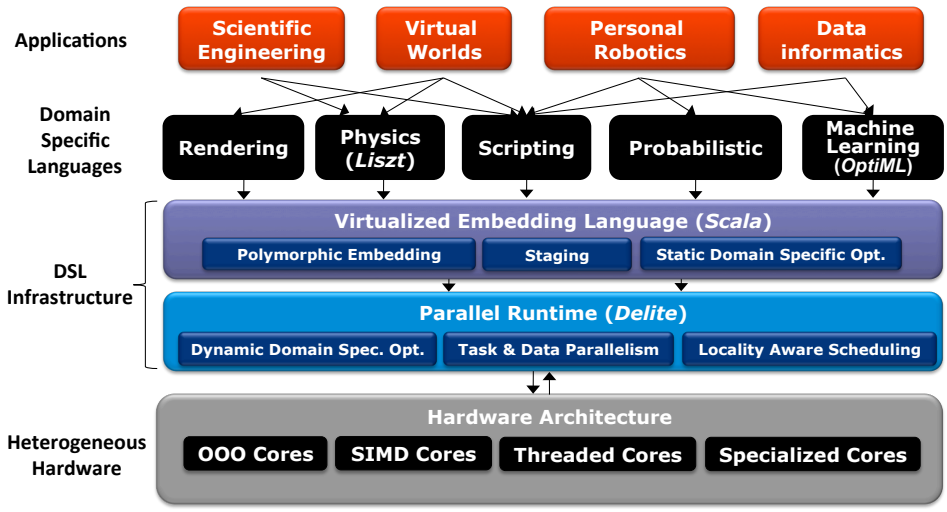

Figure 1. An environment for domain-specific programming of heterogeneous parallel architectures using language virtualization.

(3) *safety* — domain programs are guaranteed to have certain properties implied by the DSL,
(4) while at the same time requiring only modestly more *effort* than implementing a simple embedding. A subset of these features has been achieved before — most notably by Lisp, as much as 50 years ago. However, we believe we are the first to provide all of them. Section 6 provides a detailed comparison with related work. We discuss the means to achieve all of these features at once in more detail in Section 2.

There is a close analogy between language virtualization and hardware virtualization using virtual machines. In data-centers, it is often desirable to have a range of differently configured machines at one's disposal (for provisioning, fault-tolerance, and isolation), but usually it is not feasible or even desirable to operate a corresponding number of physical machines. Hardware virtualization solves this problem by embedding a number of specific virtual machines on a general-purpose host machine. A key aspect of virtual hardware resources is that they are practically indistinguishable from their real counterparts. We believe the same should be true for an embedded DSL, in the sense that it should exhibit the same expressiveness, performance and safety as if a specialized language tool chain had been tailor-made for the particular DSL.

This paper describes key elements of an ongoing effort to virtualize the language Scala [1] and how language virtualization can be used in a domain-specific programming environment for heterogeneous parallel computers. The components of this environment are shown in Fig. 1. The environment is composed of four main components: Applications composed of multiple DSLs, DSLs (e.g. Liszt and OptiML) embedded in Scala using language virtualization, a Scala-based compiler infrastructure that can perform domain-specific optimizations and a framework and runtime for DSL parallelization and mapping to heterogeneous architectures.

The remainder of this paper is organized as follows. Section 2 explains the notion of language virtualization in more detail and discusses key elements of virtualizing Scala. The next two sections describe how language virtualization can be used to develop two very different DSLs. Section 3 introduces Liszt, a DSL for scientific simulation that statically generates parallel code in C++. Section 4 introduces OptiML, a DSL for machine learning and data analysis. Section 5 describes Delite, a framework that simplifies DSL parallelization. Section 6 presents the related work for parallel programming, DSLs and language virtualization. Section 7 concludes.

2. Language Virtualization

We propose the following definition of language virtualization to capture necessary conditions for a general purpose language to serve as a successful embedding environment for DSLs:

Definition. A programming language is *virtualizable* with respect to a class of embedded languages if and only if it can provide an environment to these embedded languages that makes the embedded implementations essentially identical to corresponding stand-alone language implementations in terms of *expressiveness*, *performance* and *safety*—with only modestly more *effort* than implementing the simplest possible complete embeddings.

Expressiveness encompasses syntax, semantics and, in the case of domain-specific languages, general ease of use for domain experts. Just as virtual hardware resources are not exactly identical to real ones, we do not require that an

embedded language can exactly model the syntax of a stand-alone language but settle for a syntax that is essentially the same, i.e. modulo syntactic sugar. The same consideration applies to the other criteria as well.

Performance implies that programs in the embedded language must be amenable to extensive static and dynamic analysis, optimization, and code generation, just as programs in a stand-alone implementation would be. For many embedded languages, in particular those that are the focus of this paper, this rules out any purely interpretation-based solutions.

Safety means that the embedded implementation is not allowed to loosen guarantees about program behavior. In particular, host-language operations that are not part of the embedded language's specification must not be available to embedded programs.

Modest *effort* is the only criterion that has no counterpart in hardware virtualization. However, it serves an important purpose since an embedded language implementation that takes a DSL program as a string and feeds it into an external, specialized stand-alone compiler would trivially satisfy criteria *expressiveness*, *performance* and *safety*. Building this implementation, however, would include the effort of implementing the external compiler, which in turn would negate any benefit of the embedding. In a strict sense, one can argue that virtualizability is not a sufficient condition for a particular language being a good embedding environment because the "simplest possible" embedding might still be prohibitively expensive to realize.

2.1 Achieving Virtualization

What does it take to make a language virtualizable in practice? Various ways of fulfilling subsets of the requirements exist, but we are unaware of any existing language that fulfills all of them. The "pure embedding" approach [22] of implementing embedded languages as pure libraries in a modern host language can likely satisfy *expressiveness*, *safety* and *effort* if the host language provides a strong static type system and syntactic malleability (e.g. custom infix operators). Achieving *performance* in addition, however, seems almost impossible without switching to a completely different approach.

Expressiveness We can maintain *expressiveness* by overloading all relevant host language constructs. In Scala, for example, a for-loop such as

```
for (x <- elems if x % 2 == 0) p(x)
```

is defined in terms of its expansion

```
elems.withFilter(x => x % 2 == 0)
     .foreach(x => p(x))
```

Here, `withFilter` and `foreach` are higher-order methods that need to be defined on the type of `elems`. By providing suitable implementations for these methods, a domain-specific language designer can control how loops over domain collections should be represented and executed.

To achieve full virtualization, analogous techniques need to be applied to all other relevant constructs of the host language. For instance, a conditional control construct such as

```
if (cond) something else somethingElse
```

would be defined to expand into the method call

```
__ifThenElse(cond, something, somethingElse)
```

where `__ifThenElse` is a method with two call-by-name parameters:

```
def __ifThenElse[T]
      (cond: Boolean, thenp: => T, elsep: => T)
```

Domain languages can then control the meaning of conditionals by providing overloaded variants of this method which are specialized to domain types.

In the same vein, all other relevant constructs of the host language need to map into constructs that are extensible by domain embeddings, typically through overloading method definitions.

Performance As we have argued above, achieving *performance* requires the ability to apply extensive (and possibly domain-specific) optimizations and code generation to embedded programs. This implies that embedded programs must be available at least at some point using a lifted, AST-like intermediate representation. Pure embeddings, even if combined with (hypothetical) powerful partial evaluation as suggested in [22], would not be sufficient if the target architecture happens to be different from the host language target. What is needed is essentially a variant of staged metaprogramming, where the embedded "object" program can be analyzed and manipulated by a "meta" program that is part of the embedding infrastructure. However, any DSL will also contain generic parts, some of which will be host language constructs such as function definitions, conditionals or loops. These must be lifted into the AST representation as well.

This ability to selectively make constructs 'liftable' (including their compilation) such that they can be part of (compiled) DSL programs while maintaining *expressiveness*, *safety* and *effort* is an essential characteristic of virtualizable languages.

Modest Effort However, having to implement the lifting for each new DSL that uses a slightly different AST representation would still violate the *effort* criterion. Using an existing multi-stage language such as MetaOCaml [19, 46] would also likely violate this criterion, since the staged representation cannot be analyzed (for safety reasons we will consider shortly) and any domain-specific optimizations would require effort comparable to a stand-alone compiler. Likewise, compile-time metaprogramming approaches such as C++ templates [48] or Template Haskell [41] would not

achieve the goal, since they are tied to the same target architecture as the host language and their static nature precludes dynamic optimizations (i.e. recompilation). What is needed here is a dynamic multi-stage approach with an extensible common intermediate representation (IR) architecture. In the context of Scala, we can make extensive use of traits and mixin-composition to provide building blocks of common DSL functionality (API, IR, optimizations, code generation), including making parts of Scala's semantics available as traits. This approach, which we call lightweight modular staging, is detailed below and allows us to maintain the *effort* criterion. A key element is to provide facilities to compile a limited range of Scala constructs to architectures different from the JVM, Scala's primary target.

Safety There are two obstacles to maintaining *safety*. The first is to embed a typed object language into a typed meta language. This could be solved using a sufficiently powerful type system that supports an equivalent of GADTs [33, 40] or dependent types [32]. The second problem is that with a plain AST-like representation, DSL programs can get access to parts of their own structure. This is unsafe in general and also potentially renders optimizations unsound. Fortunately, there is a technique known as *finally tagless* [8] or *polymorphic embedding* [21] that is able to solve both problems at once by abstracting over the actual representation used.

The combination of lightweight modular staging and polymorphic embedding provides a path to virtualize Scala and actually maintains all four of the criteria listed in the definition of language virtualization.

2.2 Virtualization in Practice

We illustrate our approach of virtualization through lightweight modular staging and polymorphic embedding by means of the following very simple linear algebra example.

```
trait TestMatrix { this: MatrixArith =>
  //requires mixing-in a MatrixArith implementation
  //when instantiating TestMatrix
  def example(a: Matrix, b: Matrix,
              c: Matrix, d: Matrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }
}
```

The embedded DSL program consists of the `example` method in the trait `TestMatrix`. It makes use of a type `Matrix` which needs to be defined in trait `MatrixArith`. The clause

```
this: MatrixArith =>
```

in the first line of the example is a self-type annotation [30]; it declares the type of **`this`** to be of type `MatrixArith`, instead of just `TestMatrix`, which it would be if no annotation was given. The annotation has two consequences: First, all `MatrixArith` definitions are available in the type of the environment containing the `example` method, so this effectively constitutes an embedding of the DSL program given in `example` into the definitions provided by `MatrixArith`. Second, any concrete instantiation of `TestMatrix` needs to mix-in a concrete subclass of the `MatrixArith` trait, but it is not specified which subclass. This means that concrete DSL programs can be combined with arbitrary embeddings by choosing the right mix-in.

Using lightweight staging we can reason about the high-level matrix operations in this example and reduce the number of matrix multiplications from four to a single multiplication. Optimizing matrix operations is one of the classic examples of the use of C++ expression templates [48, 49] and is used by many systems such as Blitz++ [50], A++ [35, 36], and others. We do not have to change the program at all, but just the way of defining `Matrix`.

Here is the definition of matrix operations in `MatrixArith`:

```
trait MatrixArith  {
  type Rep[T]
  type InternalMatrix
  type Matrix = Rep[InternalMatrix]

  // allows infix(+,*) notation for Matrix
  implicit def matArith(x: Matrix) = new {
    def +(y: Matrix) = plus(x,y)
    def *(y: Matrix) = times(x,y)
  }
  def plus(x: Matrix, y: Matrix): Matrix
  def times(x: Matrix, y: Matrix): Matrix
}
```

There is nothing in the definition of `MatrixArith` apart from the bare interface. The definition `Rep[T]` postulates the existence of a type constructor `Rep`, which we take to range over possible *representations* of DSL expressions. In the staged interpretation, an expression of type `Rep[T]` represents a way to compute a value of type `T`. The definition of `InternalMatrix` postulates the existence of some internal matrix implementation, and the definition `Matrix = Rep[InternalMatrix]` denotes that `Matrix` is the staged representation of this not further characterized internal matrix type. The remaining statements define what operations are available on expressions of type `Matrix`.

Since we have not defined a concrete representation, we say that the example code, as well as the definitions of matrix arithmetic operations, are polymorphic in the chosen representation, and hence, we have polymorphically embedded [21] the language of matrix arithmetic operations into the host language Scala. We also note that the embedding is tagless [8], i.e. resolution of overloaded operations is based on static types and does not require dispatch on runtime values. If the representation is abstract, in what way does that help? The answer is that we gain considerable freedom in picking a concrete representation and, perhaps more importantly, that the chosen representation is hidden from the DSL program.

To implement the desired optimizations, we will use expression trees (more exactly, graphs with a tree-like inter-

face), which form the basis of our common intermediate representation that we can use for most DSLs:

```
trait Expressions {
   // constants/symbols (atomic)
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]

  // operations (composite, defined in subtraits)
  abstract class Op[T]

  // additional members for managing
  // encountered definitions
  def findOrCreateDefinition[T](op: Op[T]): Sym[T]
  implicit def toExp[T](d: Op[T]): Exp[T] =
                       findOrCreateDefinition(d)
  object Def { // pattern-match on definitions
    def unapply[T](e: Exp[T]): Option[Op[T]] = ...
  }
}
```

This expression representation will do a number of useful bookkeeping tasks for us, among them automatic elimination of common sub-expressions and, more generally, preventing any expression from being generated twice (e.g. we would only need to compute `a*c` once in our example). The implementation of this bookkeeping is in method `findOrCreateDefinition`, which can be overridden by the DSL designer to further customize the building of the AST. Now we pick `Rep[T] = Exp[T]` and introduce suitable case classes to represent the different node types in our expression tree. We also have to provide a dummy implementation of `InternalMatrix`:

```
trait MatrixArithRepExp extends MatrixArith
                        with Expressions {
  //selecting expression tree nodes representation
  type Rep[T] = Exp[T]
  trait InternalMatrix

  //tree node classes
  case class Plus(x: Matrix, y: Matrix)
    extends Op[InternalMatrix]
  case class Times(x: Matrix, y: Matrix)
    extends Op[InternalMatrix]
  def plus(x: Matrix, y: Matrix) = Plus(x, y)
  def times(x: Matrix, y: Matrix) = Tiems(x, y)
}
```

While we are able to eliminate redundant computation and thus optimize the example, the true power of using domain specific language is our ability to use domain knowledge to perform optimizations. In this case, we can use our knowledge of matrix operations to rewrite some of our expressions into more efficient forms. Implementing these rewritings is very simple using the framework we have developed so far. All we have to do is override the corresponding operations in one of the traits:

```
trait MatrixArithRepExpOpt
                     extends MatrixArithRepExp {
  override def plus(x: Matrix, y: Matrix) =
    (x, y) match {
      // (AB + AD) == A * (B + D)
      case (Def(Times(a, b)), Def(Times(c, d)))
        if (a == c) => Times(a, Plus(b,d))
      case _ => super.plus(x, y)
        // calls default plus() if no match
    }
}
```

Instantiating our example with

```
object MyMatrixApp extends TestMatrix
                       with MatrixArithRepExpOpt
```

constructs an object that generates an optimized version of our example code. It automatically rewrites the sum of multiplications into a single multiplication of a sum of matrices:

```
a * (b + c + c + d)
```

We assume the `println` operation to be overloaded such that it will compile and execute its argument if it is invoked with a staged expression.

The use of domain knowledge in this case yields a tremendous amount of reduction in required computation. This was achieved only using a library with the power of polymorphic embedding and staging, without having to change or create a custom compiler. Note that while we have only shown a simple mechanism for defining optimizations through transformation on operators, much more sophisticated analyses and optimizations are possible by iterating through the entire AST of the program as opposed to one node at time. Liszt, a DSL for mesh-based partial differential equations (PDEs), uses this full-program optimization approach to enable large-scale parallel execution.

3. Physical Simulation with Liszt

Simulation underlies much of scientific computing. The goal of Liszt is to express simulations at a high-level, independent of the machine platform. Liszt is inspired by the pioneering work of others to develop libraries, frameworks and DSLs for scientific computing, including the SIERRA Framework [44], Overture [6], POOMA [37], Sundance/Trilinos [20], and many others.

A major goal of the Stanford University PSAAP Center is to develop a predictive simulation of a hypersonic vehicle. More specifically, the goal of the Stanford Center is to characterize the operability limits of a hypersonic propulsion system using predictive computations, with a primary focus on the unstart phenomena triggered by thermal choking in a hydrogen-fueled scramjet.

To perform these simulations, the center is developing software, called Joe, to solve for the air-flow through the scramjet in the presence of shock waves. Joe solves a RANS (Reynolds-Averaged Navier Stokes) problem on unstructured meshes in the steady and non-steady state using the finite-volume method. Joe has been ported to several clusters and shows scalable performance to 1,000s of processors.

The scalable version is based on MPI, and has evolved from the code used to simulate a jet engine.

Working with the developers of Joe, we have designed Liszt to abstract the machine-dependent parts of the program, allowing us to target Liszt to many architectures. Here is a fragment of Liszt code performing a simple scalar convection operation.

```
val Flux = new Field[Cell,Float]
val Phi = new Field[Cell,Float]
val cell_volume = new Field[Cell,Float]
val deltat = .001
...
while(<not converged>) {
    for(f <- interior_faces) {
        val flux = calc_flux(f)
        Flux(inside(f)) -= flux
        Flux(outside(f)) += flux
    }
    for(f <- inlet_faces) {
        Flux(outside(f)) += calc_boundary_flux(f)
    }
    for(c <- cells(mesh)) {
        Phi(c) += deltat * Flux(c) /
                  cell_volume(c)
    }
    for(f <- faces(mesh))
        Flux(f) = 0.f
}
```

Liszt is designed to abstract the low-level details of mesh-based codes while leaving the application writer enough freedom to implement whatever simulation scheme they need. The language itself contains most of the features of a C-like subset of Scala with three additional domain specific constructs: a built-in mesh interface, the ability to store data on the mesh through fields or sparse matrices, and a for-comprehension over sets of mesh topology that maps computation over the mesh.

The mesh interface abstracts the details of an unstructured mesh that can contain elements of arbitrary types. Liszt meshes are 3D manifolds in 3D space, and hence contain vertices, edges, faces, and cells (the volume enclosed by faces). Liszt provides a complete interface to the mesh topology. For example, a user can obtain sets of mesh elements such as `faces(mesh)`, the set of all faces in the mesh, or for a particular face `f`, `edges(f)`, the set of edges around the face. User-defined sets of mesh topology are also available. In the example, `interior_faces` and `inlet_faces` are user-defined sets representing the interior of the mesh and a particular boundary condition, respectively.

Codes in the domain often need to represent continuous functions over space. Liszt provides fields to represent these continuous functions given by basis functions associated with values defined on cells, faces, edges and vertices of the mesh. Fields are also abstracted and accessed through methods so that their internal representation is not exposed. Accessing a field with a vertex returns the value of the field basis coefficient at that vertex; accessing a field with a set of vertices returns a set of values. Sparse matrices are also provided to support implicit methods. Like fields, sparse matrices are indexed by pairs of topological elements, not integers. Various solvers such as HYPRE [11] and PETSC [3] are built-in to the system.

Finally, a for-comprehension maps computation to each member of a particular topological set. The statement

```
for(f <- cells(mesh)) <block>
```

will execute `<block>` for each cell in `cells(mesh)`. The language definition stipulates that the individual `<block>`s may be executed independently. These semantics give Liszt the freedom to parallelize for-comprehensions and schedule them arbitrarily.

In addition to these special features, Liszt also imposes a set of domain assumptions. Liszt expects that the body of a for-comprehension operates locally on the mesh, accessing a limited neighborhood around the top-level element. Furthermore, Liszt requires that the topology of the mesh remain constant for the duration of the program. These domain assumptions allow the DSL implementor to make aggressive optimizations that would otherwise not be possible, and since these assumptions generally fit they way people write codes in this domain, they do not hamper the expressibility of the language.

This code appeals to the computational scientists because it is written in a form they understand; it also appeals to the computer scientists because it hides the details of the machine and exposes opportunities for parallel execution.

Liszt takes advantage of the language virtualization features that Scala provides to enable aggressive domain-specific optimizations. Using the technique of polymorphic embedding presented previously, Liszt code is lifted into an internal representation, where a series of program analyses and aggressive domain-specific optimizations are performed. The result is a native C++ program that can run efficiently on an MPI-based cluster. A different set of transformations is being developed that allows us to target Liszt to a GPU.

While it would be possible to write a version of Liszt in Scala without any embedding, a simple library-based implementation of Liszt could not achieve good scalable performance. Consider the first for-comprehension in the example. The body of the for-comprehension adds the value of flux to the field entries for the two cells on the sides of face `f`. Since each cell is surrounded by many faces, multiple instances of the statement body end up writing data to flux value of the same cell. To avoid race conditions in this situation, an implementation of Liszt must introduce some form of synchronization. A naive solution might use locking or atomic primitives to update the values of the `Flux` field. While this would perform correctly, it introduces high overheads to these frequently occurring operations, and would not work for distributed memory systems. Instead, the commonly used approach in the field is to decompose the do-

main by partitioning the mesh into many pieces and assigning each piece to a single thread. Now synchronization only needs to occur on the borders between partitions. This synchronization is normally accomplished using "ghost" cells. If a particular thread needs to write to a value outside of its partition, it will create a duplicate entry, a "ghost", that will hold the partial values that the thread produces for that value, and these temporaries will be sent to the owner of the value only after the end of the for-comprehension. This strategy decreases communication overhead by batching the messages to other threads. It has been used successfully by Joe to show scalable performance to 1,000s of processors.

However, efficient approaches like the one described above traditionally need to be explicitly managed by the end-user. Code to partition the mesh into many sections, manage the ghost-cells, set up communication between threads, and perform batched messages would need to be inserted to ensure the program would scale. For Joe, this boilerplate code and the libraries needed to support it account for nearly 80% of the code. Furthermore, if a different parallelization strategy were to be applied, the client code would need to be significantly modified to support it.

Liszt uses language virtualization to avoid these issues. The client code does not specify a strategy for parallelization. Instead, Liszt first creates an intermediate representation of a client program. It then uses domain knowledge to map the intermediate representation to heterogeneous parallel computers. We have identified several situations where this domain knowledge enables aggressive code transformations, allowing for automatic parallelization.

- **Automatically perform domain decomposition.** In Liszt, all mesh accesses are performed through a standard mesh interface. The Liszt compiler analyzes the pattern of access to neighbors and provides input to the domain decomposition software. In particular, Joe uses ParMETIS [27] to perform domain decomposition. ParMETIS reads a graph of vertices and edges. The vertices of the graph are mesh cells and the edges are neighboring mesh elements accessed during the computation. Liszt can create this graph automatically for ParMETIS, which is then able to decompose the mesh into optimal domains. All this is done automatically by the system.
- **Automatic determination and maintenance of ghost cells.** The same neighborhood analysis used to partition the domain can be used to determine which neighboring mesh elements are shared across a boundary. Liszt automatically finds the neighbors of a domain, and handles all communication and synchronization across the boundaries of the domain. This very domain-specific knowledge is crucial for minimizing communication in Liszt programs on distributed memory clusters.
- **Selection of mesh representation.** Again, because the mesh topology is accessed through standard APIs, Liszt can analyze those API calls to determine what neighborhood relationships are used by the program. Liszt can build a table of what neighborhood relationships are being used, and choose the best mesh representation for the application. In effect, Liszt chooses the optimal mesh data structure.
- **Optimize layout of field variables.** A major design choice is how to represent fields. Fields are associated with mesh elements. In general, there will be a set of fields stored on cells, on vertices, etc. There are two ways to represent these fields. The most natural is as an array of structures. That is, the cell contains a structure that stores all the fields associated with the cell. Another choice is as a structure of arrays. That is, each field is stored in a separate array. Which representation to use depends on how and when the fields are used, and on the machine architecture. Liszt can analyze the program for co-usage of fields and then optimally map to a given architecture.

All of these examples depend on domain-knowledge. In this case, knowledge about meshes and fields. No general-purpose compiler could possibly do these types of optimizations. After these transformations are applied, Liszt can generate code for a specific runtime. Currently a MPI-based cluster runtime exists with a CUDA-based GPU runtime in progress. We eventually plan to unite the two runtimes, allowing Liszt to target a heterogeneous system consisting of some traditional cluster nodes each accelerated with GPUs.

While it would be possible to write Liszt as a stand-alone language, embedding it as a virtual language inside Scala has several important advantages. Since it uses the same syntax and typing rules as standard Scala, it is easy for someone who knows Scala to understand and create Liszt code. IDE and tool support for Scala also works on Liszt code, making it easier to use than a stand-alone language. Furthermore, Liszt can also benefit from the staged computation that a virtualized language provides. In many scientific codes, the programmer can make a tradeoff between using a low-order (e.g linear) basis function to get a fast reconstruction of a field, or a higher-order basis function to get a more accurate reconstruction at the cost of additional computation and memory access. Often a solution may contain bases of mixed orders, with higher-order bases chosen for important parts of the mesh. This determination of the basis functions normally occurs during program initialization, but is constant after it has been chosen. Since the order of basis functions can determine the way the fields are accessed and the sparsity patterns of the matrices, it is beneficial to know this information before choosing field or matrix layouts. Using staged computation, Liszt can first generate code to compute the order of basis functions and then use the resulting values as input into the next stage of compilation, specializing the code to the particular assignment of basis functions that the first stage produced.

Through language virtualization, Liszt is able to perform the strong optimizations of a stand-alone domain-specific language without the need for a new language and compiler infrastructure. In the future this approach offers the ability to interoperate with other DSLs as well as share a common infrastructure for program analysis and code generation for heterogeneous architectures.

4. Data Analysis and Machine Learning with OptiML

The second example of the use of language virtualization is for the implementation of OptiML, a DSL for machine learning. Machine Learning (ML) algorithms are becoming increasingly prevalent in a variety of fields, including robotics, data analytics, and web applications. Machine learning applications are often used to process and find structure within very large datasets or from real-time data sources. These workloads often require great computational resources, even with efficient algorithms. To meet these computational needs effectively, much painstaking effort has been focused on taking advantage of modern multi-core and heterogeneous hardware [12,18]. However, these solutions are usually complex, platform-dependent, and require significant initial investment to understand and deploy. As a result, ML researchers and developers spend a disproportionate amount of their time hand-tuning kernels for performance, a process that has to be repeated for every kernel and every platform. An ML DSL, on the other hand, could leverage the underlying abstractions that are common across many ML algorithms to simplify the way that parallel ML applications are developed. To take advantage of these and speed up the development of parallel ML applications, we are developing the OptiML DSL.

The primary design goal of OptiML is the development of implementation-independent abstractions that capture the essential operations in machine learning. Using machine-independent representations, machine learning developers can focus on algorithmic quality (i.e. improving accuracy) and correctness while allowing the runtime system to manage performance and scaling. OptiML is implicitly parallel, so users are not required to do any explicit decomposition or parallelization.

The following code snippet from k-means clustering is an example of what an OptiML program would look like prior to compilation:

```
val x = TestSet(inputFile)
val c = BestEffortVector[Int](m)

val k = 10 // number of clusters
val m = x.numRows // number of data points
// k randomly initialized cluster centroids
val mu : Vector[Double] = initializeClusters(x)

c.setPolicy(ConvergingBestEffortPolicy(m))

until converged(mu) {
  // best effort update c --
  // calculate distances to current centroids
  for (i <- 0 until x.length) {
    c(i) = findNearestCluster(x(i), mu)
  }
  // update mu -- move each cluster centroid
  // to mean of the points assigned to it
  for (j <- 0 until k){
    var (weightedpoints, points) = sum(i,0,m) {
      if (c(i) == j) (x(i), 1) else (0,0)
    }
    mu(j) = weightedpoints/points
  }
}
```

The OptiML surface syntax is regular Scala code with high-level, domain specific abstractions introduced to make implementing ML algorithms easier. Underneath, however, OptiML is polymorphically embedded in Scala. The user program implicitly operates on abstract representations whose implementations can be chosen at stage time without modifying the application code. Like Liszt, virtualization allows OptiML to lift user applications into an internal representation. During staging, OptiML performs domain-specific optimizations on this internal representation. Before describing these optimizations, we first note the domain-specific characteristics that OptiML tries to exploit.

Most ML algorithms perform basic operations on vectors and matrices. Other fundamental operations are minimization, maximization, and convex optimization. Many of the algorithms expose parallelism at multiple levels of granularity; most have coarse-grained data parallelism (e.g. k-means) while others have only a small amount of work per data sample (e.g. linear regression). Several algorithms (logistic regression (LR), support vector machines (SVM), reinforcement learning) perform gradient descent on some objective function, iterating until they converge. This usually results in running the same kernel repeatedly on the same dataset many times. Many ML datasets (training sets) are very large, bound only by the execution time required for training; usually the execution time is bounded by the available memory bandwidth due to low arithmetic intensity. The datasets typically have some exploitable structure; the most common are audio, image, or video data. Furthermore, these datasets may contain large amounts of redundancy (e.g., network traffic) that can be exploited to reduce total computation [9]. Finally, many ML algorithms are probabilistic in nature and are robust to random errors such as dropped or incorrect computations.

These observations lead to the following proposed program transformations and analyses for OptiML. We group them by whether they are applied during staging time or runtime.

Optimizations done at staging time include:

- **Transparent compression:** We plan to explore the possibility of opportunistically compressing OptiML data structures before transferring to or from heterogeneous

computing devices. While there is significant overhead to compression, several aspects of ML discussed previously make this a candidate for a winning trade-off: large datasets, low arithmetic intensity, and compressible data types (images, audio, video). In practice, most ML algorithms are bound by memory bandwidth, so on-the-fly compression could significantly decrease the total execution time when using heterogeneous devices. Stage-time analysis can generate code paths with explicit compression and decompression for large inputs.

- **Device locality:** OptiML will use stage-time information about object lifetime and arithmetic intensity to determine the target device for individual kernels. Furthermore, when a dataset is iterated over multiple times for different operands, clever scheduling can interleave the memory accesses to improve cache utilization.
- **Choice of data representation:** OptiML will generate code for multiple input-dependent data implementations (e.g. sparse and dense matrices), allowing the developer to use simple high-level representations (e.g. Matrix) that have identical semantics without worrying about the performance impact. OptiML will also use program analysis to choose the best representation for particular kernels, and can insert conversions from one implementation to another when it is likely to be beneficial to do so.

Optimizations done at run-time include:

- **Best-effort computing:** for some datasets or algorithms, many operations are not strictly required, and can be executed with "best-effort" semantics. This allows the OptiML programmer to signal to the runtime that it can drop certain data inputs or operations that are especially expensive if it is already under heavy load.
- **Relaxed dependencies:** it is sometimes possible to trade-off a small degree of accuracy for a large gain in efficiency, especially when dealing with large datasets. This implies that some dependencies, such as parameter updates (especially across loop iterations), can be relaxed to varying degrees. By defining the strictness of a dependency, the OptiML user can choose the trade-off between exposing additional parallelism and increasing accuracy. The runtime will then dynamically decide to drop dependencies if it is advantageous to do so under the current workload.

In addition to the optimizations described here, OptiML must eventually generate parallel code targeted at heterogeneous machines. This is enabled by Delite, the framework and runtime that OptiML is being built on top of. We introduce Delite in the next section.

5. DSL Parallelization with Delite

In the Liszt example, Liszt explicitly maps programs to heterogeneous platforms by directly generating code written in lower level programming models such as CUDA and MPI. This allows Liszt to have maximal control over each stage of its compilation and runtime. This flexibility comes at the cost of implementation complexity; even within a virtualized host language, handling all aspects of the parallelization of DSL programs places a heavy burden on DSL developers. Since many components of a parallel DSL are potentially reusable (e.g. scheduling), we believe that it is important to develop frameworks and runtimes that aid in the task of parallelizing DSLs.

Delite is a framework for simplifying DSL parallelization and development. It first provides a set of predefined AST nodes for the common parts of domain-specific languages. Essentially, this is equivalent to embedding the Scala language itself. This allows the DSL developer to start with the base Scala AST and then add domain-specific nodes and semantics to the language. Since the embeddings are modularized into a variety of traits, the DSL developer is free to choose the subset of functionality to include in his virtual language.

Delite also provides a set of special case classes that can be inherited from when defining the node types of the language's AST (recall `MatrixArithRep` in Section 2). In addition to a case class that represents a sequential task, Delite provides case classes for a variety of parallel execution patterns (e.g. Map, Reduce, ZipWith, Scan). Each of these case classes constrains what information the DSL developer needs to provide so that Delite can automatically generate parallel code. For example, in OptiML, the scalar multiplication AST node inherits from Map. The DSL developer would then only have to specify the mapping function; Delite handles generating the parallel code for that operation on a variety of targets such as the CPU and GPU.

A Delite program goes through multiple compilation stages before execution. The first compilation stage, written by the DSL developer, uses virtualization to lift the user program into an internal representation and performs applicable domain-specific optimizations (e.g. the stage-time OptiML transformations). However, instead of generating explicitly parallel code, the DSL generates an AST where DSL operations are represented as Delite nodes (e.g. a Map node).

The AST is then compiled in subsequent stages by Delite. Delite expands the nodes generated by the DSL developer to handle Delite-specific implementation details (e.g. data chunking) and perform generic optimizations (e.g. operation fusing). Delite also generates a static schedule for straight-line subgraphs in the user program, which reduces the time required to make dynamic scheduling decisions during actual execution. In the last compilation stage, Delite maps each domain-specific operation to different hardware targets. The final result of compilation is an optimized execution graph along with the generated kernels. The Delite runtime

executes the graph in parallel using the available hardware resources.

6. Related Work

There is a long history of research into parallel and concurrent programming. Parallel programming is now standard practice for scientific computations and in specialized application areas such as weather or climate forecasting.

Parallel programming codes are usually procedural/imperative with explicit segmentation between processors. Communication is done by specialized libraries such as MPI or, in shared-memory environments, parallelism is made explicit via compiler directives such as in OpenMP. Interesting alternatives such as SISAL [15], SaC [39] or NESL [5] have been tried but were either ahead of their time or too specialized to be adopted by a large community. The same holds for data-parallel Haskell [25], which is built around nested array parallelism similar to NESL. The explicit message passing paradigm of MPI is not without critique; some argue that a model based on send and receive primitives is too low-level and that communication should be represented in a more structured manner, e.g. using collective operations [17]. Google's MapReduce framework is essentially such a model [13].

Recently, the high-performance programming language designs Chapel [10], X10 [38] and Fortress [26] have emerged from a DARPA initiative. These languages combine sophisticated type systems with explicit control of location and concurrency. They are targeted primarily at scientific applications for supercomputers. It remains to be seen to what degree they can be applied to general programming on a variety of heterogeneous hardware. We believe that some of the innovations of these languages (regions, configuration variables) can be generalized and made more powerful in the staged compilation setting that we propose.

When performance is paramount and the sheer number of possible combinations makes manual specialization intractable, *program generation* is an approach that is often used. Instead of building a multitude of specialized implementations, a program generator is built that, given the desired parameters as input, outputs the corresponding specialized program. A number of high-performance programming libraries are built in such a way, for example ATLAS [51] (linear algebra), FFTW [16] (discrete fourier transform), and Spiral [34] (general linear transformations). However, good program generators still take a huge effort to built and often, the resulting generator implementation will no longer resemble the original algorithm.

The ideas underlying *language virtualization* are both very old and quite new. Several of its important aspects have been with us since the invention of Lisp [43] more than 50 years ago. The Lisp model of "code as data", supported by macros, makes it very easy to define and process embedded domain-specific languages. A common viewpoint is that Lisp is not so much a programming language, but rather a way to express language abstractions. In the words of Alan Kay: "Lisp isn't a language, it's a building material".

On the other hand, language embeddings in Lisp can be *too* seamless in that they do not distinguish between the embedded DSL and the hosting framework. Embedded DSL programs can observe the way their embeddings work and can access fairly arbitrary parts of the host environment. These limitations can be overcome in principle, given enough effort. A hosting environment could statically analyze embedded domain-specific code for safety violations. However, such analyses are non-trivial tasks, as they basically subsume implementations of static type checkers.

A common approach to DSL embedding is to assemble an abstract syntax tree (AST) representation of embedded programs. However, even in statically typed settings such as LINQ [7] or earlier work on compiling embedded languages [14, 29] the type of the AST is publicly available. An embedded language might use this fact to manipulate its own representation, thus undermining its virtualization.

Tighter encapsulation is provided by *staging* (or multi-stage programming). Staging shares with language virtualization the idea that DSL programs are assembled and run as part of the host program execution. But it provides no control on the choice of representation of the language embeddings. The usual approach, taken e.g. by MetaOCaml [19], is to use different kinds of syntactic brackets to delineate staged expressions in a DSL, i.e., those that will be part of the generated program from the host program. Staged expressions can have holes, also marked by special syntax, into which the result of evaluating the contained generator stage expression will be placed. Finally, staged expressions can be run, which will cause them to be assembled as program source code, run through the compiler, with the resulting object code being dynamically loaded and executed. In essence, staging as implemented in MetaOCaml is similar to macros using quote/unquote/eval in Lisp, but with a static type system that ensures well-formedness and type safety for the generated code at the time the multi-stage program is compiled

A closely related approach to code generation is template expansion, as implemented in C++ [48] or Template Haskell [41]. The main difference to multi-staged programming is that all template expansion is done at compile-time. In contrast to staging, some template expansion mechanisms have a concept of user-definable rewriting rules which enable a limited form of domain-specific optimizations. However, the target of the compilation of embedded languages is always the same as the host language's.

A generalized approach to language embedding which is referred to as *finally tagless* or *polymorphic* embedding was introduced by Carette et al. [8] and taken up by Hofer et al. [21]. The former focus on the basic mechanism of removing interpretive overhead while the latter stress modularity and the ability to abstract over and compose seman-

tic aspects. We show in this paper how to use polymorphic embeddings in language virtualization to optimize and stage parallel domain specific languages.

7. Conclusion

Enabling mainstream programming to take advantage of heterogeneous parallel hardware is difficult. In fact it has been identified as a grand challenge problem (PPP: "Popular Parallel Programming") by the computer architecture community [24]. In this paper we have proposed an embedded domain specific approach based on language virtualization to address this challenge.

Our approach is not a "silver bullet" that will magically parallelize arbitrary programs. But language virtualization provides a path to get there. As new application domains come up, one can put effort into designing good parallel embedded languages for these domains and use a virtualization framework to map them to parallel execution environments. Likewise, as future parallel hardware evolves one can extend the framework to accommodate new architectures. This creates a realistic prospect that heterogeneous parallel execution environments can be harnessed in an increasing number of domains without requiring unreasonable effort.

References

[1] Scala. `http://www.scala-lang.org`.

[2] AMD. The industry-changing impact of accelerated computing. `http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf`.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[5] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[6] D. L. Brown, W. D. Henshaw, and D. J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *SIAM conference on Object Oriented Methods for Scientfic Computing*, volume UCRL-JC-132017, 1999.

[7] C. Calvert and D. Kulkarni. *Essential LINQ*. Addison-Wesley Professional, 2009.

[8] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated. In Z. Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2007.

[9] S. Chakradhar, A. Raghunathan, and J. Meng. Best-effort parallel execution framework for recognition and mining applications. In *Proc. of the 23rd Annual Int'l Symp. on Parallel and Distributed Processing (IPDPS'09)*, pages 1–12, 2009.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[11] E. Chow, A. Cleary, and R. Falgout. Design of the hypre preconditioner library. In M. Henderson, C. Anderson, and S. Lyons, editors, *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 21–23, 1998.

[12] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS '06*, pages 281–288, 2006.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[14] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(03):455–481, 2003.

[15] J. M. et. al. SISAL: Streams and iterators in a single assignment language, language reference manual. Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

[16] M. Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[17] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[18] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik. Parallel support vector machines: The cascade svm. In *NIPS '04*, 2004.

[19] M. Guerrero, E. Pizzi, R. Rosenbaum, K. Swadi, and W. Taha. Implementing dsls in metaocaml. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 41–42, New York, NY, USA, 2004. ACM.

[20] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[21] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of dsls. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

[22] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142, 1998.

[23] Intel. From a few cores to many: A tera-scale computing research review. `http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`.

[24] M. Irwin and J. Shen, editors. *Revitalizing Computer Architecture Research*. Computing Research Association, dec 2005.

[25] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In R. Hariharan, M. Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[26] G. L. S. Jr. Parallel programming and parallel abstractions in fortress. In *IEEE PACT*, page 157. IEEE Computer Society, 2005.

[27] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

[28] 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. This provides a current overview of the entire Telescoping Languages Project.

[29] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL: Proceedings of the 2 nd conference on Domain-specific languages: Austin, Texas, United States*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1999.

[30] M. Odersky and M. Zenger. Scalable component abstractions. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 41–57. ACM, 2005.

[31] K. Olukotun, B. A. Nayfeh, L. Hammond, K. G. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS '96*.

[32] E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Not.*, 37(9):218–229, 2002.

[33] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[34] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[35] D. Quinlan and R. Parsons. A++/p++ array classes for architecture independent finite differences computations. In *ONNSKI*, 1994.

[36] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan. Treating a user-defined parallel library as a domain-specific language. In *IPDPS*. IEEE Computer Society, 2002.

[37] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, K. Keahey, M. Srikant, and M. Tholburn. Pooma: A framework for scientific simulation on parallel architectures, 1996.

[38] V. A. Saraswat. X10: Concurrent programming for modern architectures. In *APLAS*, page 1, 2007.

[39] S.-B. Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

[40] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for gadts. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 341–352, New York, NY, USA, 2009. ACM.

[41] T. Sheard and S. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[42] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.

[43] G. L. Steele. *Common Lisp the Language*. Digital Press, Billerica, MA, 1984.

[44] J. R. Stewart and H. C. Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elem. Anal. Des.*, 40(12):1599–1617, 2004.

[45] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[46] W. M. Taha. *Multistage programming: its theory and applications*. PhD thesis, 1999. Supervisor-Sheard, Tim.

[47] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[48] D. Vandevoorde and N. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.

[49] T. Veldhuizen. Expression templates, C++ gems, 1996.

[50] T. L. Veldhuizen. Arrays in blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *ISCOPE*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998.

[51] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.