A SEMANTICS-BASED APPROACH TO OPTIMIZING UNSTRUCTURED MESH ABSTRACTIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Brian Stephen White

May 2008

A SEMANTICS-BASED APPROACH TO OPTIMIZING UNSTRUCTURED MESH ABSTRACTIONS

Brian Stephen White, Ph.D.

Cornell University 2008

Computational scientists are frequently confronted with a choice: implement algorithms using high-level abstractions, such as matrices and mesh entities, for greater programming productivity or code them using low-level language constructs for greater execution efficiency. We have observed that the cost of implementing a representative unstructured mesh code with high-level abstractions is poor computational intensity—the ratio of floating point operations to memory accesses. Related scientific applications frequently produce little "science per cycle" because their abstractions both introduce additional overhead and hinder compiler analysis and subsequent optimization. Our work exploits the semantics of abstractions, as employed in unstructured mesh codes, to overcome these limitations and to guide a series of manual, domain-specific optimizations that significantly improve computational intensity.

We propose a framework for the automation of such high-level optimizations within the ROSE source-to-source compiler infrastructure. The specification of optimizations is left to domain experts and library writers who best understand the semantics of their applications and libraries and who are thus best poised to describe their optimization. Our source-to-source approach translates different constructs (e.g., C code written in a procedural style or C++ code written in an object-oriented style) to a procedural form in order to simplify the specification of optimizations. This is accomplished through *raising operators*, which are spec-

ified by a domain expert and are used to *project* a concrete application from an *implementation space* to an *abstraction space*, where optimizations are applied. The transformed code in the abstraction space is then reified as a concrete implementation via *lowering operators*, which are automatically inferred by inverting the raising operators. Applying optimizations within the abstraction space, rather than the implementation space, leads to greater optimization portability.

We use this framework to automate two high-level optimizations. The first uses an inspector/executor approach to avoid costly and redundant traversals of a static mesh by memoizing the relatively few references required to perform the mathematical computations. During the executor phase, the stored entities are accessed directly without resort to the indirection inherent in the original traversal. The second optimization *lowers* an object-oriented mesh framework, which uses C++ objects to access the mesh and iterate over mesh entities, to a low-level implementation, which uses integer-based access and iteration.

BIOGRAPHICAL SKETCH

Brian graduated from Carnegie Mellon University in 1998, where he majored in Computer Science and minored in Mathematics. He worked under Andrew Grimshaw at the University of Virginia and graduated with a Master of Computer Science in 2002. He spent the prior year as a visiting research assistant under Jay Lepreau at the University of Utah. He received a Master of Science degree in Electrical and Computer Engineering from Cornell University in 2006, while working with Sally McKee. His minor field of concentration was Physics. Additionally, Brian spent time during the summers of 2004, 2005, and 2006 working with Dan Quinlan at Lawrence Livermore National Laboratory.

To my mother, father, and sister, Nicole, who supported and encouraged me through each of the many bends in my studies.

ACKNOWLEDGMENTS

My forays into the different nooks of computer science would not have been possible without the mentors who guided me along the way: Paul Kram, Wee Teck Ng, Liddy Shriver, Andrew Grimshaw, and Jay Lepreau. Their attention instilled confidence. I only hope that I can repay their generosity and patience through my own interactions with others.

I am deeply indebted to my advisor, Sally McKee, and Dan Quinlan for channeling my academic wandering into a focused topic and to the rest of my committee, Martin Burtscher, Rajit Manohar, Keshav Pingali, and David Rubin, for sharpening that focus through insightful comments and questions. Nevertheless, I occasionally let that focus blur to encompass other intellectual pursuits that I found to be greatly enriching and a reward for slow, but steady, thesis progress. I am forever grateful to Sally for forgiving this straying from time to time from the straight and narrow path of compiler research. Further, it was David Shalloway who gave me a chance to apply my academic meanderings through physics and biology. I am more appreciative of the time and opportunity he has given me than he could know.

Though the path has been long, it has been one of great intellectual and cultural enlightenment. For that I have to thank a wonderful group of friends that challenged me to think beyond myself and the biases of my own narrow perspectives and experiences. I will always have warm feelings for Michael's Bistro and the "inner circle" that sometimes assembled there as many as three times in one day. I treasured the deep conversations with John Regehr, Glenn and Kim Wasson, and David Coppit and was inspired by their near limitless array of competences. I am no less fortunate to have had such close cronies as Rob Schutt, Eric Smith, and Tim Bellaire. At Cornell I was blessed to count amongst my friends Jen

Williams, Kathryn Prybylski, Kirsten Myers, Tuncay Alan, Paulo Santos, Jordan Suter, Martin and Laura Schulz, Pete Szwed and Julibeth Corwin, Attila Bergou, and Susan Kendrick. The years have chased many of them away and driven me from one establishment to another, each filled with memories of good conversation and strong libations: political discussions at the Chapter House, chats over tokens at Moonshadows, and book critiques at Karova. Without these dear friends my reflections on Cornell would fall on little more than books and computer screens, instead they are colored with grumblings at Teagle, sunsets on the Bosporus, and Weisswurst in German beer halls.

Most importantly, I thank my mother, my father, and my sister, Nicole. When I had nearly given up, they never lost faith. When I continually complained and questioned my career, they never lost patience. When the days were dark and even the motivation of pride was drained away, their love was sustaining.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Problem: Abstractions Improve Programmer Productivity but Degrade Program Efficiency

Abstractions obfuscate; they should instead illuminate.

The use of high-level abstractions, such as matrices and differential operators, is a key to achieving high-productivity scientific computing [44]. Abstractions are frequently implemented in domain-specific libraries as user-defined types and the procedures acting on them. Such abstractions are a closer match to the concepts and notations employed in scientific domains than low-level implementations that expose raw details, such as matrix element storage and mesh spacing. Ignoring such details allows domain experts to more concisely setup and solve problems. For example, Dinesh *et al.* [21] found a 30% reduction in code size and, with it, a significant and quantifiable improvement in programmer productivity and code maintainability when side-effect free algebraic notion was used to solve partial differential equations rather than an object-oriented style inconsistent with mathematical notion. Choosing from a library of well-constructed abstractions improves code reuse, and with it software maintainability, and allows domain experts to focus their efforts on science rather than on computer science. The attendant improved productivity can have a significant impact on funding since scientific applications at government laboratories, such as Lawrence Livermore National Laboratory, are evaluated primarily on the richness of their feature sets.

Abstractions mitigate the growing complexity of scientific applications and computer architectures. Scientific frameworks tend to be highly configurable: mesh solvers can be parameterized according to equation (e.g., Navier-Stokes or Euler); assumptions (e.g., the ideal gas law or van der Waals equation of state); and fluid (e.g., water or a monatomic gas). The implementations swell as a result: Spheral++ [64], a meshless code for studying hydrodynamics in astrophysics simulations, has 400 files and 900K lines of code; SAMRAI [38], a framework providing adaptive mesh refinement, has 9500 files and 100K lines of code; Trellis [8], a package for the solution of partial differential equations has 900 files and 300K lines of code. Implementing such large software projects in low-level C or Fortran without encapsulating functionality in abstractions would have an adverse effect on code readability and debugging. Similarly, abstractions can mask architectural complexity to provide performance portability across systems [44]. For example, ATLAS [88] is a library of empirically-optimized linear algebra routines. ATLAS installation determines how best to set optimization parameters, such as loop blocking factors, resulting in a library that is tuned to an architecture without requiring it to be hardcoded to that architecture's characteristics.

These benefits have made abstractions and the higher-level languages in which they are most often implemented viable competitors to the mainstay of scientific computing, Fortran 77. For example, Spheral++ employs C++ classes to implement tensor and vector abstractions and has even embraced the convenience of python to quickly instantiate simulations that ultimately invoke computational modules implemented in C++.

Despite these benefits their inferior performance remains the major impediment to the wide-scale adoption of abstractions [7]. Anecdotes in the literature mention

university groups favoring the convenience of Matlab's abstractions during initial development, but ultimately having to manually recode their implementations in C or Fortran to get acceptable performance on large-scale problems [45, 74]. Abstractions coded in C++, which does not suffer from Matlab's frequent run-time type checking, show similarly degraded performance. For example, Yi and Quinlan describe an automatic procedure for effectively *lowering* container abstractions to Fortran-style arrays to expose optimization opportunities and improve performance [91].

Abstractions perform poorly because they obfuscate compiler analysis. From a programmer's perspective, type or procedure abstractions defined in a library extend the language. This view is particularly apt in C++ where, through operator overloading, abstractions are utilized as effortlessly as built-in types and operators. Nevertheless, without support from the compiler, abstractions can not be considered language extensions. Thus, whereas a compiler recognizes that a conditional using the built-in equality operator, `==`, is side-effect free, it will have to prove this same fact through analysis for a user-defined `operator==`, despite its likely semantic equivalence. If the implementation of `operator==` is not accessible to the compiler at the site of its invocation, the compiler will be forced to treat the procedure as a black box and to conservatively assume it has side effects.

Figure 1.1 provides a simple, yet representative, example in which a compiler's inability to reason about abstractions prevents optimization. The example is a simplified calculation from molecular dynamics applications and consists of two loops: the first updates the velocity of each particle in a list according to Newton's Law, while the second updates the particles' positions based on the new velocities. The loops should be fused to exploit the reuse of each particle by the second loop.

```
void updateParticles(list<particle *> &particles, double dt)
{
  list<particle *>::iterator it;

  // Iterate over particles, updating each of their velocities
  // in the presence of a force experienced at that particle.
  for(it = particles.begin(); it != particles.end(); ++it) {
    (*it)->velocity -= (*it)->force * dt / (*it)->mass;
  }

  // Iterate over particles, updating each of their positions.
  for(it = particles.begin(); it != particles.end(); ++it) {
    (*it)->position += (*it)->velocity * dt;
  }
}
```

Figure 1.1: Molecular dynamics code for updating particle velocity and position.

Unfortunately, because the compiler can not infer the uniqueness of the particles within the list, it must assume that each loop carries a flow dependence, i.e., that each statement within each loop body reads a variable written in a previous iteration. There is a true flow dependence from the first loop to the second due to the accesses to the `velocity` field. Taken together these potential dependences prevent the compiler from fusing the loops to realize the temporal reuse.

1.2 Solution: Improve Execution Efficiency through Abstraction Semantics

Far from being a hindrance to compiler optimization, abstractions should instead enable traditional optimizations and illuminate novel, domain-specific optimization opportunities. For example, knowledge of the uniqueness of list elements in the above velocity update code ensures that there are no loop-carried dependences and

that the loops are candidates for traditional fusion techniques. Similarly, Ding and Kennedy describe a novel, locality-grouping transformation for optimizing the particle-particle interactions in molecular dynamics applications that leverages domain knowledge for correctness and profitability [22]. Forces are calculated between pairs of particles by iterating over an interaction list. By reasoning that the loop performing the force calculations does not carry dependences, they assert that reordering the iteration space is permissible. By recognizing that a particle is included in multiple interaction pairs, not all of which are contiguous in the original list, they contend that reordering to introduce temporal locality is profitable. Such an optimization is necessarily domain specific. However, the particle interaction list is a motif prevalent in molecular dynamics applications; therefore, an ideal transformation should be neither application- nor implementation-specific.

Abstractions are defined to a large extent by their semantics, which are independent of any particular implementation. It is these semantics that guide application programmers to select and utilize a particular abstraction. Compilers should adopt a similar perspective: by targeting abstractions according to their semantics, rather than their implementations, compiler transformations can be broadly applied across a domain. Optimizations that are instead encumbered by low-level implementation details, such as whether an operator is defined as a Fortran procedure or a C++ method, are more difficult to write and maintain and are less likely to be widely accepted. It is impractical for each scientific code team to invest effort re-implementing transformations specific to their implementation that have developed elsewhere for a different application in the same domain. Unfortunately, it is these code teams, the library developers, that are best attuned to the semantics of the abstractions they define and use, and so are the best equipped to implement or at least to design these transformations. Writing portable opti-

mizations applicable domain-wide would allow their development to be amortized across code teams, which should then show an increased willingness to participate in a communal optimization effort.

To summarize, abstractions present a number of obstacles to automated transformation:

Abstractions are best characterized by their semantics and not by their implementations. These semantics are difficult to infer from source-code inspection. Since abstractions are frequently pointer rich, analyses are challenged to navigate potential aliases in order to infer program properties. Often semantics are not codified explicitly within an application or library. Therefore, considerable analysis effort may be expended to determine incidental effects, such as side effects due to reference counting, without capturing high-level properties.

There is no means to communicate abstraction semantics to a compiler. Given the difficulty of automatically inferring semantics, the library writer should be able to explicitly specific semantics. Traditional compilers support a limited set of low-level keywords, pragmas, and flags, including the `const` and `restrict` keywords, but lack a more general framework that accommodates a high-level domain perspective. Such a framework would make abstractions amenable to the wealth of existing compiler transformations.

A general-purpose compiler can not anticipate the optimization needs and opportunities of domain-specific abstractions. An ability to specify semantics is sufficient to enable traditional optimizations, but to fully exploit the performance potential of high-level abstractions an extensible compiler framework

should allow library writers to convey domain-specific transformations.

1.3 Specific Approach: Target Abstractions Rather Implementations with Expert-Driven Optimizations

To overcome these difficulties we propose to reason about and optimize abstractions in a space in which they are viewed as abstract concepts, defined by their semantics, independent of their implementations. This view is expressed schematically in Figure 1.2. The technique is based on a common target—an abstract syntax—for high-level optimizations of diverse implementations. This approach is related to the use of intermediate representations (IRs) in traditional compilers. Compiler frontends represent code from a diverse set of source languages in the IR, which is then analyzed and optimized before being emitted in native form. Since analyses and optimizations target the IR, they need not be re-implemented for each source language. Similarly, our source-to-source approach translates different constructs (e.g., C code written in an imperative style or C++ code written in an object-oriented style) to a procedural or imperative form in order to simplify the specification of optimizations. This is accomplished through *raising operators*, which *project* a concrete application from an *implementation space* to an *abstraction space*. In the example depicted in the figure, the source code providing the concrete implementation of gradient and divergence operators is projected into an abstraction space.

Because reasoning within the implementation space is limited by the precision and scope of compiler analyses, analyses and transformations act instead within the abstraction space, guided by semantics. Analysis of the source code would have

to determine, for example, that none of the statements induce side effects except through the obvious assignments. Within the abstraction space, the semantics of the gradient and divergence operators dictate that they are pure functions, dependent only on the mesh and an input field. This domain knowledge allows the fusing of the two operators into a single gradient-divergence operator that simultaneously iterates over the mesh, acting on and returning separate fields. Consistent with the ROSE philosophy, the specification of such optimizations is left to domain experts and library writers who best understand the semantics of their applications and libraries and who are thus best poised to describe their optimization. The transformed code in the abstraction space is finally reified as a concrete implementation via *lowering operators*. In the example, the two loops are fused to implement the gradient-divergence operator.

Projecting an implementation to a standardized abstract interface has several benefits. The domain-specific optimization, which targets the abstract interface, will be applicable and portable across implementations of abstractions and the frontend languages supported by ROSE (currently C, C++, and Fortran 2003). Also, the burden of implementing the optimization is lessened since the domain expert can target a single, simple procedural abstract interface, without needing to differentiate between C functions and C++ methods and without concern for the vagaries of C++. Finally, attributing semantics to abstractions, rather than to implementations, relieves developers of the tedium of specifying semantics for each implementation. Thus, abstraction semantics specifications may be reused across implementations and applications; the developer need only provide a new mapping between a concrete implementation and the abstraction interface it fulfills. There is a long history of using such isomorphisms to transform a problem from a concrete space to an abstract space within which it may be more readily analyzed

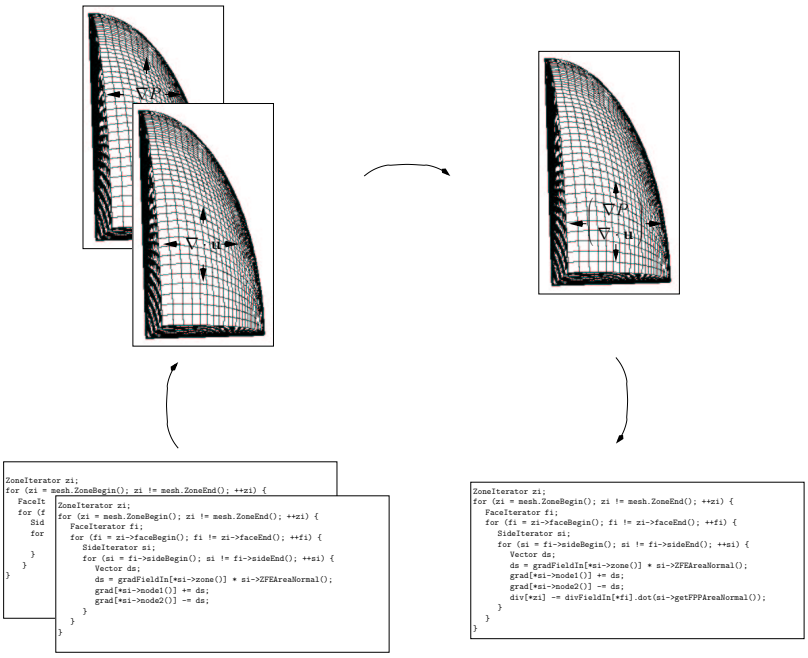

Figure 1.2: Semantics-based approach to abstraction optimization.

and transformed; for example, register allocation can be couched in terms of graph coloring [12] and iteration space transformations can be expressed with polyhedral algebra [48].

The abstract syntax is described by an interface specified by an expert on a per-domain basis. In a mesh context, e.g., it contains procedures to retrieve mesh entities and to access data stored in the mesh. The expert also provides the lowering operators, which map each procedure in the interface to its concrete implementation within the application. The framework automatically inverts lowering operators to define raising operators. Each abstraction interface effectively defines a set of semantics and operations that must be realized by any of its implementations. Since the specifications are written in C/C++, the domain expert is freed from having to learn yet another annotation or "little" language. However, the system does not require that an implementation mimic the style or form of the abstraction. In particular, a concrete procedure may be implemented as a C++

method, a C function, or a built-in expression, though the abstract procedure is always a function. Second, the number and order of formal parameters of the abstract and concrete procedures need not coincide: a novel data-flow analysis infers implicit arguments based on annotations from the domain expert.

1.4 Contributions

The primary contribution of this thesis is a framework for optimizing abstractions that targets their semantics rather than their implementations. This differs from previous approaches that target implementations (by annotating them with semantics) or that expect the application to be written in standardized, high-level interface. We instead project an implementation into an abstract form, whose semantics are understood by the domain expert. To evaluate the framework, we apply it to the unstructured mesh domain, though the approach is in no way specific to this area.

This thesis makes the following specific contributions:

1. **We quantify overhead in a representative unstructured mesh library.** Unstructured meshes play a critical role in the solution of partial differential equations over complex geometries within the Department of Energy. Chapter 2 provides additional detail on the application and organization of unstructured mesh frameworks and describes the `KOLAH` library for benchmarking C++ numerical methods on arbitrary polygonal and polyhedral meshes as a specific instance of such a framework. We quantify the overhead in `KOLAH` attributable to memory accesses; given our experience with unstructured grid technology, we believe that `KOLAH` is not exceptional

in its high ratio of mesh overhead to computation, but is an exemplar of object-oriented mesh-based libraries.

2. **We identify barriers to compiler optimization.** The pointer-rich abstractions used in meshes present barriers to compiler analysis. In Chapter 3, we discuss how semantic knowledge overcomes these barriers and would enable automation of optimizations targeting mesh overheads. We subsequently consider a series of optimizations and apply them manually.

3. **We identify traditional compiler optimizations important to mesh-based codes.** We show that two well-known optimizations—lazy evaluation and data packing—are significant in reducing memory and instruction overhead to expose the floating-point computation. Though data packing has been applied in an unstructured mesh domain [22], we believe that `KOLAH` better represents production applications than those previously considered.

4. **We introduce novel optimizations targeting unstructured meshes.** Our observed mesh overheads motivate a series of novel optimizations that improve computational intensity and overall performance:

    - *Iteration-space narrowing:* We propose extracting side-effect free function calls from a loop if they are executed repeatedly with the same inputs. Memoizing the results in a new iteration space then avoids this repetitive execution.
    - *Mesh precomputation:* Physical quantities, such as volumes, change every time step, but mesh connectivity information, such as the list of mesh edges, does not. Computing these static quantities once during an *inspector phase* and storing them for subsequent access during *executor phases* relieves the code from recomputing them at each time step.

- *Lowering:* We translate a computationally inefficient, object-oriented mesh framework, which uses C++ objects to access the mesh and iterate over mesh entities, to a low-level, imperative implementation, which uses integer-based access and iteration.

Though we apply these optimizations to the `testhydro1` benchmark written using the `KOLAH` library, they should be generally applicable to any unstructured mesh application in which abstractions implement iteration over the mesh and access to data stored in fields through the use of pointer and method indirection that the compiler is unable to optimize away. Insofar as any computation over a volume or surface requires iterating over the mesh to access data stored within it, these optimizations target the two fundamental mesh facilities that should be prevalent in any mesh solver. Further, mesh precomputation is not limited to the specific gradient and divergence operations targeted here, but is applicable to any nested loop over mesh elements with similar loop exit conditions, including the curl and averaging operators defined in libraries more sophisticated than `KOLAH`.

5. **We propose a framework for optimizing abstractions based on the projection between implementation and abstraction spaces.** The framework, introduced in Chapter 4, includes a novel data-flow analysis for coping with potential interface inconsistencies between the two spaces. The framework is evaluated by using it to automate the mesh precomputation and lowering optimizations. The relation between this framework and related projects is discussed in Chapter 5, as are related unstructured mesh optimizations.

Chapter 2

An Unstructured Mesh Framework

This chapter provides background on meshes and introduces `KOLAH`, the representative mesh library studied in this thesis. Using a `KOLAH` code fragment, it continues by enumerating specific barriers to optimization, which will be overcome in Chapter 3 through the use of domain-specific semantics.

2.1 Mesh Background

Meshes are a popular means of discretizing a continuous domain and are employed to do so across a broad range of disciplines, from texture mapping to computational fluid dynamics. This work examines meshes within the context of scientific computing where they are the basis for solving partial differential equations that model the dynamics of physical systems. Continuous (partial) derivatives can be approximated as finite differences, which are sampled at discrete points described by mesh elements. The following discussion of meshes follows Kirk [47].

The most fundamental concept within a mesh is a node, or a physical location in space to which higher-dimensional mesh elements are connected. Nodes are generally associated with a unique identifier. Standard elements include zones, faces, edges, and nodes, in decreasing dimensionality from three to zero. Elements may contain sub-elements: for example, an edge contains two nodes and a face is outlined by some number of edges. Given an element, a mesh interface provides a means of extracting its neighbors.

We differentiate between structured and unstructured meshes, though the taxonomy is not so simple and includes overset and "Dragon" grids [47]. Structured meshes correspond to a regular decomposition of a two- or three-dimensional space. This regularity implies that an element's neighbors can be implicitly inferred from its identifier. For example, a simple affine relation on an identifier can efficiently identify an element's neighbors according to a four-point stencil in two dimensions or an six-point stencil in three dimensions. No simple map exists between an element and its neighbors in an unstructured mesh. Instead, an element must explicitly maintain its connectivity information, generally in the form of a list.

The explicit managing of connectivity in unstructured meshes leads to considerable source code complexity and run-time overhead. For example, by one estimate, a 125,000 node mesh with 117,649 hexahedral elements requires 3MB of storage for the nodes but an additional 3.76MB of storage for the connectivity [47]. More significantly, the irregular geometry and neighbor relations of unstructured meshes induce irregular, non-strided memory accesses. Loops implementing stencil computations in structured meshes have affine loop bounds and array accesses; the loops may be readily parallelized and memory accesses within them may be statically predicted and prefetched. Unstructured mesh codes do not enjoy this simple loop structure; cache hit rates suffer as a result.

Despite their additional complexity, unstructured meshes are advantageous for many applications. They are easier to generate than structured meshes because they place fewer constraints on the mesh. They also allow for a richer variety of mesh element types. Unstructured meshes simplify refinement, the process of improving the quality of a mesh by replacing a (section of a) mesh with a finer mesh with more elements. Local refinement is complicated in structured meshes

because large swathes of the mesh must be replaced to conform to the imposed structure. Finally, unstructured meshes can dramatically reduce the number of elements because the mesh resolution may vary over the course of the mesh according to the demands of the physical system. For example, Shewchuk considers a fluid flow problem in which smaller elements are required within the volume experiencing turbulence than in that part of the system that does not [78]. Within a structured mesh, the mesh resolution would be dictated by the requirements of the finest scale, which leads to more mesh elements, more spatial overhead, and more computation time.

2.2 KOLAH

KOLAH is one of a number of libraries [8, 60] providing mesh element abstractions and the management of their connectivity. In addition, KOLAH scalar and vector field abstractions sample continuous physical quantities at discrete locations corresponding to mesh elements. These capabilities facilitate benchmarking of C++ numerical methods on arbitrary polygonal and polyhedral meshes. KOLAH's design was motivated by the classes and patterns used in production codes at Lawrence Livermore National Laboratory. It relies upon a generic mesh interface and provides reference mesh implementations along with a variety of mesh utility functions. `testhydro1` is a benchmark using these facilities to solve the Euler equations using the Lagrangian method and an ideal gas law equation of state.

Numerical algorithms are written to KOLAH's generic mesh interface, enabling underlying mesh implementations to be evaluated without rewriting the algorithm for each mesh instance. KOLAH interprets a variety of input mesh specifications,

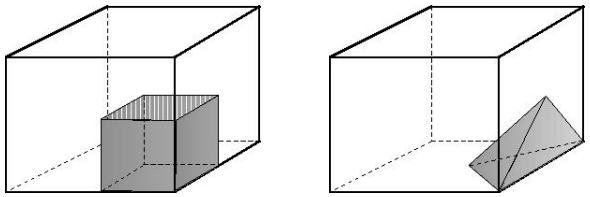

Figure 2.1: Mesh elements: corner (left) and side (right) within a zone.

```
for(ZoneIterator zi = mesh.zoneBegin();
    zi != mesh.zoneEnd(); ++zi)
{
   double tmp = P[*zi] * div[*zi] * zi->volume()
      - zoneHeating[*zi];
   e[*zi] = e[*zi] - dt * tmp / mass[*zi];
}
```

Figure 2.2: Zone iteration and mesh element-based field accesses using KOLAH's interface.

representing each as a class. After reading the input mesh, a compatibility layer converts and copies data from the underlying mesh implementation to a common mesh form.

The generic polyhedral mesh interface provides geometric mesh elements including zones, sides, faces, corners, edges, and nodes. All but sides and corners follow the standard terminology. Figure 2.1 shows a zone as it would appear in a rectilinear mesh, along with a representative corner and side. In general, a zone is a three-dimensional subvolume used to partition the volume discretized by the mesh; it needn't be a cube. Faces are two-dimensional elements that cover a zone's surface. A zone volume is itself subdivided into three-dimensional corners. A corner corresponds to each zone node and also has as vertices the zone center, the face centers of all faces containing the node, and the edge centers of all edges containing the node. A zone volume may also be subdivided into three-dimensional sides, whose vertices are two orthogonal face centers and the two nodes shared by those faces.

The mesh interface maintains connectivity information, allowing accesses via STL-like iterators, as shown by the code in Figure 2.2 that iterates over zones. Mesh element abstractions provide similar iterator access to neighboring elements. For example, given a node, one can iterate over all zones containing it using meta-data contained in the node object. This fairly complete connectivity information provides great flexibility in writing numerical algorithms. The figure also shows the mesh element-based field access paradigm popular in `KOLAH`: it dereferences a `ZoneIterator` to obtain a zone element and uses that element to index a field, such as the pressure field `P`.

With their heavy use of indirect addressing and pointer chasing, already evident in the simple loop of Figure 2.2, unstructured grid codes are highly sensitive to memory performance [32]. Abstraction-oriented implementations exacerbate poor memory performance through the additional indirection induced by mesh element-based indexing of fields, such as momentum and pressure: where an imperative approach uses a `for` loop with an integer induction variable to access an array, these codes dereference mesh element iterators and then use the element to index into field abstractions.

2.3 Barriers to Optimization

Because user-defined abstractions, such as STL-based iterators and mesh elements, are not part of the base language, compilers do not recognize them and so are not aware of their high-level properties. Instead compilers rely on a myopic approach that cobbles together alias and side-effect information on those parts of the abstraction that are defined in the base language. In some cases this task is futile

because source code is not distributed with libraries implementing the abstractions. When source code is available, a potentially costly global analysis may be needed to examine both the use of the abstractions and their implementation. Finally, without any *a priori* understanding of abstractions, compiler analysis is often too conservative to apply transformations to them.

The simple loop shown in Figure 2.2 highlights both the need for optimization in mesh-based applications and the inherent difficulties a compiler faces in performing those optimizations. This section shows that traditional compiler analyses are inadequate to determine the correctness of applying common subexpression elimination and iteration-space reordering to this loop. We discuss simple semantics of mesh abstractions in general, rather than of `KOLAH` in particular, that would enable these optimizations.

Though the repeated dereferences of the `ZoneIterator` in the loop seem good candidates for common subexpression elimination, the `KCC` compiler does not perform the optimization because it can not determine that `operator[]` is side-effect free. `getID`, one of the methods invoked during the object-based field reference, is neither declared `inline` nor implemented in the header file; as such, it is bypassed by `KCC`'s aggressive inlining. Unable to inline `operator[]` completely, the compiler cannot analyze its implementation at the callsite and must conservatively assume it generates side effects that potentially modify the common subexpression `*zi`.

A number of remedies would enable common subexpression elimination. Moving the implementation of `getID` to its header file would allow inlining and would overcome the immediate barrier to applying the optimization. Annotating `getID`'s prototype with a declaration that it is side-effect free would have the same effect. However, these approaches are intimately tied to the implementation of field ad-

dressing in KOLAH. Other mesh libraries are likely to offer the same, side-effect free style of access, but are unlikely to employ getID in doing so; the tedious, iterative round of discovering and annotating any functions that cannot be inlined will need to be done anew for each application.

Our solution recognizes fields as an abstraction common across mesh libraries. Doing so allows us to imbue them with semantics that carry over from one implementation to another: mesh element-based indexing, the overloaded indexing of a field with a mesh element, is a pure operation—side-effect free and dependent only on its mesh element argument. This expressive statement would enable common subexpression elimination since the compiler could be confident nothing within the loop modifies the ZoneIterator.[1] The approach taken in Chapter 4 differs somewhat in that abstractions are not explicitly annotated with their semantics. Rather, for the purposes of our prototype, the semantics of an abstraction are described in its documentation. The expert then guarantees that an implementation conforms to an abstraction's semantics before mapping an implementation as one of its concrete instances. These semantics could be codified in annotations that are parsed by a compiler framework.

Nested loops in testhydro1 access zones in a non-strided manner. Packing the zones can mitigate the effects of such accesses by rearranging their memory layout. Unfortunately, reordering data to benefit nested loops leads to computational reordering of the loop in Figure 2.2. In order to reorder this loop safely, a compiler must disambiguate the loop's pointers to guarantee that there are no loop-carried dependences. If (as is likely) alias analysis is unable to determine the uniqueness of each element in the iteration space, the compiler would have to

[1]Type-based alias analysis could guarantee that the write to the field does not modify the ZoneIterator.

assume the iterator is non-trivial and can repeatedly access a zone to create a flow dependence on `e`. Such a dependence would prevent loop reordering. Fortunately, the simple assertion that mesh iterators do not revisit elements ensures there is no such dependence, since each loop instance accesses a unique zone. These semantics would complement side-effect analysis, allowing it to determine that the loop could be reordered. Such semantic assertions on an iterator are useful in general, beyond the specific example of mesh iteration, and may be used to facilitate reordering and parallelism [49] in cases where alias and side-effect analyses are insufficient to infer their safe application.

Chapter 3

Mesh Optimizations

This chapter quantifies the memory overheads of `testhydro1`, which are likely to plague other mesh applications and libraries employing high-level abstractions. The platforms on which `testhydro1` is executed are described in Section 3.1, with the resulting performance characteristics reported in Section 3.2. The few mesh semantics of Section 3.3 may be leveraged to ameliorate `testhydro1`'s poor computational intensity through the series of manual optimizations discussed in Section 3.4. These semantics ensure the correctness of traditional optimizations, such as lazy evaluation. Further, they suggest domain-specific optimizations, such as iteration-space narrowing, mesh precomputation, lowering, and the previously investigated data packing [22].

3.1 Methodology

We apply optimizations to the computational core of the `KOLAH`-based hydrodynamics benchmark `testhydro1`. Results are reported as averages over five runs of ten time steps. Standard deviations were within one percent of the averages and are not shown. Due to the duration of production runs, we do not account for application, mesh, or optimization configuration time, which would be amortized over many time steps. The input data set, `ellipsoid`, is the largest provided with `KOLAH`, with 70K zones, 383K faces, 530K corners, 1.5M sides, 195K edges, and 66K nodes.

`testhydro1` was executed on the IBM POWER5 platform described in Table 3.1 and the Intel Xeon platform described in Table 3.2. In addition, the POWER5 and Xeon have similar hardware-based prefetch engines that detect sequential (or strided, in the Xeon) accesses and prefetch data following successive misses. Each processor supports multiple, simultaneously-active prefetch streams. Performance metrics on the POWER5 were collected using the IBM HPM hardware performance monitor Tool Kit [39], while wall clock execution times are reported on the Xeon. Table 3.3 lists the compilers used on the two platforms. On the POWER5, `testhydro1` was compiled with the `KCC` front-end optimizing compiler and passed the `+K3` optimization flag to instruct it to perform branch simplification, loop unrolling, small object optimization, and function inlining. `KCC` produces intermediate C code that is compiled by IBM's `xlc` back-end compiler. We pass `xlc -O3`, as well as `strict` to ensure the safety of applied optimizations, `arch=pwr5` to enable POWER5-specific optimizations, and `ignerrno` to allow the compiler to emit the `sqrt` instruction. `gcc` is used on the POWER5 and Xeon architectures and is passed the `-O3` optimization flag in both instances. In what follows, we refer to the various combinations of platform and compiler as `KCC`/POWER5, `gcc`/POWER5, and `gcc`/Xeon.

3.2 Mesh Overheads

The results for the unoptimized, baseline version of `testhydro1` are shown in Tables 3.4, 3.7, and 3.9 for `KCC`/POWER5, `gcc`/POWER5, and `gcc`/Xeon, respectively. Metrics describing instruction mixes indicate the number of *dynamic* instructions executed, rather than the number of *static* instructions in the program text. Since the POWER5 and Xeon are out-of-order, speculative processors, the

Table 3.1: IBM POWER5 hardware specification.

Processor	1.9 GHz POWER5 p5 575
Integer Units	2
Floating-point Units	2
Peak IPC	5
Peak Gflops (Per Core)	7.6
Registers	32
Data TLB	128 entries
Data TLB Associativity	128-way
L1 Data Cache Size	32 KB
L1 Instr Cache Size	64 KB
L1 Data Cache Latency	2 cycles
L1 Data Cache Line Size	128 B
L1 Data Cache Associativity	4-way
L2 Size	1.9 MB (per processor pair)
L2 Latency	10 cycles
L2 Line Size	128 B
L2 Associativity	10-way
L3 Size	36 MB (per processor pair)
L3 Latency	90 cycles
L3 Line Size	256 B
L3 Associativity	12-way
Memory	32 GB
Memory Latency	220 cycles

Table 3.2: Intel Xeon hardware specification.

Processor	3.46 GHz Pentium D Xeon
Integer Units (Multiple Function)	2
Floating-point Units (Multiple Function)	1
Peak μop/cycle	3
Peak Floating-point IPC	1
Peak Gflops	3.46
Integer Registers	128
Floating-point Registers	128
Data TLB	64 entries
Data TLB Associativity	64-way
L1 Data Cache Size	16 KB
L1 Instr Cache Size	12K μop
L1 Data Cache Integer Latency	4 cycles
L1 Data Cache Floating-point Latency	12 cycles
L1 Data Cache Line Size	64 B
L1 Data Cache Associativity	8-way
L2 Size	2 MB
L2 Integer Latency	20 cycles
L2 Floating-point Latency	20 cycles
L2 Line Size	64 B
L2 Associativity	8-way
Memory	4 GB

Table 3.3: Compiler specifications.

Compiler	Platform	Version	Options
KCC/xlc	POWER5	4.0 (KCC); 9.0.0.1 (xlc)	+K3 -O3 -qignerrno -qstrict -qarch=pwr5
gcc	POWER5	3.3.2	-O3
gcc	Xeon	4.1.2	-O3

number of dynamic instructions will likely be greater than that which would have resulted from execution on a non-speculative processor. For example, "L1 cache accesses" represents the total number of dynamic loads and stores *issued*, including those executed down a speculated path that was later squashed.

Tables 3.4 and 3.7 indicate the relative paucity of floating point instructions. On `KCC`/POWER5, 13.9 loads or stores and 3.8 branches are executed per floating point operation (FLOP). Under `gcc`/POWER5, there are 8.2 loads or stores and 2.3 branches per FLOP, respectively. As scientific computations are carried out in floating point arithmetic, the large relative contribution of memory accesses to the instruction mix strongly suggest that `testhydro1` expends considerable execution time accessing operands and little execution time computing with them.

As execution is a function of the input mesh, we examine the performance profile of `testhydro1` when run on the `10x10x3` mesh. The simulation also uses suitable initial and boundary conditions and executes for 4000 iterations, but is otherwise identical to the `ellipsoid` run. The `10x10x3` mesh is a rectangular decomposition and, with only 620 zones, 2120 faces, 3680 corners, 2120 sides, 1243 edges, and 484 nodes, is significantly smaller than `ellipsoid`. Nevertheless, the table shows that it induces similar overheads—with ratios of 13.5 loads or stores and 3.7 branches per floating point operation under `KCC`/POWER5. In general, the overhead is tied intimately to the unstructured *representation* of the mesh within `KOLAH`, however regular the mesh decomposition itself may be. It will also be influenced by the degree of connectivity of the mesh elements, a factor

most influenced by the dimensionality of the problem. Therefore, we expect the (fractional) overheads to be largely mesh independent.

This significant overhead may seem surprising given the efficiency of STL and similar libraries that are the foundations of `KOLAH`. Though the implementation of underlying abstractions is highly tuned, optimizing across library invocations requires contextual information unavailable during library development. Even after inlining abstractions to view this context, compiler analysis is often too conservative to certify the safety of many relevant optimizations, such as loop fusion [73]. Our remedy is not to pursue more sophisticated pointer alias analysis, but to exploit higher-level, expressive semantic information. High-level knowledge of an iterator, for example, can transform it from an unintelligible collection of pointers to a well-defined abstraction with specific semantics that complement traditional, fine-grained analysis [91].

Comparison of Tables 3.4 and 3.7 reveals that `KCC` is better able to optimize `testhydro1` than `gcc` on the POWER5. For example, when compiled with `gcc`, `testhydro1` performs 1.6× more L1 cache accesses and 1.5× more L2 cache accesses, executes 1.6× more branches, 2.6× more FLOPS, and 1.4× more instructions, and runs 1.4× longer.

Table 3.4: Performance metrics for 10 iterations of `testhydro1` on POWER5 compiled with KCC. Values may be reported in millions (M) or billions (B) of instances.

Metric	Baseline	Lazy evaluation	Iteration-space narrowing	Mesh precomputation	Lowering	Data packing	Iteration-space partitioning	Multiple-constraint reordering
L1 cache accesses (B)	129.23	116.62	32.92	22.41	21.12	21.12	18.37	20.88
L2 cache accesses (B)	3.99	3.77	1.82	1.26	1.32	1.17	1.26	1.38
L2 traffic (GBytes)	453.82	428.82	206.68	143.16	149.73	132.88	142.84	156.81
Memory bandwidth (GBytes/s)	22.96	24.66	45.57	52.24	53.30	47.92	61.64	50.50
DTLB misses (M)	28.08	27.99	25.79	23.96	23.10	22.91	24.82	23.75
Branches (M)	35055.17	31543.76	8971.01	6353.66	5869.15	5875.31	5016.33	5759.07
Unconditional branches (M)	14029.10	12654.39	3708.27	2341.07	2086.00	2082.15	1688.48	2039.55
Mispredicted branch direction (M)	400.87	366.99	261.41	217.03	213.23	214.53	125.98	204.98
Mispredicted branch target (M)	951.98	840.92	66.17	64.67	58.24	59.15	48.01	57.90
Flops (B)	9.33	8.82	6.98	6.98	6.99	7.00	6.45	6.94
Computational intensity	0.07	0.08	0.21	0.31	0.33	0.33	0.35	0.33
Instr completed (B)	249.97	225.82	66.46	45.80	43.21	43.22	38.35	42.96
Wall clock time (sec)	154.89	143.82	70.70	55.38	53.48	48.93	65.30	50.77

Table 3.5: Performance metrics for 4000 iterations of `testhydro1` using `10x10x3` mesh on POWER5 compiled with KCC.

Metric	Baseline
L1 cache accesses (B)	117.96
L2 cache accesses (B)	4.43
L2 traffic (GBytes)	503.78
Memory bandwidth (MBytes/s)	0.04
DTLB misses (M)	6.32
Branches (M)	31974.95
Unconditional branches (M)	12876.88
Mispredicted branch direction (M)	430.06
Mispredicted branch target (M)	843.29
Flops (B)	8.71
Computational intensity	0.07
Instr completed (B)	225.75
Wall clock time (sec)	111.12

Table 3.6: Fractional improvement over baseline of performance metrics for 10 iterations of `testhydro1` on POWER5 compiled with KCC.

Metric	Lazy evaluation	Iteration-space narrowing	Mesh precomputation	Lowering	Data packing	Iteration-space partitioning	Multiple-constraint reordering
L1 cache accesses	1.11	3.92	5.77	6.12	6.12	7.03	6.19
L2 cache accesses	1.06	2.20	3.17	3.03	3.41	3.18	2.89
L2 traffic	1.06	2.20	3.17	3.03	3.42	3.18	2.89
Memory bandwidth	1.07	1.98	2.28	2.32	2.09	2.69	2.20
DTLB misses	1.00	1.09	1.17	1.22	1.23	1.13	1.18
Branches	1.11	3.91	5.52	5.97	5.97	6.99	6.09
Unconditional branches	1.11	3.78	5.99	6.73	6.74	8.31	6.88
Mispredicted branch direction	1.09	1.53	1.85	1.88	1.87	3.18	1.96
Mispredicted branch target	1.13	14.39	14.72	16.34	16.09	19.83	16.44
Flops	1.06	1.34	1.34	1.33	1.33	1.45	1.34
Computational intensity	1.06	2.94	4.33	4.60	4.61	4.88	4.61
Instr completed	1.11	3.76	5.46	5.79	5.78	6.52	5.82
Wall clock time	1.08	2.19	2.80	2.90	3.17	2.37	3.05

Table 3.7: Performance metrics for 10 iterations of `testhydro1` on POWER5 compiled with gcc.

Metric	Baseline	Lazy evaluation	Iteration-space narrowing	Mesh precomputation	Lowering	Data packing	Iteration-space partitioning	Multiple-constraint reordering
L1 cache accesses (B)	201.61	185.36	64.13	39.58	35.71	34.89	29.85	34.86
L2 cache accesses (B)	6.16	3.83	1.62	1.47	1.67	1.50	1.49	1.45
L2 traffic (GBytes)	700.42	435.42	184.33	167.35	189.38	170.02	168.78	164.28
Memory bandwidth (GBytes/s)	14.11	15.15	31.46	36.96	37.65	36.09	49.04	36.19
DTLB misses (M)	25.22	25.04	23.47	21.47	20.99	20.81	22.84	20.88
Branches (M)	56922.02	52112.92	17916.76	11002.17	9042.65	8920.95	7725.46	8836.67
Unconditional branches (M)	25275.41	23041.30	7297.77	4233.33	3428.59	3382.88	2883.89	3362.20
Mispredicted branch direction (M)	454.06	441.06	297.08	229.00	253.57	253.69	232.97	231.17
Mispredicted branch target (M)	931.92	821.79	168.47	149.42	116.73	118.06	88.77	123.65
Flops (B)	24.54	23.26	13.93	12.16	11.81	11.76	10.41	11.78
Computational intensity	0.12	0.12	0.22	0.31	0.33	0.34	0.35	0.34
Instr completed (B)	354.64	322.42	118.07	73.13	63.92	63.92	54.74	63.51
Wall clock time (sec)	219.56	204.69	97.40	71.75	68.14	63.23	78.98	65.48

Table 3.8: Fractional improvement over baseline of performance metrics for 10 iterations of `testhydro1` on POWER5 compiled with gcc.

Metric	Lazy evaluation	Iteration-space narrowing	Mesh precomputation	Lowering	Data packing	Iteration-space partitioning	Multiple-constraint reordering
L1 cache accesses	1.09	3.14	5.09	5.65	5.78	6.75	5.78
L2 cache accesses	1.61	3.80	4.19	3.70	4.12	4.15	4.26
L2 traffic	1.61	3.80	4.19	3.70	4.12	4.15	4.26
Memory bandwidth	1.07	2.23	2.62	2.67	2.56	3.47	2.56
DTLB misses	1.01	1.07	1.17	1.20	1.21	1.10	1.21
Branches	1.09	3.18	5.17	6.29	6.38	7.37	6.44
Unconditional branches	1.10	3.46	5.97	7.37	7.47	8.76	7.52
Mispredicted branch direction	1.03	1.53	1.98	1.79	1.79	1.95	1.96
Mispredicted branch target	1.13	5.53	6.24	7.98	7.89	10.50	7.54
Flops	1.05	1.76	2.02	2.08	2.09	2.36	2.08
Computational intensity	1.02	1.78	2.52	2.71	2.76	2.86	2.77
Instr completed	1.10	3.00	4.85	5.55	5.55	6.48	5.58
Wall clock time	1.07	2.25	3.06	3.22	3.47	2.78	3.35

Table 3.9: Elapsed time for 10 iterations of `testhydro1` on Xeon compiled with gcc.

Metric	Baseline	Lazy evaluation	Iteration-space narrowing	Mesh precom-putation	Lowering	Data packing	Iteration-space partition-ing	Multiple-constraint reordering
Wall clock time (sec)	116.65	107.69	55.59	45.07	43.38	39.71	42.99	40.21

Table 3.10: Fractional improvement over baseline of elapsed time for 10 iterations of `testhydro1` on Xeon compiled with gcc.

Metric	Lazy evaluation	Iteration-space narrowing	Mesh precom-putation	Lowering	Data packing	Iteration-space partition-ing	Multiple-constraint reordering
Wall clock time	1.08	2.10	2.59	2.69	2.94	2.71	2.90

3.3 Mesh Semantics

We leverage the following semantics of `KOLAH`, which we anticipate will hold for unstructured meshes in general, to manually introduce safe optimizations that ameliorate mesh-induced overhead:

- Mesh element-based field indexing is a pure function—side-effect free and dependent only on its explicit actual arguments;
- Elements enumerated by a mesh iterator are unique;
- The mesh (structure) is updated infrequently.

3.4 Mesh Optimizations

This section highlights well-known optimizations from the literature—lazy evaluation and data packing—as well as novel, domain-specific optimizations that significantly improve the performance of `testhydro1` and that we believe are relevant to mesh applications in general. Code elimination optimizations reduce dynamic instruction counts by ensuring that calculations are not performed unnecessarily (i.e., if they are never used) or redundantly. These include, lazy evaluation, memoization of calculations dependent on mesh elements, and precomputation of mesh connectivity metadata. The lowering optimization transforms high-level mesh abstractions to more efficient, but less readable, low-level code. Finally, we consider three types of domain-specific data and computation reordering transformations.

The effects of the optimizations are shown in Tables 3.6, 3.8, and 3.10, which normalize the results of Tables 3.4, 3.7, and 3.9 to the unoptimized baseline

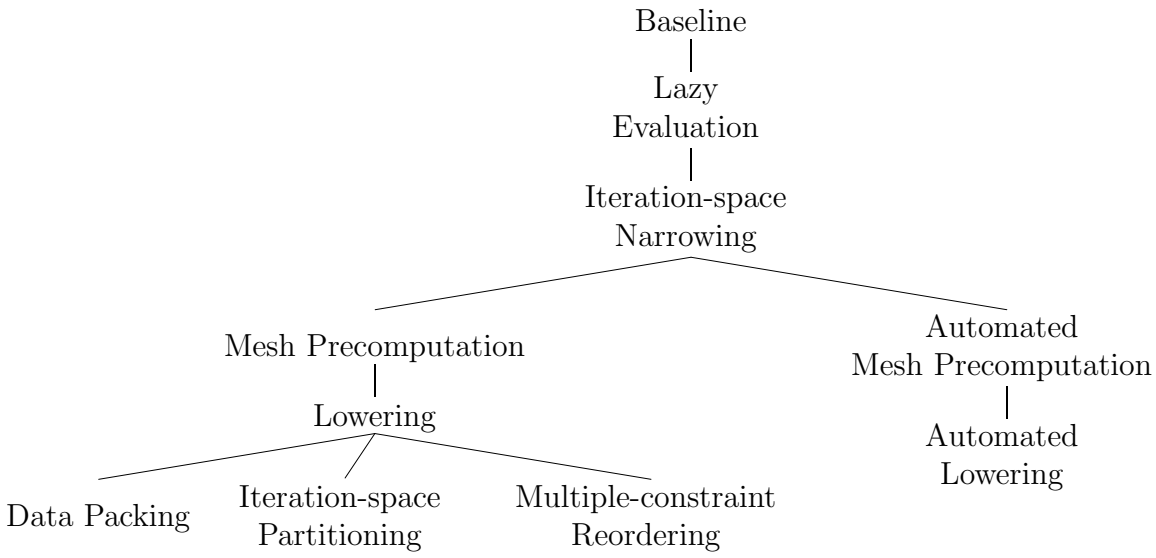

Figure 3.1: Application order of optimizations. An optimization named in a parent node is applied before that named in the child node. Optimizations in sibling nodes are independently applied following the optimization named by the parent node.

for `KCC`/POWER5, `gcc`/POWER5, and `gcc`/Xeon, respectively. Tables 3.6, 3.8, and 3.10 show fractional improvement over the baseline, with one indicating no change and a number larger than one indicating an improvement. The optimizations are manually applied in the order described by the tree in Figure 3.1. An optimization named at a tree node is applied before its children. Optimizations named in sibling nodes are alternate optimizations that are applied independently to the parent. For example, data packing, iteration-space partitioning, and multiple-constraint reordering, are three different packing strategies that are applied following the lowering optimization. Two of the optimizations—mesh precomputation and lowering—are automated, as discussed in Chapter 4.

3.4.1 Code Elimination

`KOLAH`'s design stresses flexibility over performance: its interface facilitates nu-

merical programming by providing convenient access to mesh entities, but also encourages unnecessary and redundant computation. The following optimizations transform application structure in a way that reduces such superfluous computation: code motion of statements to the single branch of a conditional in which they are active avoids their unnecessary execution when the other branch is taken; narrowing an iteration space to a unique set of mesh elements avoids redundant computation of a pure function dependent on those elements; and precomputing static mesh connectivity information avoids having to re-evaluate it during each mesh traversal.

Lazy Evaluation

Lazy evaluation is a traditional technique to reduce the number of instructions between a definition and use. When this involves code motion to a single branch of the condition, it effectively removes a definition that would not be used within the alternate branch. This section applies lazy evaluation to the "Caramana loop" of Figure 3.2, which calculates an artificial viscosity and is one of several dominant loops in the `testhydro1` time step. Though lazy evaluation is well-known and implemented within standard compilers, applying it to this loop is prevented by conservative analysis, which must assume that the mesh element-based field indexing executed within the loop induces side effects. Nevertheless, mesh semantics ensure that this is a pure operation: domain-specific knowledge thus enables a traditional optimization.

The ellipsis on line 35 of the Caramana loop abstracts 23 statements, including memory accesses, a conditional, method invocations, and long-latency square root instructions. Though these are dependents of the `doPhysics` function invoked

```
  for (SideIterator sideIt = mesh->sideBegin();
 2     sideIt != mesh->sideEnd(); ++sideIt) {
     // extract the edge from the side
 4   EdgeIterator edgeIt = sideIt->edge();

 6   // extract the nodes from this edge
     NodeIterator node1It = edgeIt->node1();
 8   NodeIterator node2It = edgeIt->node2();

10   // calculate change in position & sign
     const Vector &S = sideIt->ZFEAreaNormal();
12   Vector deltaX = node1It->position()
        - node2It->position();
14   int sign = (S.dot(deltaX) < 0) ? -1 : 1;

16   // calculate change in velocity
     Vector deltaV = velocity[*node1It]
18      - velocity[*node2It];

20   rho1 = volumeWeightedAvg(zoneMass,node1It);
     rho2 = volumeWeightedAvg(zoneMass,node2It);
22
     ZoneIterator zoneIt = sideIt->zone();
24   soundSpeed = sqrt( inGamma * pressure[*zoneIt] /
        max(rho1, rho2) );
26
     map<int,<pair<int,int> >::iterator mapIt =
28      edgeMap.find(edgeIt->getID());

30   int leftEdgeIndex = (*mapIt).second.first;
     if( leftEdgeIndex > -1 ) {
32      // ...
     }
34
     // ... mem ops, method calls, sqrts, etc.
36
     int signDotProd = sign * S.dot(deltaV);
38   if (signDotProd < 0.0) {
        edgeForcing = doPhysics(deltaV, ...);
40   } else {
        edgeForcing.Zero();
42   }

44   nodeForcing[*node1It] += edgeForcing;
     nodeForcing[*node2It] -= edgeForcing;
46 }
```

Figure 3.2: Iteration over sides.

within the true branch of the conditional on line 38, they do not contribute to its false branch. In fact, none of the statements following line 17 are used outside of the true branch. Therefore, in order to avoid unnecessary computation, these statements should be evaluated lazily—i.e., moved to the true branch of the conditional.

Performing lazy evaluation requires compiler analysis to determine that all moved statements are side-effect free. `KCC` inlines `volumeWeightedAvg`, so that the only obstacles are the overloaded indexing operator of line 24 and the invocation of STL's `find` on line 27, which can not be completely inlined to determine its side-effect behavior.[1] The semantics of mesh element-based field indexing and an additional annotation on `find` could instruct the compiler that both are side-effect free and that it may safely perform the optimization.

Of the 1,500,000 total iterations of the loop, lazy evaluation results in fewer dynamic instruction executions in the 183,000 iterations in which the conditional evaluates to false. As shown in Tables 3.6 and 3.8, the optimization leads to a reduction in dynamics instructions on the POWER5 of 9-10%. Computational intensity is largely unaffected, since lazy evaluation removes floating point instructions, as well as memory accesses and branches. Tables 3.6, 3.8, and 3.10 show that the effect on overall performance is a fractional improvement of 1.07-1.08 on all platforms.

[1]This optimization could change program behavior if `sqrt` generates an exception or if the memory access causes a segmentation fault. Nevertheless, because the optimization doesn't *introduce* any potential exceptions, it is safe.

Iteration-space Narrowing

The connectivity between KOLAH's mesh objects provides latitude in object traversal and algorithm design. Computation acting on all mesh entities of a given type, such as the code in Figure 2.2, is best implemented as a simple iteration over those elements. Other computation, including the program fragment of Figure 3.2, is a complex function of multiple iteration spaces—those ranging over zones, side, edges, and nodes. In such cases, the choice of iteration space or spaces is not obvious since one mesh entity domain can be reached from any other through their interconnections. This flexibility allows a programmer to implement all operations involving a logical computation within a single loop, rather than distributing them over multiple iteration spaces.

While such flexibility facilitates scientific programming, the resulting implementation is potentially inefficient. The lack of a bijection between mesh entity domains means that iteration spaces that uniquely visit sides, for example, may revisit any mesh entity they access from a side. This many-to-one and onto (surjective) mapping is a property of the mesh [69] and is exhibited by the code in Figure 3.2, which revisits nodes, since the same node may be associated with different sides. The original loop structure makes 3,000,000 invocations of `volumeWeightedAvg`, though only 66,351 of those invocations access unique nodes. Thus the vast majority of these calls incur the unnecessary loop and memory accesses of `volumeWeightedAvg`.

Iteration-space narrowing eliminates redundancy by extracting a pure function that is re-evaluated with the same arguments and executing it within an iteration space that uniquely visits those arguments. In the above example, iteration-space narrowing instantiates a loop that iterates over nodes, invokes `volumeWeightedAvg`

on each, and memoizes the results for subsequent access in the original loop. Memoizing results of `volumeWeightedAvg` within the original loop is less efficient, because it requires a conditional to check whether the result has already been calculated. Such branches degrade performance directly by introducing pipeline stalls and indirectly by complicating compiler- or hardware-directed prefetching.

Iteration-space narrowing is the most powerful optimization considered. By eliminating the many repeated invocations on `volumeWeightedAvg`, it provides a 2.7-3.5× reduction in executed instructions and memory accesses over the previous lazy evaluation optimization. The result is a doubling in performance.

The legality of the transformation follows directly from the mesh semantics used to facilitate lazy evaluation and to determine that loops are side-effect free. The profitability of the transformation could be gauged by inferring the presence of redundant execution based on a knowledge of mesh semantics. Recognizing that elements are being revisited requires characterizing the domain of a loop nest succinctly. Ahmed *et al.* [1] describe a statement iteration space, based on the loops surrounding a statement, that characterizes the dynamic instances of a statement as a set of points in an iteration space induced by affine functions of the loop indices. Since iterators introduce non-affine expressions into loops, this approach is not applicable. Strout *et al.* [82] avoid this problem by describing dependences using Presburger arithmetic with uninterpreted functions and resolving the dependences at run time. This inspector/executor-inspired approach [17] could also be used to determine whether a loop revisits entities by simply traversing the iteration space and keeping a record of any accessed element.

We propose a symbolic approach that codifies the relations between mesh entities, e.g., that an edge is associated with two nodes. This allows a compiler to

determine statically whether iteration-space narrowing is likely to be profitable. For example, given the previous assertion, a compiler could infer that a function on nodes invoked within a loop over edges would be repeatedly executed with the same arguments. If the compiler determines that the function is sufficiently complex and that its computation is side-effect free and independent of the surrounding loop, the compiler could hoist it from the loop and re-instantiate it within a loop that directly iterates over its unique arguments.

Mesh Precomputation

`testhydro1` frequently iterates over mesh elements simply to access connected elements. For example, the code for the gradient operator `grad` shown in Figure 3.3 iterates over zones in order to iterate over the zones' faces and then iterates over the faces' sides to finally access members of the sides. That is, only attributes of the sides contribute to the calculation accumulated in the `grad` field. Since the zones and faces are traversed solely to reach the sides, only their mesh connectivity metadata is examined and accesses to them represent pure overhead.[2] Fortunately, such overhead is avoidable.

Mesh structure is often static throughout an application's execution. Adaptive mesh-based schemes reconstitute a mesh automatically when accuracy falls to unacceptable levels, but also hold the mesh static across a large number of iterations. This static property allows an inspector phase [17] to evaluate and store mesh connectivity metadata prior to performing computation over the mesh, during application initialization or immediately after remeshing in an adaptive

[2] As accesses to an entity's (mesh connectivity) metadata may not be temporally related to accesses to its data (e.g., area normal, volume, etc.), it may be profitable to split the structure [15] into two separate structures—one holding the metadata and the other holding the data.

scheme. Subsequent execution by an executor then accesses the precomputed, stored values to avoid the iteration or pointer chasing needed to recompute static properties, including the mesh interconnectivity accessed by the gradient operator and the relationship of an edge to a side accessed by line 4 of Figure 3.2. Line 11 accesses the area normal, a dynamic property, of the side; such a result can not be precomputed since it potentially changes each iteration.

Figure 3.3 shows our intuitive approach to mesh precomputation. The inspector `gradInspector` mimics the original loop structure of `grad` to precompute and store for subsequent retrieval only those target mesh entities needed for the calculation, rather than those mesh entities that are merely traversed to reach the calculation's operands. In the case of `grad`, this requires storing in an STL vector the zone and nodes associated with a side, but does not require storing the face. The original three, perfectly nested loops are transformed into the single loop of the executor `gradExectuor`, which accesses the stored objects linearly from the vectors to avoid the loop and indirection overheads inherent in mesh traversal.

Mesh precomputation significantly reduces the number of static (and dynamic) instructions to achieve an overall 19-26% performance improvement across platforms. Since the optimization does not modify or remove any floating point calculations, but rather the means of accessing operands, no changes should be reported in floating point operations. This is indeed the case under the `KCC`/POWER5 platform. Surprisingly, however, the number of floating operations is reduced on the `gcc`/POWER5 platform. This likely occurs because `gcc` uses floating point operations in unexpected contexts, e.g., using floating point registers to copy data between memory locations. Nevertheless, under both compilers, 70-80% of the instructions removed are loads, stores, or branches.

```
void grad(Field& field, Mesh& mesh, Field& grad) {
   for (ZoneIterator zi = mesh.zoneBegin();
        zi != mesh.zoneEnd(); ++zi) {
      for (FaceIterator fi = zi->faceBegin();
           fi != zi->faceEnd(); ++fi) {
         for (SideIterator si = fi->sideBegin();
              si != fi->sideEnd(); ++si) {
            Vector ds = field[*si->zone()] * si->ZFEAreaNormal();
            grad[*si->node1()] += ds;
            grad[*si->node2()] -= ds;
         }
      }
   }
}

void gradInspector(Field& field, Mesh& mesh) {
   for (ZoneIterator zi = mesh.zoneBegin();
        zi != mesh.zoneEnd(); ++zi) {
      for (FaceIterator fi = zi->faceBegin();
           fi != zi->faceEnd(); ++fi) {
         for (SideIterator si = fi->sideBegin();
              si != fi->sideEnd(); ++si) {
            zoneIts.push_back(si->zone()->getID());
            sideIts.push_back(si);
            Vector ds = field[*si->zone()] * si->ZFEAreaNormal();
            node1Its.push_back(si->node1()->getID());
            grad[*si->node1()] += ds;
            node2Its.push_back(si->node2()->getID());
            grad[*si->node2()] -= ds;
         }
      }
   }
}

void gradExecutor(Field& field, Field& grad) {
   for (int i = 0; i < zoneIts.size(); ++i) {
     SideIterator si = sideIts[i];
     Vector ds = field[zoneIts[i]] * si->ZFEAreaNormal();
     grad[node1Its[i]] += ds;
     grad[node2Its[i]] -= ds;
   }
}
```

Figure 3.3: Gradient operators. Original gradient operator (top). Inspector code to precompute mesh connectivity of gradient operator (middle). Executor code to access precomputed mesh connectivity to implement gradient operator (bottom).

3.4.2 Lowering

`testhydro1` frequently dereferences iterators to access physical fields, as illustrated by the code fragments in Figures 2.2 and 3.2. The succinctness of mesh field accesses aids programmer productivity by hiding the complexity of the underlying implementation: mesh element-based indexing extracts an integer `MeshID` from the object through a series of method invocations that culminates in a call to `getID`. This integer ultimately indexes into a STL vector representing the field. Dereferencing an iterator to extract the mesh element involves three method invocations, evaluating a conditional, three member field accesses, and at least two pointer dereferences. Indexing with that element additionally leads to five method invocations, two pointer dereferences, and one array access.

Such code complexity frequently results from the fine-grained parameterization employed by STL and `KOLAH` to provide programming convenience. Inlining is crucial for ensuring that this convenience does not degrade performance: it removes function calls and exposes compilation opportunities that would have required interprocedural analysis. For example, `KCC` is able to simplify the for loop header of Figure 2.2 by replacing the `ZoneIterator` with an integer induction variable. However, inlining requires access to the complete definition of the callee when the call site is compiled. This condition is met (e.g., in STL) by providing definitions in header files. Unfortunately, relying on this approach is fragile as programmers can make subtle performance bugs by providing a definition instead in a separate module. Such is the case with `KOLAH`'s `getID` method, as discussed in Section 2.3. The compiler's subsequent inability to infer that `getID` is side-effect free prevents further simplification of the loop (e.g., hoisting of array base calculations past invocations of `getID` and out of the loop). Whole-program analysis, such as link-time

optimization, spans modules to overcome this problem, but may be infeasible for some applications due to its compile-time expense. These analyses are performed indiscriminately and globally: compilation effort may be expended without benefit. In other situations, source code may not be available for inlining if the callee definition is encapsulated in a library. Both of these challenges are addressed by an abstraction-based approach, which describes mesh element-based indexing as a pure function, independent of its definition and the location of that definition.

Either the semantics of mesh element-based indexing or the inlining of `getID` and the compiler's subsequent ability to infer that it is side-effect free allow the multiple invocations of `getID` to be eliminated by common subexpression elimination. Nevertheless, further optimization opportunities remain. Assuming the loop header has been simplified through inlining to use integer induction (as accomplished by `KCC`), the loop has an integer induction variable and an integer identifier returned by `getID`. However, the compiler would recognize no semantic relation between the two. Rather, the integer induction variable would be used to index the mesh in order to obtain a pointer to a zone. This zone pointer would then be dereferenced to access its identifier, which would finally be used to index the field. This effective translation between the integer induction variable and the integer identifier thus involves two loads. The translation may be removed by iterating over the space defined by the identifiers rather than the space spanned by the induction variable. Changing the iteration order in this manner would seem to violate flow dependences on `e`, as discussed in Section 2.3. Fortunately, the semantics of mesh iterators guarantee that they do not revisit elements. Therefore, each loop instance accesses a unique element and there are no loop-carried dependences.

Consideration of the semantics of mesh element-based indexing and mesh iter-

```
int numZones = mesh.numberOfZones();
for(int zi = 0; zi < numZones; ++zi)
{
   Zone *zone = mesh.getZones()[zi];
   double tmp = P[zi] * div[zi] * zone->volume()
      - zoneHeating[zi];
   e[zi] = e[zi] - dt * tmp / mass[zi];
}
```

Figure 3.4: Integer-based iteration and field accesses following lowering.

ators thus leads to a domain-specific optimization more powerful, though less general, than common subexpression elimination. *Lowering* replaces the C++ objects used to access the mesh and iterate over its entities with more efficient integer-based access and iteration, as shown in Figure 3.4. This example is complicated by the loop's use of the zone not only to access a field, but also to determine the zone's volume. This latter direct use of the zone requires indexing the mesh with the induction variable to obtain a pointer and thus surrenders one of the two loads removed beyond the simplifications from common subexpression elimination. Nevertheless, roughly half of `testhydro1`'s loops access the mesh element solely for field indexing purposes. Further, lowering does not require that the compiler be able to infer side-effect properties of the lowered constructs. Hence, it is a domain-specific optimization that both overcomes potential limitations of conservative compiler analysis and provides a small additional performance benefit over traditional optimizations, such as inlining and common subexpression elimination.

Lowering required two simple modifications to `KOLAH`'s interface: the addition of integer-based field indexing (i.e., an overloaded `operator[]` accepting an integer argument) and of the `getZones()` method to expose the mesh's private zone array member variable. Our work assumes that the library writer, as the domain expert, is best suited to write domain-specific optimizations on that library. As such, we do not consider it a burden for that expert to recognize that the library should

export additional methods to be targeted by those optimizations. This approach is reminiscent of the dual APIs proposed by Mateev *et al.* [51]—a high-level interface is used by a programmer to concisely express an algorithm and a low-level interface is targeted by the compiler for greater efficiency. Insofar as `KOLAH` is a high-level interface translated to a low-level, integer-based interface, lowering may be viewed as an application of the authors' proposal, originally intended for sparse matrix computations, to the mesh domain.

Applied manually, this transformation sacrifices the expressive power of a mesh-independent construct for the efficiency of one that is intimately aware of mesh internals. Automating this optimization within a semantics-aware compiler, as done in Section 4, improves performance without imposing on the programmer. The performance improvements, though statistically significant, are a modest 3-5% over that achieved by mesh precomputation. This figure is somewhat misleading since features of lowering were incorporated in mesh precomputation. In particular, `gradInspector` in Figure 3.3 stores identifiers rather than pointers to mesh elements, so that `gradExectuor` uses these identifiers to perform the same integer-based field indexing achieved by lowering. At least 1.5M instances of the loop body are executed per invocation of the (nested) gradient and divergence operators targeted by mesh precomputation. Given the number of invocations of these operators, the number of field accesses is of the same order of magnitude as those loops targeted by lowering—singly-nested loops, similar to Figure 2.2, with at most 70K iterations and the single Caramana loop with 1.5M iterations. Hence, much of the benefit of lowering has already been provided by the mesh precomputation optimization.

3.4.3 Data and Computation Reordering

Data packing strategies that reorder the layout of data elements have been successful in improving locality and reducing bandwidth consumption [37]. We show in this section that KOLAH benefits from this traditional [22] use of data packing. In addition, we propose using packing to manipulate iteration spaces, making them amenable to code restructuring. We introduce a loop partitioning scheme that reorders computation and creates several partially evaluated versions of a loop to facilitate data reuse through blocking. We show that the memory layout induced by loop partitioning degrades performance and propose a compromise packing strategy that allows partitioning without blocking. In contrast to our earlier results obtained using KCC on a POWER3 [89], this strategy is inferior to traditional data packing.

Each of the following data packing strategies target sides, since these are accessed in the innermost loop of the performance-critical gradient and divergence operators. As sides are visited in memory order by the Caramana loop of Figure 3.2, changing their data layout has the side effect of changing the loop's iteration order. As a consequence of the non-commutativity of floating point operations, these optimizations may thus produce results that are not byte equivalent to those of the unoptimized code. For this reason, these optimizations may be considered unsafe in some contexts.

Data Packing

The triply-nested loop structure that implements the gradient and divergence operators leads to non-strided memory accesses over faces and sides. The implemen-

tation of `grad` in Figure 3.3 shows that the iteration space on sides traversed in the inner loop is defined by a face, whose iteration space is in turn defined by an enclosing loop over zones. Each of these loops accesses a subvolume of the mesh. Since these accesses do not match memory order, they result in a non-strided access pattern.

Consecutive packing [22] reduces the impact of this non-sequential behavior by linearizing mesh entities in memory according to their access order within the loop, to the extent allowed by repeated accesses. Thus mesh entities accessed consecutively in time are more likely to be stored consecutively in memory. This effectively increases spatial locality for small objects. Unfortunately, a cache line is not large enough to accommodate multiple mesh entities; consecutive accesses do not enjoy spatial reuse of a cache line. Although `testhydro1` does not benefit from fine-grained spatial reuse, consecutive packing produces a sequence of addresses more amenable to stream prefetching than those resulting from coarse-grained packing strategies, such as bucket tiling [58]. Data packing transforms the address stream such that it has many short sequences of strided addresses that can be identified and exploited by the hardware-based prefetchers. By doing so, it increases the number of L1 prefetches by 18-20% over lowering (data not shown) and reduces the number of L2 cache accesses by 10-11%. The result is a 7-9% performance improvement over lowering and an overall 2.9-3.5× performance improvement over the baseline, making it the best performing of the `testhydro1` variants across all platforms. As expected, data packing does not affect the number of instructions completed since it does not reorder or remove any computation. Similarly, the dynamic instruction mix of the `KCC`-derived executable is nearly unchanged. The application compiled with `gcc` does show small, but statistically significant, reductions in dynamic loads, stores, branches, and floating point operations following

data packing. While the static instruction mix should be identical between the lowering and data packing `testhydro1` variants, speculation within the processor can behave differently given the reordered data accesses induced by data packing. This, in turn, can result in a different dynamic instruction mix.

Iteration-space Partitioning

Execution within mesh algorithms is often conditionalized on geometric properties. For example, the loop in Figure 3.2 performs additional computation if the conditional on line 31 evaluates to true, indicating that the edge has a "left" neighbor. The complete loop has a symmetric test for the "right" neighbor. Such conditionals reduce basic block size, making it more challenging for the processor to effectively schedule instructions. This degrades performance since accommodating the frequent memory accesses and long-latency floating-point instructions common to unstructured mesh codes requires a balanced instruction mix [31].

Though these properties are not known until run time, after the mesh has been constructed, many of them remain invariant after initialization. This invariance allows the conditionals to be removed, just as static mesh connectivity allows the elimination of unnecessary recomputation. A variant of the Caramana loop is created for each of the four possible outcomes of the two branches: both taken, neither taken, left branch taken, or right branch taken. In all cases, the conditionals are removed and their bodies are inlined or removed as appropriate. For example, in the specialized loop corresponding to the existence of the right neighbor only, the two conditionals are removed and replaced with the inlined body corresponding to the right neighbor.

A one-time traversal of the iteration space evaluates the conditionals and as-

Table 3.11: Packing data to reorder Caramana loop degrades memory performance on POWER5 compiled with KCC. Performance metrics for 10 iterations of `testhydro1`.

Metric	Lowering	Iteration-space partitioning (packing without loop restructuring)
L1 cache accesses (B)	21.12	20.56
L2 cache accesses (B)	1.32	1.30
L2 misses (B)	0.24	0.35
L2 traffic (GBytes)	149.73	147.76
Memory bandwidth (GBytes/s)	53.30	59.23
DTLB misses (M)	23.10	24.99
Branches (M)	5869.15	5602.05
Unconditional branches (M)	2086.00	1985.42
Mispredicted branch direction (M)	213.23	124.22
Mispredicted branch target (M)	58.24	50.88
Flops (B)	6.99	6.84
Computational intensity	0.33	0.33
Instr completed (B)	43.21	43.22
Wall clock time (sec)	53.48	67.59

signs an edge to one of four partitions. These partitions of the original iteration space then form the sub-iteration spaces for the specialized loops. Packing reorders the original iteration space so that elements in each successive partition are arranged before elements in any unpacked partition. The lengths of the four partitions then divide the reordered iteration space across the four loops.

The loop corresponding to edges with neither a left nor a right neighbor admits further partial evaluation, and in fact can be removed entirely. Through constant folding and aggressive inlining after both conditionals and their bodies have been removed, a compiler should be able to determine that line 37 sets `signDotProd` to

Table 3.12: Packing data to reorder Caramana loop degrades memory performance on POWER5 compiled gcc. Performance metrics for 10 iterations of `testhydro1`.

Metric	Lowering	Iteration-space partitioning (packing without loop restructuring)
L1 cache accesses (B)	35.71	34.38
L2 cache accesses (B)	1.67	1.40
L2 misses (B)	0.22	0.35
L2 traffic (GBytes)	189.38	158.82
Memory bandwidth (GBytes/s)	37.65	46.53
DTLB misses (M)	20.99	23.04
Branches (M)	9042.65	8619.47
Unconditional branches (M)	3428.59	3294.70
Mispredicted branch direction (M)	253.57	178.31
Mispredicted branch target (M)	116.73	115.89
Flops (B)	11.81	11.61
Computational intensity	0.33	0.34
Instr completed (B)	63.92	63.92
Wall clock time (sec)	68.14	83.99

zero, so that the conditional on line 38 fails and line 41 sets `edgeForcing` to the zero vector. Since adding a zero vector to `nodeForcing` has no effect[3], the entire loop is side-effect free and may be eliminated. Eliminating this loop reduced the number of executed loop instances by approximately 30K out of 1.5M.

This transformation to statically evaluate and remove conditionals is a specific instance of *iteration-space partitioning and loop specialization*. Mellor-Crummey *et al.* [53] also recognized that data packing reorders computation when the data is stored in an array that is accessed without indirection both prior to and af-

[3]Menon *et al.* [55] discuss a compiler framework that incorporates a semantic understanding of vectors and matrices. In this case, such knowledge is not required because the code is inlined as scalars, which the compiler is able to analyze.

ter packing. By using a space-filling curve to reorder computation, they attained significant cache miss reductions. Our approach differs since it reorders computation according to some property of the induction variable to facilitate subsequent optimizations. This subsequent specialization and restructuring of loop bodies contrasts with computation reordering optimizations that only reorder iteration spaces or introduce additional loop nests. Optimizations that introduce temporal locality illustrate these differences.

Gropp *et al.* [32] reorder a loop over edges in the unstructured mesh code FUN3D to introduce locality across loop body statements operating on nodes. By sorting the edges according to the identifier of the node at either end, they move loop body instances accessing the same node temporally close to one another so that they reuse data in cache. In their study of irregular scientific applications, Mellor-Crummey *et al.* [53] extend blocking used in dense-matrix calculations to interaction lists in molecular dynamics applications. They do so by first assigning a block number to each particle based on its memory location and then accessing particles by iterating over blocks.

The loop over sides in Figure 3.2 also exhibits temporal reuse; edges are revisited since they are not unique to a given side. By sorting the sides based on their edge's identifier those sharing an edge are placed contiguously in the iteration space to provide for temporal reuse of the edge. This reuse may be used to tile the loop by lifting all statements that are dependent solely on an edge before any statements dependent on the side induction variable. Because edges are reused across consecutive loop instances, we introduce an inner loop over all sides sharing an edge. This register tiling over edges ensures an edge and computation on that edge are reused across sides sharing it.

Despite reductions of 13-16% in dynamic loads, 15% in branches, and 11-14% in instructions over lowering, iteration-space partitioning performs worse than lowering and data packing (except on `gcc`/Xeon, where it slightly outperforms lowering). Tables 3.11 and 3.12 compare the performance of lowering and an optimization that uses the same data packing strategy as iteration-space partitioning, but which does not perform any loop restructuring. Therefore, unlike under iteration-space partitioning, the number of completed instructions is unchanged from lowering. As seen under data packing, there are small differences in the number of dynamic loads, stores, branches, and floating point operations. The tables show that the iteration-space partitioning data packing strategy impairs memory performance by causing a 31-36% increase in L2 cache misses and an 8-10% increase in data TLB misses. The poor memory performance may result from the packing order interfering with hardware prefetching: there are 39-40% reductions in both the number of L1 and L2 prefetches (data now shown). Unfortunately, the significant reduction in dynamic instructions does not account for the poor data layout and, as a result, performance suffers under iteration-space partitioning.

Multiple-constraint Reordering

Applications studied in previous work [22] have a single dominant loop that provides an obvious packing order; the above two sections demonstrate that this is not the case in `testhydro1`, where packing orders inspired by different loops induce different performance. The presence of multiple packing order preferences implies that orders that balance data packing's reduction in L2 accesses with iteration-space partitioning's reduction in dynamic instructions may be able to improve overall performance. Unfortunately, the compromise packing order that successfully outperformed the two previous packing strategies on an older POWER3 platform

using KCC [89] failed to do so on the more modern platforms considered here.

This compromise order visits sides in gradient-induced order but divides them into the same four partitions described above. Though partition membership is unchanged, the ordering within each partition is conducive to prefetching. As above, the loops are specialized, with the side-effect free loop eliminated. Register tiling is not applicable since this order does not contiguously place those side sharing an edge. Enforcing this additional constraint on the ordering would allow little freedom to accommodate the gradient operator.

Multiple-constraint reordering recovers most of the performance lost by iteration-space partitioning. It significantly reduces L2 misses (data not shown) and DTLB misses, though the L2 misses are not completely reduced to their number under data packing. As a result, and in contrast to our previous results using KCC on a POWER3 platform, the performance of this optimization falls short of that of data packing. Nevertheless, the results indicate the importance of appreciating (e.g., through modeling) the effect of competing packing order strategies. These considerations are important for applications that have loops of different structure or iteration spaces, which motivate different packing orders. Not surprisingly, the optimal strategy is a function of the architectural costs of memory accesses and misses: the optimal packing strategy on the POWER3 [89] is suboptimal on the POWER5.

Chapter 4

Semantics-based Abstraction Optimization

This chapter introduces a compiler framework for optimizing abstractions that targets their semantics rather than their implementations. The approach is summarized in Figure 4.1 and elaborated upon below. The projection framework is built within the ROSE infrastructure, which is reviewed in Section 4.1. Section 4.2 describes how a domain expert defines an abstraction specification in terms of abstract data types, the interfaces of procedures acting on them, and the semantics of those procedures. In addition, the expert provides one or more implementations of the abstraction interface, as well as optimizations using the ROSE infrastructure that target the abstractions. Our framework optimizes applications that use abstractions written in terms of one of the specified implementations. From these specifications, it defines lowering and raising operators, described in Sections 4.3 and 4.4, respectively. Using the raising operator, the framework automatically projects the original implementation into an abstraction space via the approach discussed in Section 4.5. Once in abstraction space, the application is transformed by the specified, domain-specific optimizations. The benefits of optimizing within the abstraction space, as opposed to one of implementation spaces, are summarized in Section 4.6. Finally, the framework projects the optimized, abstract code into a target implementation, which may differ from the original implementation, using the lowering operators and the mechanisms of Section 4.7. Several `testhydro1` loops are more complicated than the loop of Figure 4.2 because they pass variables as actual arguments to both abstractions and non-abstractions. Handling such non-abstraction uses, while retaining the ability to project the invocations

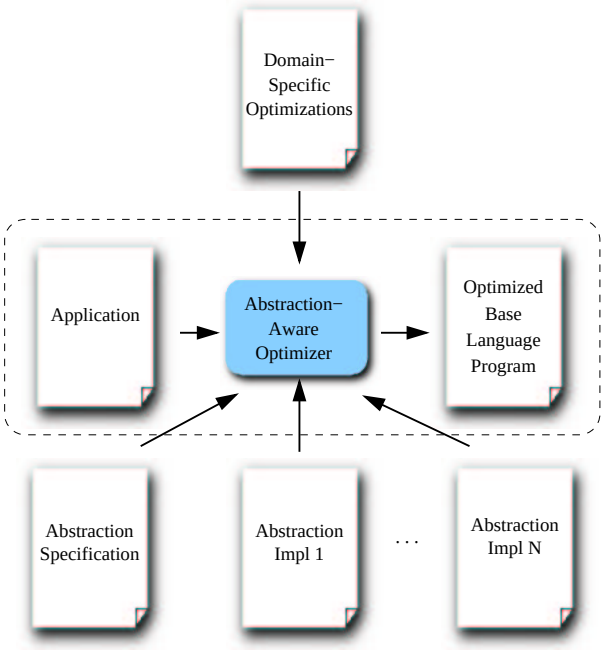

Figure 4.1: Automated abstraction recognition and optimization.

of abstraction implementations to abstraction space, requires special handling as discussed in Section 4.8. The resulting framework is applied within Section 4.9 to implement the mesh precomputation and lowering optimizations considered in the previous chapter.

The simple loop of Figure 4.2 iterates over zones to calculate the average of two pressure fields. This chapter describes the framework in terms of its action in lowering the original, KOLAH-based implementation of this loop to the target, integer-based implementation by way of the intermediary, abstract representation. The transformation of this loop from an implementation space to the abstraction space and back is summarized in Figure 4.2.

```
ZoneIterator zi;
for(zi = mesh.zoneBegin(); zi != mesh.zoneEnd(); ++zi) {
  PAvg[*zi] = 0.5 * ( P[*zi] + PNew[*zi] );
}
```

⇓

```
for(ZoneIter zi = beginZone(mesh);
    neqZoneIterator(zi, endZone(mesh));
    nextZoneIterator(zi)) {
  accessZoneScalarField(PAvg, getZoneFieldIndex(zi)) = 0.5 *
    ( accessZoneScalarField(P, getZoneFieldIndex(zi)) +
      accessZoneScalarField(PStar, getZoneFieldIndex(zi)) );
}
```

⇓

```
for(int zi = 0; zi < mesh.getNumberOfZones(); ++zi) {
  PAvg[zi] = 0.5 * ( P[zi] + PNew[zi] );
}
```

Figure 4.2: Projection of a field-averaging loop from its original, KOLAH-based implementation to abstraction space and back to an alternate, integer-based, target implementation.

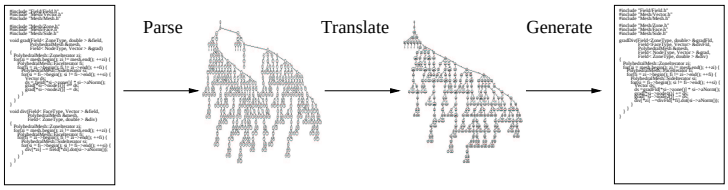

Figure 4.3: Program optimization using ROSE.

4.1 ROSE Overview

ROSE [66, 75] is a compiler framework, being developed at Lawrence Livermore National Laboratory, for the optimization of user-defined abstractions deployed within scientific codes. It aims to overcome the gap in semantic understanding between the domain experts using scientific abstractions and the compilers optimizing them. This gap is closed by having library developers and domain experts define transformations over the use of abstractions they have written. As such,

ROSE provides mechanisms to enable the creation of domain-specific optimizers. Through a set of program query and traversal mechanisms, the ROSE project lowers the barrier to specifying compiler transformations such that the technology is accessible to domain experts who are likely not compiler experts. This thesis furthers this goal by facilitating the recognition and semantic specification of abstractions in general, and mesh abstractions in particular.

Figure 4.3 shows the path that source code takes through a compiler generated using the ROSE infrastructure. Execution within a ROSE-derived compiler may be coarsely divided into three phases—source code is parsed into an abstract syntax tree (AST) in the frontend; domain expert-defined analyses and transformations act on AST in the midend; finally, the potentially modified AST is translated back to source code and compiled to object code in the backend.

4.1.1 Frontend

ROSE's frontend leverages and extends existing compiler infrastructure, including the EDG C++ frontend [24] and a modification of the SageII intermediate representation (IR) [9], dubbed SageIII. SageIII is a high-level object-oriented AST developed within the ROSE project to extend SageII [9] for greater portability and a more complete implementation of the C++ language. Source code presented to ROSE is first passed to the EDG frontend, a popular and comprehensive C++ parser, which generates a proprietary AST. This proprietary interface is not accessible to user-defined transformations, and so is next translated to the public SageIII interface.

The translation from the EDG AST to the Sage AST is provided by ROSETTA,

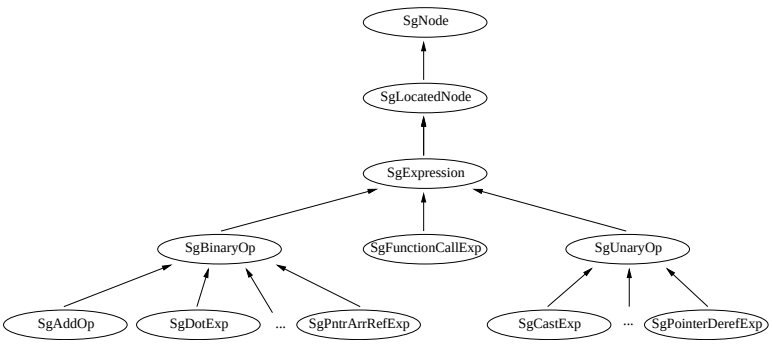

Figure 4.4: Simplified Sage class hierarchy.

a tool for defining grammars and recognizers for those grammars [68]. A grammar is specified by production rules involving terminals, non-terminals, and the SageIII source code to implement them. The terminals and non-terminals are codified within a meta-program that, when executed, produces a recognizer for that grammar. The output of this recognizer is an AST, whose nodes, including statements, expressions, types, and symbols, implement the SageIII interface. Example terminals include `SgDotExp`, for representing object-oriented method dispatch and field access on an object, and `SgAddOp`, for representing an addition expression on built-in types. A non-terminal, such as `SgBinaryOp`, which represents a binary expression, is defined on the left-hand side of a production rule with the non-terminals and terminals comprising it, including in this case `SgDotExp` and `SgAddOp`, on the right-hand side. Because each terminal only appears on the right-hand side of a single production, the production rules induce an inheritance tree on the elements, as diagrammed in Figure 4.4.

Every node in the AST is directly or indirectly derived from `SgNode`. Any node inheriting from `SgLocatedNode` retains line and column information from the original source code. `SgExpression`, derived from `SgLocatedNode`, represents an expression and is specialized by, amongst others, `SgUnaryOp`, `SgBinaryOp`, and `SgFunctionCallExp`, which define unary, binary, and function call expressions,

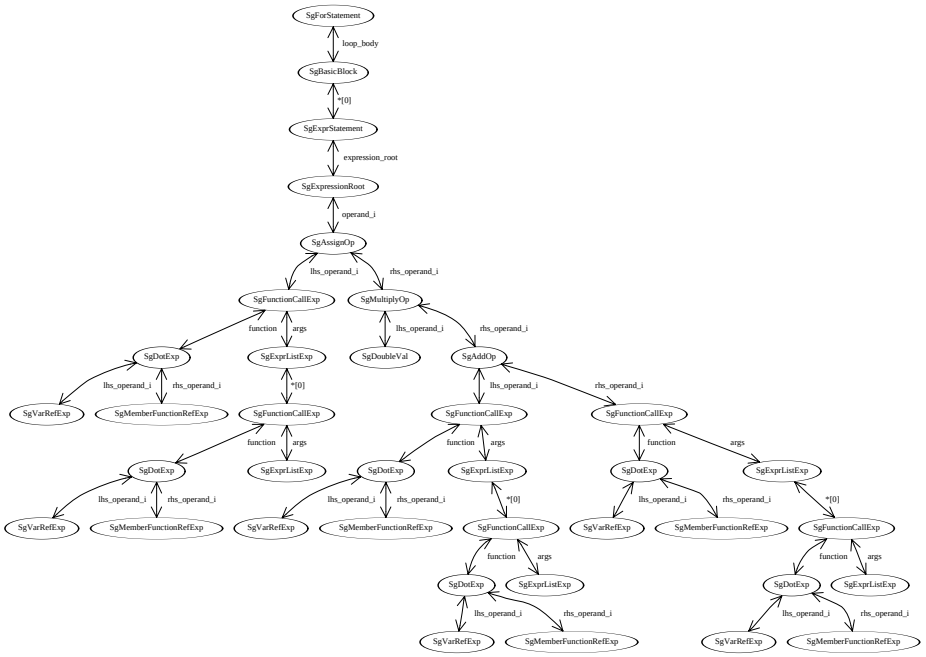

Figure 4.5: AST for body of field-averaging loop.

respectively. In turn, `SgUnaryOp` and `SgBinaryOp` are further specialized.

The AST of Figure 4.5 represents the body of the `KOLAH`-based implementation of the loop in Figure 4.2, which iterates over mesh elements to calculate the average of fields `P` and `PNew`. The top-level node in Figure 4.5, `SgForStatement`, corresponds to the `for` statement. The Sage nodes for the loop test and increment have been repressed, leaving only the `SgBasicBlock` representing the basic block of the loop body. Notice that, despite their syntactic similarity, `PAvg`, `P`, and `PNew` are user-defined field abstractions rather than arrays. Therefore, the apparent array expressions are actually invocations of the `operator[]` method on a field receiver, i.e., syntactic sugar for `PAvg.operator[](*zi)`, captured by the common motif rooted by `SgFunctionCallExp`. `PAvg` is the variable held at the `SgVarRefExp` on the left-hand side of the `SgDotExp`, while the right-hand side is the `SgMemberFunctionRefExp` for `operator[]`. The argument passed to `operator[]` itself results from a function call expression stored in the `SgExprListExp` for the

actual arguments at the callsite.

4.1.2 Midend

The ROSE infrastructure implements a number of analyses and optimizations on the AST. However, ROSE is not itself an optimizing compiler, but enables domain experts to use the general facilities it provides to craft domain-specific optimizations. These optimizations act within the midend.

ROSE provides several traditional compiler analyses, including the construction of control flow graphs and dominance trees. A wealth of additional analyses, including the side-effect and alias analyses required by our framework, are provided by OpenAnalysis [83]. OpenAnalysis decouples analysis from a language's intermediate representation (IR) through analysis-specific interfaces. These interfaces provide an analysis with program information in a representation-independent manner. Porting an analysis between languages thus consists of providing an interface to the language's IR rather than rewriting the entire algorithm. Building on a previous implementation that handled the imperative subset of C++, we worked with Michelle Strout and her research group to implement the alias IR interface for the object-oriented features of SageIII. This entailed, for example, representing references and (virtual) method invocations using OpenAnalysis's imperative analysis interfaces.

ROSE transformations are specified as semantic actions associated with a node or nodes within the AST [73]. For example, an implementation of common subexpression elimination might examine each `SgExpression` node to ascertain whether its subtree re-evaluates a previously computed expression. Transformations tra-

verse the AST to add, replace, or delete subtrees within it. The two ASTs in Figure 4.3 depict a program before and after loop fusion, in which the first AST has two large subtrees corresponding to the two loops, while the second AST has only a single large subtree for the one loop remaining after fusion.

ROSE simplifies the process of writing transformations through flexible traversal, query, and rewrite mechanisms. Pre-defined top-down, bottom-up, and top-down/bottom-up traversals require only that the transformation specify the visitor routine at a Sage node. A top-down traversal visits the AST in a top-down manner and passes inherited attributes computed at parent Sage nodes to current child node being processed. Inherited attributes are useful for passing context information down the AST [73]. Similarly, bottom-up traversals execute in the reverse direction and create synthesized attributes at the current parent node being processed from the previously processed child nodes. The final traversal mechanism merges the first two approaches. Attributes are frequently utilized to evaluate constraints or preconditions that trigger a transformation. To implement common subexpression elimination, a bottom-up traversal might concatenate nodes, passing them up as a synthesized attribute to a `SgExpression` where they would be available in some linearized form.

Query mechanisms are effectively more specialized traversals with simpler interfaces. They allow a transformation to interrogate an AST for a list of nodes having a specific type or name, for example. A generalized interface takes a solver routine that is automatically executed on each node and that returns a boolean indicating whether the node should be collected and returned.

ROSE offers string- and node-based AST transformation mechanisms. A replacement or insertion code fragment may be specified by a source string, in which

case it is presented to the frontend and translated to an AST fragment which is then grafted into the AST [75]. The translated fragment is verified to be correctly typed before insertion into the AST. The node-based rewrite mechanism is more direct, allowing a transformation to directly specify an AST subtree that will be inserted or deleted at some node within the original AST.

4.1.3 Backend

The backend unparses the AST to generate source code and optionally passes the source code to a more traditional compiler, such as `KCC` or `gcc`. Using a ROSE translator to compile an application is as easy as replacing the compiler named in the makefile with the name of the ROSE translator. By leveraging traditional compilers, a ROSE source-to-source translator is spared the additional complexity of low-level register allocation and object code generation. Further, ROSE translators can focus on high-level semantics-based optimizations, leaving well-known, general-purpose optimizations such as dead-code elimination to the traditional compiler.

4.2 Abstraction Specification

The inability of compilers to recognize high-level abstractions hinders analysis and prevents optimization because compilers must make conservative assumptions about their implementations. Annotations permit domain experts to supplement traditional analyses by communicating their understanding of application semantics [69, 33, 45, 91]. These annotations frequently are couched in terms of low-level

effects such as variable use, modification, and aliasing relationships that are readily incorporated into traditional compiler analyses. They may also express higher-level concepts, such as banded, diagonal, or symmetric matrix shape information, which suggest domain-specific optimizations.

Whether communicating general-purpose analysis information or domain-specific properties, previous annotation languages are implementation-centric in that annotations are ascribed to concrete function, method, and type implementations. This approach is not scalable as it requires that annotations be repeatedly specified for each implementation. Domain-specific concepts will be re-implemented by different applications or libraries. For example, there are a variety of public implementations of mesh and field abstractions [8, 60, 89]. Despite the shared semantics of these implementations, each would have to be separately and redundantly annotated.

We instead propose an abstraction-centric approach, which allows abstraction specifications to be reused across library implementations. A domain expert communicates a set of abstractions to the projection framework through an abstraction specification. A fragment of the specification describing mesh iterator and field abstractions is shown in Figure 4.6. It opens with declarations of abstract data types, or abstraction types, and continues with the interface describing operations on those types. The example interface declares mesh, iterator, and field types. Operations on these types include those to retrieve a zone iterator referencing the "first" zone within a mesh, to advance an iterator to refer to the next zone, and to access a scalar field using a zone index.

The specification provides an imperative, rather than an object-oriented, interface. Though abstraction types are introduced through the C++ `class` keyword,

these classes serve only to define new type names and do not define methods on those types. Therefore, the entire specification could have been written in C had the inessential `class` keyword not be used as syntactic sugar for `typedef struct`. The choice of an imperative specification may seem counterintuitive since an abstraction is properly viewed as a type or set of types and the procedures acting on them. Nevertheless, given this decision, implementations of an abstraction, be they built-in C/C++ expressions, function calls, or method invocations, will be projected to invocations of abstraction *functions* in abstraction space. Thus, optimizations targeting abstractions and acting within abstraction space need not differentiate between functions and the numerous variants of methods (e.g., static or virtual) or consider the type of the receiver (i.e., pointer or reference/object), and then account for the implications these differences have on their representation within the object-oriented SageIII AST. Instead, optimizations may effectively target a C AST rather than a more cumbersome C++ AST.

Though implementations realize an abstraction, it is actually the latter that follows from the former. For example, Musser and Stepanov [61] generalize efficient implementations in order to arrive at the more abstract STL interface. It is this requirement that an abstraction interface accommodate each of its implementation that explains the seemingly redundant inclusion of iterators for zones (i.e., `ZoneIter`) and nodes (i.e., `NodeIter`) in the specification. Certainly, within a representation in which fields are indexed by integers, iterators over zones and nodes are implemented as integers and there is no need for a distinction between the two. However, a `KOLAH`-based implementation has separate template instantiations (that is, unique types) for zone and node iterators. Hence, while the use of distinct abstraction iterators is redundant for the integer-based implementation, the use of a single iterator abstraction type would be insufficient for the

KOLAH-based implementation. Further, the set of abstraction types may be richer than required for any one particular implementation. For example, the abstraction specification includes iterator types (i.e., ZoneIter and NodeIter) as well as types to index a field (i.e., ZoneFieldIndex and NodeFieldIndex). Under a integer-based approach each of these is implemented as an integer, however KOLAH uses a ZoneIterator to iterate over PolyhedralZones, which then index fields.

Following the approach used in STL [61], abstraction semantics are captured in naming conventions and documentation. For example, the beginZone and nextZoneIterator procedure abstractions are annotated to indicate they access and return static mesh connectivity metadata and are thus candidates for mesh precomputation. Similarly, since accesses to a field follow "array-like" semantics, the corresponding operations are annotated as side-effect free, thus facilitating the lowering optimization.

The specifications of semantics within documentation is fragile in that the optimizations that exploit them do not check abstraction properties, but rather make assumptions based on the documentation. If the documentation was updated to reflect different semantics, the optimization would make unsafe assumption unless it too were updated. Fortunately, this is a shortcoming of the implementation, rather than the design, of our approach. These semantics could easily be codified within the body of the procedure abstraction specifications by adding annotation keywords, such as NO_SIDE_EFFECTS or MODIFIES(var). Doing so within STL would be require extending the C++ language to account for the new keywords and would therefore not be feasible. Fortunately, since our approach uses source-to-source translation, we can use an elegant strategy to effectively add keywords to the annotation language. The parser for the specification consists of the ROSE

frontend C/C++ parser and our own specification parser that traverses the resulting AST. If the specification were written in a new language or required modifying an existing language, a new grammar and language parser would need to be written—i.e., SageIII and its parser would have to be modified. Instead, we define the specification language as being C/C++ with a few extensions that are implemented as reserved function calls. For example, rather than being a true reserved keyword in the specification language, `NO_SIDE_EFFECTS` would be a function defined in an annotation header file. Extending the specification language therefore requires the much simpler task of modifying the specification parser to understand a new function invocation and its semantics. In this case, it would understand that the abstraction specification invoking `NO_SIDE_EFFECTS` does not modify any global state. Using C/C++ as the specification language also relieves the domain expert from having to learn another language in order to specify abstraction semantics. While this change would be straightforward, the current implementation relies on documented semantics since nothing would be gained within our prototype through the use of explicit semantic specification. In fact, this strategy of effectively adding keywords to the specification language is used for an alternate purpose in Section 4.8.

A more serious concern is our reliance on the correctness of the semantics, however they are provided. However, in so doing, we follow previous approaches [69, 33, 45, 91], that require domain experts to manually provide annotations, which are accepted as valid semantic assertions. Some earlier work views annotations as a means of supplying the compiler with transformations that it could not otherwise discover or prove to be legal [46]. In principle, some degree of automatic annotation generation and verification should be possible. Such automation could exploit the mechanisms discussed here but is beyond the scope of

our current work. We believe that the *ex nihilo* generation of annotations poses greater difficulties than verification: whereas the latter is focused on proving a specific statement about a program, the former is a blind exploration of program properties. Automated annotation generation is likely to get mired in implementation internals, such as book keeping and the temporary relaxation of invariants within procedures, which obscure those semantics that are accessible to the programmer [91]. Overcoming these low-level implementation obstacles will require sophisticated, and hence inefficient, analyses that will generate a considerable number of irrelevant annotations. In some cases no traditional analysis suffices. For example, Kulkarni *et al.* [49] noted that the internal representation of a set could be dependent on the order in which items are added to it. Therefore worklist-based iterations that add elements to the set will appear to be non-commutative—the loop can not be reordered. Nevertheless, they found that the operations do commute at the semantic level, a fact that no automated annotation generation would be able to infer.

4.3 Lowering Operator Specification

The domain expert provides one or more concrete implementation *domains* that realize the entire set of procedures supported by an abstraction. Figures 4.7 and 4.8 provide `KOLAH`- and integer-based implementation domains, respectively, corresponding to the subset of mesh abstractions listed in Figure 4.6. An implementation is implicitly mapped to its abstraction by sharing the same function name. Each procedure implementation is effectively a lowering operator, or transducer, from the abstraction sharing its name to its body, which provides a realization of that abstraction. In principle, mapping mechanisms other than this named-based

```
class Mesh { };

class ScalarType { };
class VectorType { };

class ZoneIter { };
class ZoneFieldIndex { };
class ZoneScalarField { };
class ZoneVectorField { };

class NodeIter { };
class NodeFieldIndex { };
class NodeScalarField { };
class NodeVectorField { };

// Procedure abstractions on iterators.

// Retrieve the first zone of a mesh.
// Semantics: No side effects.
//            Static mesh connectivity operator.
ZoneIter beginZone(Mesh &mesh) { }

// Increment a zone iterator.
// Semantics: Static mesh connectivity operator.
ZoneIter nextZoneIterator(ZoneIter &it) { }

// Procedure abstractions on fields.

// Access a scalar field with a zone index.
// Semantics: No side effects.
ScalarType accessZoneScalarField(ZoneScalarField &field,
                                 ZoneFieldIndex &indx) { }

// Access a vector field with a node index.
// Semantics: No side effects.
VectorType accessNodeVectorField(NodeVectorField &field,
                                 NodeFieldIndex &indx) { }
```

Figure 4.6: Subset of the specification of mesh abstraction types and procedures pertinent to lowering.

scheme could be employed, including one using an explicit `implements` keyword or pragma. The simplicity of the current approach should facilitate its use. Mateev *et al.* [51] use the more obvious and elegant approach of employing (pure virtual) base classes to represent abstractions and subclasses to represent implementations. Unfortunately, that approach is not applicable to legacy code, such as the scientific codes targeted by this work. That is, having `KOLAH`'s `ZoneIterator` provide a concrete implementation of the abstract `ZoneIter` would require modifying the definition and inheritance relations of the `ZoneIterator` class. Further, the approach is not amenable to basic types: though an integer provides the functionality of a `ZoneIter` for the integer-based implementation, it is not possible to make it a subclass of `ZoneIter`.

A traversal over the AST defined by an implementation domain ensures that each procedure abstraction has a corresponding implementation and establishes a map from the procedure abstraction and domain name to the corresponding procedure implementation. It also establishes a map between abstraction and implementation types by comparing the types of procedure abstractions and implementations. For example, by comparing the type signatures of the `beginZone` abstraction with its integer-based implementation, the traversal establishes a map between the `Mesh` abstraction type and the `PolyhedralMesh` implementation type and between the `ZoneIter` abstraction type and the `int` implementation type. Similarly, examination of `accessZoneScalarField` results in a map between the `ZoneFieldIndex` abstraction type and the `int` implementation type. Thus, the map from abstraction to implementation types is surjective (or onto). Individual abstraction types can map to alternate implementation types in a different implementation domain. For example, under the `KOLAH`-based implementation, the `Mesh` abstraction type continues to be mapped to the `PolyhedralMesh` im-

```
// Pull in KOLAH typedefs so that we can refer to them.
#include ``KOLAHTypedefs.h''

// Implementation of iterator abstractions.

ZoneIterator beginZone(PolyhedralMesh &mesh)
{
  return mesh.zoneBegin();
}

ZoneIterator nextZoneIterator(ZoneIterator &it)
{
  return ++it;
}

// Implementation of field abstractions.

Scalar accessZoneScalarField(ZoneScalarType &field,
                             PolyhedralZone &indx)
{
  return field[indx];
}

Vector accessNodeVectorField(NodeVectorType &field,
                             PolyhedralNode &indx)
{
  return field[indx];
}
```

Figure 4.7: KOLAH-based implementation of mesh abstractions.

plementation type, while the `ZoneFieldIndex` abstraction type is mapped to the `PolyhedralZone` implementation type.

4.4 Raising Operator Inference

The abstraction implementations provide a direct means of projecting from abstraction space to implementation space: the appropriate implementation is effectively inlined for the abstraction invocation. The reverse projection from imple-

```
// We still need KOLAH type declarations.
// Not all abstraction types are implemented
// as ints or vectors.
#include ‘‘KOLAHTypedefs.h’’
#include <vector>

// Implementation of iterator abstractions.

int beginZone(PolyhedralMesh &mesh)
{
  return 0;
}

int nextZoneIterator(int &it)
{
  return ++it;
}

// Implementation of field abstractions.

Scalar accessZoneScalarField(ZoneScalarType &field,
                             int &indx)
{
  return field[indx];
}

Vector accessNodeVectorField(NodeVectorType &field,
                             int &indx)
{
  return field[indx];
}
```

Figure 4.8: Integer-based implementation of mesh abstractions.

mentation space to abstraction space may be inferred by inverting the lowering operator, in principle, to define a raising operator. For example, if an invocation of the function `getFirstZone` sufficed to implement the `beginZone` abstraction, then the lowering operator of Figure 4.9 could be inverted to define the raising operator `getFirstZoneRaisingOperator`, whose name indicates the implementation `getFirstZone` it raises. Raising would then proceed by comparing the name and type signature of a function invoked from the original implementation against the

```
// Lowering operator from abstraction beginZone to
// implementation getFirstZone.
int beginZone(PolyhedralMesh &mesh)
{
  return getFirstZone(mesh);
}

// Raising operator from implementation getFirstZone
// to abstraction beginZone.
int getFirstZoneRaisingOperator(Mesh &mesh)
{
  return beginZone(mesh);
}
```

Figure 4.9: Function-based lowering and inverted raising operators.

names and type signatures of defined raising operators. Notice that the raising operator is defined in terms of abstraction types, e.g., its formal mesh parameter has the abstraction type `Mesh` rather than the implementation type `PolyhedralMesh`.

While we are free to mandate that abstraction procedures be defined as functions, their implementations may be defined as methods (as in the `KOLAH`-based implementation domain) or as built-in language expressions (as in the integer-based implementation domain). Further, the above approach assumes that an implementation is restricted to a single function invocation. This technique is generalized so that the name of the raising operator is irrelevant. Instead, the AST of the body of the lowering operator is mapped to its associated abstraction: the AST is mapped to a raising operator that invokes the abstraction, but whose name is effectively meaningless, as shown in Figure 4.10. The next section describes how this AST is bound to a potential invocation of an abstraction implementation in order to project it to an invocation of that abstraction.

```
// Lowering operator from abstraction beginZone to
// implementation getFirstZone.
int beginZone(PolyhedralMesh &mesh)
{
  return getFirstZone(mesh);
}

// Raising operator from implementation getFirstZone
// to abstraction beginZone.
int beginZoneRaisingOperator(Mesh &mesh)
{
  return beginZone(mesh);
}
```

Figure 4.10: Lowering and inverted raising operators.

4.5 Projection to Abstraction Space

The AST of the lowering operator's body serves as a means of recognizing invocations of abstraction implementations. A traversal over the application AST examines each function or method invocation. It attempts to bind the AST of each invocation expression with the AST of the lowering operator's body[1]. A successful comparison requires that the two subtrees match structurally, with the formal parameters of the lowering operator serving as wildcards that match with any subexpression from the invocation expression, so long as their types unify. Successful bindings are recorded, but do not immediately result in a transformation of the AST.

Projection must also convert the implementation types of variables to abstraction types. Not all variables are candidates for projection. Formal parameter types are never converted to abstraction types. If the target and original implementation types corresponding to the formal parameter's abstraction type differed, then

[1]Binding actually occurs between the AST of an invocation expression and the AST of the expression returned by the lowering operator.

projection to the target implementation would change the formal parameter type. The legality of this transformation would require examination of all callsites invoking the formal parameter's procedure to ensure that the type of the corresponding actual argument could be converted at the caller. For simplicity, we avoid such interprocedural analysis by disallowing the conversion of formal parameter types.

Similarly, a variable's type is only converted to an abstraction type if all of the variables uses are within an abstraction context—i.e., uses and definitions of the variable are restricted to its being passed as an actual argument to an abstraction invocation, its being initialized to an abstraction, or its being declared without an initializer. In any other case, the variable would be required to have its original type, which might change as a result of projection. If the variable's original and target implementation types differ, but a conversion between the two types exists, then the variable type may still be promoted to an abstraction type. Whenever the variable's original type is required, it may be converted from the target implementation. This accommodation of non-abstraction variable uses is discussed further in Section 4.8.

An invocation of an abstraction implementation is only projected to an invocation of the abstraction if it may be further projected back down to the target implementation[2]. We call such projections *valid*. An invocation projection is valid if each of its actuals has a type that may be projected to abstraction space or has a type that is invariant between the original and target implementations. If an invocation is determined to be invalid, then it will not be projected, and all of its actual arguments are effectively used in a non-abstraction context. Any of these actual arguments that are variables can no longer be considered for projection. This, in turn, may cause other invocation projections to be invalid, which causes

[2]When the original and target implementations are the same, this condition always holds.

the algorithm to iterate. Since there are a finite number of candidate invocation projections and each may be changed only from valid to invalid, the algorithm terminates.

4.6 Optimization in Abstraction Space

Once an application implementation has been projected to abstraction space, domain-specific optimizations may be applied. Whereas optimizations acting in implementation space would target specific implementations, those acting in abstraction space target the semantics of those implementations. Therefore, abstraction space provides a more expressive view in which opaque function and method invocations have been replaced by invocations to abstract procedures with well-defined semantics. For example, the abstract representation of the loop in Figure 4.2 consists of abstraction invocations, which have been annotated as side-effect free. Such annotations would allow common subexpression elimination of the repeated `*zi` expression, which was not possible in the original implementation space because of a conservative assumption that `operator[]` induces side effects. Instead, the related, but more powerful, lowering optimization translates the original loop to a integer-based target implementation.

Optimizations acting on abstractions should be portable across implementations within a domain, in the same manner in which low-level compiler optimizations such as common subexpression elimination are applicable across base languages (such as C or Fortran) because they target a common intermediate representation. For example, if an implementation specification for one of several other mesh libraries [8, 60] were provided, then the optimizations applied to `KOLAH`

in Section 4.9 should also be applicable to algorithms written using these libraries.

Previous projects [34, 45] have used sophisticated code pattern replacement mechanisms to specify program transformations. Pattern matching and replacement are well suited for small, local source code modifications, such as callsite specialization, but are less accommodating of higher-level transformations, such as loop fusion, which require contextual information and consideration of complex preconditions [45]. To support a more general flavor of optimization, we allow library writers to specify arbitrary transformations on the program using either string replacement or direct insertion, deletion, and modification of AST subtrees. The ROSE project has taken great pains to facilitate writing transformations so that doing so does not require extensive compiler expertise. In this work we further ameliorate the task of writing transformations by representing abstractions as function invocations within abstraction space, regardless of whether their implementations use functions, methods, or built-in expressions. Thus, domain-specific optimizations do not need to anticipate which of these approaches is used to represent abstractions, which would bind them to a specific implementation of those abstractions, or to handle all possible implementation styles, which would be a burden to the expert providing the optimization.

Section 4.9.1 describes how mesh precomputation acts within the abstraction space. However, projection itself may be of value even if no domain-specific optimization is explicitly applied. For example, as discussed further in Section 4.9.2, lowering may be implemented simply be specifying a target implementation that differs from the original implementation.

4.7 Projection to Implementation Space

The validity of lowering procedure and type abstractions from abstraction space to the target implementation space is guaranteed before projection to abstraction space. Therefore lowering is a simple matter of traversing the AST of the optimized, abstract application to determine which functions invoke abstractions and which declarations introduce abstraction types. In the former case, the appropriate lowering operator is selected, i.e., that with the same name as the invoked abstraction and with a signature that unifies with the invoked abstraction after types have been mapped from abstraction space to the target implementation space. The body of this lowering operator is inlined for the invocation of the abstraction. A type is projected from abstraction to target implementation space by changing the abstraction type mentioned in a variable declaration to the implementation type to which it is mapped.

4.8 Accommodating Non-abstraction Invocations

To avoid costly whole-program analysis, the projection framework does not permit the types of formal variables to be raised to abstraction space. For example, the original implementation type `PolyhedralNode` of the first formal parameter of the `moveNode` method invoked in Figure 4.11 is not projected to its corresponding abstraction type `NodeFieldIndex`. If it was, subsequent lowering to an integer-based target implementation would translate the `NodeFieldIndex` abstraction type to the `int` target type. Therefore, it would be necessary to modify the implementation of `moveNode` to accept an `int`, rather than a `PolyhedralNode`, argument.

The restriction against projecting formal types could in turn prevent the raising of any variables accessed in the corresponding actual argument. In this case, the `NodeIterator ni` would not be projected to the `NodeIter` abstraction type. This, in turn, would prevent projection of any invocation involving `ni` and would ultimately prevent projection, and consequently optimization, of the loop.

A correct, but less conservative, approach is to seek a conversion from the target implementation type to the original implementation type. If such a conversion is available, the variable used in the expression passed as the actual argument may be translated back to original implementation, as shown in Figure 4.12. Figure 4.13 lists the abstraction specification and implementation of the conversion operator that would be provided by the domain expert to effect this translation. The conversion abstraction `getNodeIterator` is labeled as a conversion operator via the `CONVERSION_OPERATOR` annotation. The `EXCLUDED_OPERATOR` annotation indicates that the projection framework should not attempt to project implementations of `getNodeIterator` to the abstraction space during raising, but should only use it for purposes of type conversion.

The operator converts its first `int` actual argument to a `NodeIterator`, but additionally requires a second `PolyhedralMesh` argument. Therefore, in order to convert the `int ni` back to a `NodeIterator`, the projection framework needs to determine this second actual argument. From the correspondence between `ni` and `mesh` in the loop header of the original implementation, it is clear that `mesh` should be passed as this second argument.

Automating this inference of actual arguments requires associating `mesh` with `ni` so that it may be accessed at the `*ni` callsite and passed to the conversion operator. This problem is solved by allowing attributes to be defined on variables.

Attributes tag variables, but exist only within the analysis of the loop and are not defined syntactically within the application. Procedure abstractions may use the intuitive annotation language of Figure 4.14 to assign a variable to an attribute, to transfer one attribute to another, and to assign an attribute's value to a variable. The attributes are then propagated by a data-flow solver defined within the OpenAnalysis framework.

The data-flow annotations necessary for associating `mesh` with `ni` are specified in the procedure abstractions for the conversion operator `getNodeIterator` of Figure 4.13 and in the procedure abstractions `nextNodeIterator` and `beginNode` of Figure 4.15. The annotation of `beginNode` assigns the actual argument `mesh` passed to the abstraction invocation as the "mesh" attribute of the return value. The data-flow analysis then insures that the attributes of the return value are propagated to the left-hand side during assignment or initialization. Therefore, after being raised to the abstraction space, data-flow analysis of the loop initialization `NodeIterator ni = mesh.nodeBegin()` associates `mesh` with the "mesh" attribute of `ni`. Such an attribute may be propagated by an annotation such as that specified for `nextNodeIterator`, which assigns the "mesh" attribute of its formal parameter `it` to the "mesh" attribute of its return value. Finally, the attribute may be queried where it is needed, at the abstraction conversion procedure `getNodeIterator`. Its annotation specifies that the `mesh` formal parameter should be assigned the "mesh" attribute of its first `it` formal parameter. This effectively provides a default value for a parameter.

```
for(NodeIterator ni = mesh.nodeBegin();
    ni != mesh.nodeEnd(); ++ni)
{
  theTransformation.moveNode(*ni, .5 * dt * uAvg[*ni]);
}
```

Figure 4.11: Node used both in an abstraction context (as an actual parameter to `uAvg`) and in a non-abstraction context (as an actual parameter to `moveNode`).

```
for(int ni = 0; ni != mesh.getNumberOfNodes(); ++ni)
{
  NodeIterator convertedVar(mesh.nodeBegin() + ni);
  theTransformation.moveNode(*convertedVar, .5 * dt * uAvg[ni]);
}
```

Figure 4.12: A variable used in an expression passed as an actual parameter is converted from the target implementation type back to original implementation type.

```
// Abstraction specification of conversion operator from
// int type to NodeIterator type.
NodeIterator getNodeIterator(int &it, PolyhedralMesh &mesh)
{
    VAR(mesh) = PROPERTY(VAR(it), "mesh");
    PROPERTY(RET(), "mesh") = PROPERTY(VAR(it), "mesh");
    CONVERSION_OPERATOR();
    EXCLUDED_OPERATOR();
}

// Implementation of conversion operator.
NodeIterator getNodeIterator(int it, PolyhedralMesh &mesh)
{
    return ( mesh.nodeBegin() + it );
}
```

Figure 4.13: Conversion operator from `int` type to `NodeIterator` type.

VAR	→	C/C++ variable name
	|	"ret"
ATTRIBUTE	→	string
PROPERTY	→	VAR "." ATTRIBUTE
ATTR_ASSIGN	→	PROPERTY "=" VAR
PROPAGATION	→	PROPERTY "=" PROPERTY
VAR_DEFN	→	VAR "=" PROPERTY
ANNOTATION	→	ATTR_ASSIGN
	|	PROPAGATION
	|	VAR_DEFN

Figure 4.14: Data-flow annotation language for propagating attributes.

```
NodeIter nextNodeIterator(NodeIter &it)
{
    PROPERTY(RET(), "mesh") = PROPERTY(VAR(it), "mesh");
}

NodeIter beginNode(Mesh &mesh)
{
    PROPERTY(RET(), "mesh") = VAR(mesh);
}
```

Figure 4.15: Specification of data-flow problem with abstraction specification.

4.9 Automated Mesh Optimizations

To illustrate the versatility of the projection framework, we use it to automate two optimizations that were manually applied in Chapter 3. Mesh precomputation consists of an optimization phase that replaces the original gradient and divergence operators with the two-phased inspector/executor versions described in Section 3.4.1. Lowering, discussed in Section 3.4.2, leverages projection without requiring any explicit transformation in the optimization phase.

Table 4.1: Automated mesh precomputation and lowering attain performance similar to manual optimizations on POWER5 compiled with KCC. Performance metrics for 10 iterations of `testhydro1`.

Metric	Iteration-space narrowing	Mesh precomputation	Automated mesh precomputation	Lowering	Automated lowering
L1 cache accesses (B)	32.92	22.41	22.76	21.12	21.45
L2 cache accesses (B)	1.82	1.26	1.21	1.32	1.31
L2 traffic (GBytes)	206.68	143.16	137.70	149.73	149.19
Memory bandwidth (GBytes/s)	45.57	52.24	49.72	53.30	51.60
DTLB misses (M)	25.79	23.96	23.39	23.10	22.87
Branches (M)	8971.01	6353.66	6416.16	5869.15	5866.99
Unconditional branches (M)	3708.27	2341.07	2409.20	2086.00	2139.79
Mispredicted branch direction (M)	261.41	217.03	212.46	213.23	212.14
Mispredicted branch target (M)	66.17	64.67	68.03	58.24	56.80
Flops (B)	6.98	6.98	6.98	6.99	6.97
Computational intensity	0.21	0.31	0.31	0.33	0.33
Instr completed (B)	66.46	45.80	46.58	43.21	43.89
Wall clock time (sec)	70.70	55.38	55.07	53.48	53.51

Table 4.2: Automated mesh precomputation and lowering attain performance similar to manual optimizations on POWER5 compiled with gcc. Performance metrics for 10 iterations of `testhydro1`.

Metric	Iteration-space narrowing	Mesh precom-putation	Automated mesh precom-putation	Lowering	Automated lowering
L1 cache accesses (B)	64.13	39.58	40.17	35.71	37.02
L2 cache accesses (B)	1.62	1.47	1.42	1.67	1.56
L2 traffic (GBytes)	184.33	167.35	161.09	189.38	177.36
Memory bandwidth (GBytes/s)	31.46	36.96	35.92	37.65	38.58
DTLB misses (M)	23.47	21.47	21.30	20.99	20.89
Branches (M)	17916.76	11002.17	11115.50	9042.65	9188.57
Unconditional branches (M)	7297.77	4233.33	4303.46	3428.59	3582.71
Mispredicted branch direction (M)	297.08	229.00	262.29	253.57	252.33
Mispredicted branch target (M)	168.47	149.42	155.00	116.73	121.24
Flops (B)	13.93	12.16	12.18	11.81	11.96
Computational intensity	0.22	0.31	0.30	0.33	0.32
Instr completed (B)	118.07	73.13	74.11	63.92	65.50
Wall clock time (sec)	97.40	71.75	72.27	68.14	68.92

Table 4.3: Automated mesh precomputation and lowering attain performance similar to manual optimizations on Xeon compiled with gcc. Performance metrics for 10 iterations of `testhydro1`.

Metric	Iteration-space narrowing	Mesh precomputation	Automated mesh precomputation	Lowering	Automated lowering
Wall clock time (sec)	55.59	45.07	44.48	43.38	41.55

4.9.1 Automated Mesh Precomputation

Mesh precomputation first determines candidate loop nests for optimization. Precomputation is only of benefit for nested loops, as the overhead of mesh traversal in non-nested loops is better handled by lowering. To target the gradient and divergence operators, we require that target nests have three or more nested loops. Each of these must iterate over the mesh—i.e., it must have a single *mesh iterator* that is advanced by one of the iterator-modifying abstraction procedures (e.g., `nextNodeIterator`, `addZoneMeshIndexOffset`).

Loops iterating over the mesh will not be emitted in the executor, therefore the values of any expressions that access mesh iterators within the loop, other than those controlling iteration and annotated as static mesh connectivity operators (e.g., `beginZone`, `endZone`, `nextZoneIterator`), should be precomputed within the inspector so that they are available within the executor. For the divergence operator of Figure 4.16, such expressions include the three dereferences of a zone iterator `*zi` used to index the `div` field, the dereference of a side iterator in `si->getFPPAreaNormal`, and the node iterator dereferences `*si->node1()` and `*si->node2()` that index `field`. Precomputation within the inspector is performed by evaluating the expression (e.g., `*zi`) and storing it in an STL vector (e.g., `zone1Its.push_back(*zi)`). As discussed in Section 3.4.2, rather than

```
void divInspector(Field& field, Mesh& mesh) {
   int tripCnt = 0;
   for (int zoneID = 0; zoneID < mesh.getNumberOfZones(); ++zoneID) {
      ZoneIterator zi = mesh.zoneBegin() + zoneID;
      for (FaceIterator fi = zi->faceBegin();
           fi != zi->faceEnd(); ++fi) {
         for (SideIterator si = fi->sideBegin();
              si != fi->sideEnd(); ++si) {
            sideIts.push_back(si->getID());
            Vector area = si->getFPPAreaNormal();

            // The values stored in zone1Its and zone2Its
            // are the same as those stored in zone3Its.
            // However, zone3Its is updated once per zone,
            // whereas the other two arrays are updated
            // much more often (once per side) with the
            // same value.
            zone1Its.push_back(zi->getID());
            node1Its.push_back(si->node1()->getID());
            div[*zi] -= 0.5*field[*si->node1()].dot(area);

            zone2Its.push_back(zi->getID());
            node2Its.push_back(si->node2()->getID());
            div[*zi] -= 0.5*field[*si->node2()].dot(area);
            ++tripCnt;
         }
      }
      tripCounts.push_back(tripCnt);
      zone3Its.push_back(zi->getID());
      div[*zi] /= zi->getVolume();
   }
}
```

Figure 4.16: A naive implementation of the divergence inspector stores identical values of `zi->getID()` in three separate arrays, including within an inner loop within which the value does not change.

storing mesh elements, the inspector stores integer mesh element identifiers (e.g., `zone1Its.push_back(zi->getID())`). This effectively implements lowering since these integer identifiers, rather than the associated mesh elements, will be used to index fields in the executor.

Figure 4.16 shows a naive implementation of the divergence inspector. It pre-

```
void divInspector(Field& field, Mesh& mesh) {
   int tripCnt = 0;
   for (int zoneID = 0; zoneID < mesh.getNumberOfZones(); ++zoneID) {
      ZoneIterator zi = mesh.zoneBegin() + zoneID;
      for (FaceIterator fi = zi->faceBegin();
           fi != zi->faceEnd(); ++fi) {
         for (SideIterator si = fi->sideBegin();
              si != fi->sideEnd(); ++si) {
            sideIts.push_back(si->getID());
            Vector area = si->getFPPAreaNormal();

            // Values of zi->getID() are not stored here,
            // but rather in the outer loop.
            node1Its.push_back(si->node1()->getID());
            div[*zi] -= 0.5*field[*si->node1()].dot(area);
            node2Its.push_back(si->node2()->getID());
            div[*zi] -= 0.5*field[*si->node2()].dot(area);
            ++tripCnt;
         }
      }
      tripCounts.push_back(tripCnt);
      zoneIts.push_back(zi->getID());
      div[*zi] /= zi->getVolume();
   }
}
```

Figure 4.17: An efficient implementation of the divergence inspector stores identical values of `zi->getID()` only once.

computes and separately stores the three `*zi` expressions, though they dereference the same zone iterator `zi` and hence evaluate to the same result. Further, the inner loop over sides is executed for 1.5M iterations, each of which stores `*zi` twice. The outer loop uniquely visits the 70K zones and is the properly place to store the expression. The naive implementation hence uses memory needlessly. Figure 4.17 shows an efficient implementation of the divergence inspector that only stores the zone iterator dereference once in the outermost loop.

The naive implementation is inefficient because it stores unique expression instances, whereas the optimized version gains efficiency by storing unique *values*. That is, though the three invocations of `*zi` are unique expressions (i.e., occur

within different statements), they do not yield unique values. Our approach is therefore to precompute expressions with unique values, which we describe with *value names*. Notice that it is insufficient and potentially incorrect to consider syntactically-identical expressions as computing the same value: if `zi` were modified in between invocations of `*zi`, then the syntactically-identical expressions would produce different values.

Value names are based on static single assignment (SSA) names [16], though the analysis does not instantiate and does not require $\Phi-$functions. Briefly, SSA analysis updates a variable's name whenever it is modified. This (conservatively) ensures that whenever a variable takes on a new value it is assigned a unique SSA name, though variable expressions with the same value may be assigned different names. The value naming analysis effectively associates variable SSA names with locations in the control flow graph. Though it does not instantiate $\Phi-$functions, it does update a variable's SSA name wherever a $\Phi-$function would be required, e.g., following the join of an if statement's true and false branches and preceeding a loop body. Value names are assigned in a straightforward manner from SSA names. A formal parameter or a variable without an initializer takes its SSA name as its value name. A variable declared with an initializer is given the value name of its initializer, while an assigned variable is given the value name of the right-hand side assigned to it. Finally, the value name of a function invocation is the name of that function prepended to the value names of its actual arguments.

The inspector is generated by a top-down traversal of the loop nest, in which a loop's non-loop statements are visited before any nested loops. Whenever the traversal encounters a candidate expression whose value name has not been previously precomputed, it emits a statement to compute and store the expression's

value. Because of the order of the traversal, a candidate expression computed and used within several nested loops is stored within the outermost loop. This ensures, for example, that `zi->getID()` is stored in the outermost loop over zones rather than the innermost loop over sides.

The inspector is also responsible for calculating the trip counts for each nested loop to be emitted as a function of its enclosing loop. A loop will only be emitted in the executor if it performs some computation beyond simple iteration. For example, one loop will be emitted for the outer loop since it divides the divergence field by a zone volume. The iteration space for this loop can be determined from the size of the `zoneIts` vector holding the precomputed values, as shown by the divergence executor in Figure 4.18. The second emitted loop corresponds to the innermost loop in the original operator that accumulates results in the divergence field. The number of iterations of this inner loop per iteration of the outer loop can not be determined from any of the vectors storing precomputed values. Instead, the inspector stores the (cumulative) trip count of an emitted inner loop at iteration i of the enclosing outer loop in `tripCounts[i]`. Therefore, the loop bounds of the inner loop during the i^{th} iteration of the outer loop are [`tripCounts[i-1]` , `tripCounts[i]`).

The executor of Figure 4.18 was generated by replacing each access to a value that was precomputed with an access to the vector holding that precomputed value. For example, the mesh element-based field access of the original loop `field[*si->node1()]` was replaced by the (lowered) integer-based access `field[node1Indx]`, where `node1Indx` is an element of the `node1Its` vector holding values precomputed for this expression.

Like the simple inspector implementation, this executor is naive and inefficient:

```
void divExecutor(Field& field, Mesh& mesh) {
   int tripCntrIndx = 0;
   int si = 0;
   for (int zi = 0; zi < zoneIts.size(); ++zi) {

     for (; si < tripCounts[tripCntrIndx]; ++si) {

       int sideIndx = sideIts[si];
       Side *side = &(mesh.getSides()[sideIndx]);
       Vector area = side->getFPPAreaNormal();

       // Accesses to zoneIts within this inner loop are
       // redundant with the one in the outer loop to
       // initialize zoneIndx3 and occur much more
       // frequently (once per side, as opposed to once
       // per zone).
       int zoneIndx1 = zoneIts[zi];
       int node1Indx = node1Its[si];
       div[zoneIndx1] -= 0.5*field[node1Indx].dot(area);

       int zoneIndx2 = zoneIts[zi];
       int node2Indx = node2Its[si];
       div[zoneIndx2] -= 0.5*field[node2Indx].dot(area);
     }
     int zoneIndx3 = zoneIts[zi];
     div[zoneIndx3] /= (mesh.getZones()[zoneIndx3]).getVolume();
   }
}
```

Figure 4.18: A naive implementation of the divergence executor redundantly accesses an array element of zoneIts, including within an inner loop.

though the inspector has properly precomputed the zone identifier once for its three uses, the executor accesses the same vector element three times, once for each use. This introduces two additional loads *per iteration* of the innermost loop. Figure 4.19 shows the efficient loop nest actually emitted by the optimization: it performs common subexpression elimination on the value names corresponding to the accesses to the vector holding precomputed values. Analysis determines the node in the control flow graph that dominates those nodes accessing the same value name. Since the values of these expressions are the same at each of these

```
void divExecutor(Field& field, Mesh& mesh) {
   int tripCntrIndx = 0;
   int si = 0;
   for (int zi = 0; zi < zoneIts.size(); ++zi) {

     // Access zoneIts once and reuse below.
     int zoneIndx = zoneIts[zi];
     for (; si < tripCounts[tripCntrIndx]; ++si) {

       int sideIndx = sideIts[si];
       Side *side = &(mesh.getSides()[sideIndx]);
       Vector area = side->getFPPAreaNormal();

       // No accesses to zoneIts within the inner loop.
       int node1Indx = node1Its[si];
       div[zoneIndx] -= 0.5*field[node1Indx].dot(area);

       int node2Indx = node2Its[si];
       div[zoneIndx] -= 0.5*field[node2Indx].dot(area);
     }
     div[zoneIndx] /= (mesh.getZones()[zoneIndx]).getVolume();
   }
}
```

Figure 4.19: An efficient implementation of the divergence executor accesses each array element of zoneIts once by holding its value in a temporary and reusing within the inner loop.

nodes, their value must also be the same at the dominating node. Therefore, the expression is assigned to a temporary at this node, which is then used to replace the original expressions. In this case, the access `zoneIts[zi]` occurs immediately before the inner loop and the temporary `zoneIndx` to which it is assigned is used within and following that inner loop.

Figures 4.1, 4.2, and 4.3 compare the automated application of mesh precomputation to `testhydro1` following its optimization with iteration-space narrowing against its manual application to the same baseline, for the `KCC`/POWER5, `gcc`/POWER5, and `gcc`/Xeon platforms, respectively. In all three instances, the performance of the automatically optimized application is within 1% of that of the

manually optimized application.

4.9.2 Automated Lowering

Automating the lowering optimization of Section 3.4.2 within the projection framework simply requires specifying a integer-based target implementation. Doing so translates the `KOLAH`-based mesh iteration and element field indexing to a more efficient implementation without the need for the domain expert to specify any explicit AST transformation. Figures 4.1, 4.2, and 4.3 show that automating lowering within the projection framework retains the benefits of the manual lowering optimization.

Chapter 5

Related Work

This thesis has examined issues relevant to the unstructured mesh domain, specifically, and to the optimization of abstractions, generally. Our characterization of `testhydro1`'s performance and study of optimizations that improve it complement existing performance studies of unstructured mesh applications. We have described the high-level data types common to these benchmarks and applications, including fields and iterators, as abstractions that are defined by their semantics, which transcend any particular implementation. Viewed within the abstraction space, an application may be considered a generic program to be specialized by any of a number of associated implementations. Generic programs are defined in terms of concepts, related to our notion of abstractions, which are modeled by specific implementations. As in our own work, concrete implementations are generalized to define a concept and are then mapped to that concept.

The mechanics of projecting between implementation and abstraction spaces are related to those techniques employed in code transformation systems. When projection is performed without optimization, as in the lowering optimization, the framework implements a sophisticated rewrite mechanism.

Our approach shares much with other work that targets specific domains by abstracting its constructs into a higher-level intermediate representation for subsequent optimization. However, like Broadway, Telescoping Languages, and previous work leveraging ROSE, ours is a general approach that is neither tailored nor limited to one particular domain.

5.1 Performance Studies of Scientific Codes

Several reports indicate the significant performance impact of indirect memory accesses on unstructured grid applications. Anderson *et al.* [3] concentrate on minimizing memory references in the Fortran 77 unstructured mesh code FUN3D for solving compressible and incompressible Euler and Navier-Stokes equations. In their performance evaluation of scientific codes, Vetter and Yoo [85] study the unstructured mesh code UMT. In contrast to our findings, they report a high computational intensity. This likely results from the mixed C and Fortran implementation of UMT, which is more stream-lined than `KOLAH`'s more general, object-oriented implementation: the function dominating UMT's runtime was translated from Fortran to C and accesses data stored in raw arrays, rather than field abstractions. The authors do find that UMT suffers poor cache performance and significant stalls due to loads. The regularity metric, defined by Mohan *et al.* [59] as the number of memory accesses that occur within a strided stream divided by the total number of accesses, lends insight to this poor cache performance: UMT has a relatively low regularity of 0.44, a result consistent with its heavy use of indirection. Jin and Mellor-Crummey [41] optimize stencil computation in SMG98 by targeting *hypre*, a library that provides abstractions of Cartesian grids, grid hierarchies, and iterators for use in creating multigrid applications.

Bagge and Haveraaen [5] applied CodeBoost to a mesh application written using the Sophus library. They described a mesh interface similar to that used in `KOLAH`, which provides `float operator[](const Mesh &, const MeshPoint &)` for sampling a mesh at a given `MeshPoint`, `MeshPoint setlex(const MeshShape &, const int &)` that returns the `MeshPoint`, `int getlex(const MeshPoint &)` that returns a numeric index associated with a mesh element, and `int`

`getsize(const MeshShape &)` that returns the number of mesh elements of a given shape or type. As we found in `KOLAH`, the authors' SeisMod solver makes frequent use of the idiom that traverses over the mesh to perform arithmetic operations on each of the mesh elements, as shown in 2.2. They considered a related lowering optimization that replaces the use of `getlex` and `MeshPoint`-based mesh accesses with more efficient integer-based mesh accesses to achieve a $4.8 - 5.7\times$ speedup.

5.2 Generic Programming

Abstractions are fundamental to generic programming, as they are to our own work. Generic algorithms are written in terms of the properties (syntactic or semantic) of types, rather than in terms of concrete types. Therefore, a generic algorithm is applicable to any set of types that has the specified syntactic or semantic behavior. Their use can lead to significant productivity gains. For example, linear algebra routines need to account for various type precisions and matrix shapes, sparsities, and row/column orientations. Accommodating the entire space of possibilities manually requires code duplication and leads to poor software maintainability. Instead, it is possible to hide these implementation concerns and to write linear algebra routines that access matrices and vectors only through iterators, as done in the Matrix Template Library (MTL) [79]. The resulting procedures are parameterized by a matrix type, which must be instantiated with a concrete type having particular precision, shape, sparsity, and orientation characteristics. Thus, instead of having separate routines for each matrix shape, such as BLAS `xGEMV`, `xSYMV`, and `xTRMV` for general, symmetric, and triangular matrix-vector multiplication, a single MTL routine suffices for the multitude of matrix types.

5.2.1 Abstractions as Concepts

Stepanov and Austern [4] described a concept as a set of requirements on a type that formalize it being an abstraction. A type is said to *model* a concept if it fulfills its requirements. Their work lead to the development of the C++ Standard Template Library (STL) [81], which defines generic algorithms by extracting the type- and implementation-dependence of routines and replacing them with type parameters that are constrained by concepts.

Thus, a generic algorithm is created from an efficient, concrete procedure by abstracting away inessential implementation details. For example, Gregor *et al.* [28] *lift* unnecessary requirements on concrete types to arrive at an algorithm at a higher level of abstraction. Musser and Stepanov [61] identify container access operations within an efficient, concrete implementation. These operations are generalized to determine the minimum behavior they must exhibit for proper use within the algorithm. After a generic algorithm has been defined, several variants may be separately implemented using different internal representations that offer, for example, better expected performance within certain contexts. These tradeoffs are then thoroughly described within documentation that allows programmers to judiciously choose the appropriate implementation given their requirements. The goal of such a process is the most general realization of an algorithm that may later be specialized based on context to provide an efficient implementation. Musser and Stepanov [61] consider generic programs written using the C++ template mechanism and instantiated with specific types before use.

In our approach, library writers or domain experts similarly abstract away unnecessary detail from implementations to arrive at an abstraction interface and then define the abstractions in terms of the implementations via projection oper-

ators. A major difference between work on generics and our own is that in the former case algorithms are expected to be written to this abstraction interface whereas in the latter low-level implementations are projected into an abstraction space. Thus, while work on generics has lead to the development of new libraries, such as the MTL, which improve programming productivity, the approach is not amenable to legacy scientific codes. Our approach makes no such imposition on the implementation of the algorithm. In some sense, the projection of a concrete implementation to an abstraction space automatically defines a generic algorithm.

Though Musser and Stepanov recognized the need to specify requirements on the type parameters, C++ provides no such mechanism for constraining type parameters (of templates). Therefore, the requirements of a particular type (e.g., that it provide random access iterator semantics including `operator+=(int n)` to advance `n` items in the sequence) are reflected in the STL documentation and naming convention (e.g., `RandomAccessIterator`), but are not enforced by any language mechanism: concepts are implicit in C++. The requirements of iterators [61] are similar to, but differ somewhat from, those defined above for mesh iterators. In particular, the authors require a dereference operation on iterators, `operator*()`, that returns the contents of the current container location referenced by the iterator. In this interface, the container is implicit and must thus be associated with the iterator (e.g., via a pointer member variable). We have instead defined a semantically similar `getZonePtr` operator that explicitly takes as arguments both the iterator and the container (or, more properly, the mesh from which the container of zones may be extracted). This interface specification is more general as it allows basic types, particularly `int`, to be treated as iterators. Such basic types can not implicitly refer to a container, so that both the index and the container must be passed to the abstract access routine. Instantiating such an

abstract access routine for an STL-like implementation thus requires inferring the container associated with an iterator. This motivated our use of data-flow-based attribute propagation.

The inability to codify requirements on type parameters in C++ leads to a two-phased approach to compilation and type-checking of templates. The first phase is triggered upon discovering a template definition and checks expressions involving non-dependent types (i.e., those independent of the template type parameters). The second phase occurs when templates are instantiated. At this point, the type parameters are bound to a particular type and the dependent expressions may be checked. Unfortunately, this deferred compilation produces obscure errors [28] referencing the implementation of the instantiated template, though the problem instead occurs because a type used during template instantiation does not fulfill its implicit requirements.

When the template type parameter requirements are made explicit via concepts, checking of definitions and uses may proceed separately. In this scenario, a concept (i.e., type requirement) effectively "stands in" for the dependent type at the template definition. Thus, since only types fulfilling the requirements will be passed as template parameters, checking of the definition may proceed independently of its invocation and prior to its instantiation. Compilation errors are expressed as failures of a type to meet a particular constraint and thus, unlike the obscure messages reported when definitions are fully checked only upon instantiation, provide a programmer with a clear indication of the bug.

5.2.2 Mapping Concepts to Models

Just as abstractions are projected to implementations, concepts are mapped to models to achieve good performance without sacrificing programmer productivity. This mapping is implicitly established by requiring that abstractions and their implementations share a name. The following reviews alternate approaches used within the generic programming community.

CLU [50] was one of the first languages to offer an explicit distinction between abstractions, which defined a behavior in terms of a set of operations, and the program or modules that implemented them. An abstraction such as `sorted_bag` is described via the `cluster` keyword as containing the procedures `create`, `insert`, `size`, and `increasing` as follows:

```
sorted_bag = cluster [t: type] is create, insert, size, increasing
  where t has equal, lt: proctype (t,t) returns (bool);
  rep = record [contents : tree[t], total : int];

  create = proc () ...
    ...
  end create;
  ...
end sorted_bag;
```

An abstraction is associated with an abstract type and an internal representation type. Only the former may be accessed and manipulated outside of the cluster defining the abstraction. The representation provides the implementation of the abstract type. A `sorted_bag` is implemented via a `tree`, holding the contents of the `sorted_bag`, and an integer `total` holding the number of items in the `sorted_bag`.

Like `sorted_bag`, `tree` is parameterized by a type `t`. In the case of `sorted_bag`, this type is constrained by a `where` clause to provide a total ordering via less than and equal operations. As in most approaches, CLU programmers have no means of specifying the semantics of the less than and equal operations beyond their type signatures.

An abstract type is represented by its interface specification, which contains the constraints on type parameters and the name and interface of each operation. As this interface specification contains all of the information required to type check uses of an abstraction, a module's use of an abstract type may be type-checked independently of any implementations of it.

A non-template approach to generic programming in object-oriented languages, such as C++, C#, Java, and Eiffel, constrains types through subtyping: in C++ terminology, a concept is represented as a (pure) virtual class and models of that concept are defined as subtypes derived from the virtual base class. Unfortunately, as mentioned in Section 4.3, this approach does not support retroactive modeling, wherein an existing model is mapped, without modification to that model's definition, to a concept. Retroactive modeling is necessary to support legacy codes.

Garcia *et al.* [25] discuss the generics capabilities of six languages: Standard ML, C++, Haskell, Eiffel, Java, and Generic C#. Figure 2 from Ref [25] compares the syntax of concepts as implemented in each of the six languages. The authors note that Haskell [65] effectively separates the definition of procedures (or methods), the definition of a concept (via type classes [87]), and the mapping of procedure definitions to a concept's requirements (via instances), such that retroactive modeling is supported. A type class is not itself a type (i.e., it can not be instantiated). Rather, it lists the operations required of a concept. A model of

that concept is established by an instance of the type class, which maps existing procedural definitions to their respective concept requirement. For example, the following states `a` is an instance of the type class `Num` if it provides methods `(+)` and `negate`, with the specified function types:

```
class Num a where
  (+)    :: a -> a -> a
  negate :: a -> a
```

The type `Int` may be declared an instance of type class `Num` by mapping its methods to those required by `Num`:

```
instance Num Int where
  x + y      = addInit x y
  negate x   = negateInt x
```

ML [57] defines a model through use of a signature, which constrains the type names, values, and nested structures appearing within it. Structures are named modules that package related functions, types, values, and nested structures. A structure definition provides a concrete implementation for each required type (e.g., `type vertex_t = int`) and function (e.g., `fun vertices (Data(n,g)) = n`) to support retroactive modeling. Nevertheless, while there is a map between type and function components of a signature and a structure, there is no explicit mapping between the signature itself and a structure that implements it. Instead, structural matching provides an implicit mapping. However, it is possible to statically determine whether a structure satisfies the requirements of a signature. For example, a structure may be assigned to another structure that is constrained by a signature to verify that it meets the required constraints. Generic ML algorithms are written as functors, whose parameters are constrained by signatures. Instantiating a

functor yields a structure specialized for the parameters, so that a functor behaves like a constrained template.

Eiffel [56] similarly allows constraints on formal type parameters and thus supports generics through type parameterization of classes. Formal parameters are specified in brackets following the class name, with constraints on a parameter following an arrow.

Eiffel represents concepts using deferred classes, which are similar to abstract classes in C++. Under this approach, a class, such as `SAVINGS`, models a concept described by a deferred class, such as `ACCOUNT`, by inheriting from it to *effect* (i.e., define) the implementation of deferred routines.

```
deferred class ACCOUNT
feature
  withdraw(amount: REAL) is deferred end
end

class SAVINGS inherit ACCOUNT
creation make
feature
  withdraw(amount: REAL) is
    do
      if bal > amount then bal := bal - amount end
  end
end
```

Java [27, 11] represents a concept, such as `Comparable`, with a type-parameterized interface that lists the concept's methods:

```
interface Comparable<T> {
  boolean better(T x);
}
```

As in ML and Eiffel, type parameters may be constrained: a type parameter in Java `extends` an interface to inherit its methods. For example, the formal parameters of `pick` extend the `Comparable<T>` interface, and so provide the `better` method:

```
class pick {
  static <T extends Comparable<T> >
  T pick(T a, T b) {
    if (a.better(b)) return a; else return b;
  }
}
```

Type parameters that are so constrained are said to be *bounded*. A type models a concept (i.e., interface) via the `implements` keyword:

```
interface Iterator<A> {
  public A next();
  public boolean hasNext();
}

class LinkedListIterator<A> implements Iterator<A> {
  public A next() { ... }
  public boolean hasNext() { ... }
}
```

The generics extensions [43] to C# closely follow the Java syntax: a concept, such as `ISet<T>`, is represented by an interface, which is modeled by a class, such

as `ArraySet<T>`, that inherits from it. An interface defines the methods supported by a concept and, like classes and methods, may be parameterized by type:

```
interface ISet<T> {
  bool Contains(T x);
  void Add(T x);
  void Remove(T x);
}

class ArraySet<T> : ISet<T> {
  public bool Contains(T x) { ... }
  public void Add(T x) { ... }
  public void Remove(T x) { ... }
}
```

A type may be constrained by a `where` clause, which indicates that the type models an interface.

Siek and Lumsdaine [80] developed the F^G language, which extends $SystemF$ with concepts, models, and `where` clauses for constraining parameter types with respect to model requirements. For example, the `Semigroup` concept names the operations that a `Semigroup` is required to support:

```
concept Semigroup<t> {
  binary_op : fn(t,t) -> t;
}
```

A type `t`, such as an `int`, models `Semigroup` by satisfying the interface requirements, as specified via a mapping from the operators named in the concept to concrete implementations:

```
model Semigroup<int> {
  binary_op = iadd;
}
```

where `iadd` is a pre-defined operation. Once defined, a concept such as `Semigroup` may be used to constrain a type parameter `t` of a generic algorithm. For example, the clause `t where Semigroup<t>` ensures that `t` supports the `binary_op` operation.

The authors' experience with F^G lead them to propose similar syntactic extensions for C++, called ConceptC++ [40, 28]. As in F^G, a concept provides the signatures of any required operations that use the concept's type parameters. For example, an `EqualityComparable` concept requires that equality and inequality operators be defined over a type:

```
template<typename T>
concept EqualityComparable {
  bool operator==(const T& x, const T& y);
  bool operator!=(const T& x, const T& y);
};
```

ConceptC++ supports *refinement* of a concept, through which it is specialized. For example, an `InputIterator` extends the requirements of `EqualityComparable` by demanding it support dereference (i.e., `operator*`) and advancement (i.e., `operator++`) operations.

```
template<typename Iter>
concept InputIterator : EqualityComparable<Iter> {
  typename value_type;
  where CopyConstructible<value_type>;
```

```
  value_type operator*(const Iter&);
  Iter& operator++();
};
```

In addition, `InputIterator` requires that the associated type `value_type` be copy constructible, as indicated by the `where` clause.

A model links an abstract concept and a concrete implementation by providing bindings for the associated types and operations required by the concept. For example, a pointer can implement the `InputIterator` model:

```
model InputIterator<char*> {
  typedef char value_type;
  char operator*(char* const& p) { return *p; }
};
```

The model satisfies all of the requirements of the concept, in the case of `operator*` through explicit binding and in the cases of `operator++`, `operator==`, and `operator!=` through implicit binding. The compiler guarantees that the model meets all requirements of the concept. Most recently [28], the authors have replaced the `model` keyword with `concept_map`, though the two provide the same functionality.

Dos Reis and Stroustrup [23] described an alternate means of specifying concepts. They argue that explicitly enumerating the procedure signatures for a concept is tedious and non-scalable. As an example of the potential growth in the number of syntactically-distinct signatures required to specify a single semantic operation, they consider an addition operator for a type `X`. Such an operator may take one argument or two, each of which may be defined as an object, a reference,

or a `const` reference. The operator and its return value may or may not be declared `const` as well. Therefore, rather than specify a set of abstract signatures, they propose specifying the required operations implicitly by using them in expressions on the abstract type. For example, the `InputIterator` would be expressed as:

```
concept InputIterator<typename Iter, typename T> {
  Var<Iter> p;       // a variable of type Iter.
  Var<const T> v;    // a variable of type const T.

  Iter q = p;        // an Iter is copy constructible.

  bool eq = (q == p); // must support equality operator,
                      // which returns a boolean.

  bool neq = (q != p); // must support inequality operator,
                       // which returns a boolean.

  v = *p;            // must support dereference,
                     // which returns a T.

  q = ++p;           // must support pre-increment,
                     // which returns an iterator.
};
```

These use patterns are reminiscent of those provided in the STL documentation to describe type requirements. For example, `bool neq = (q != p)` indicates that an `InputIterator` must provide an inequality operator that returns a boolean. The use patterns may be less restrictive. For example, `q != p;` would express the

requirement for an inequality operator, but would not impose any restriction on its return type.

Our goal is two-fold: to establish the requirements for an abstraction (i.e., concept) and to map those requirements to the implementation procedures that model them. Inferring this map from use patterns is insufficient for our purposes because it requires that the abstraction operator have the same name as the implementation operator. For cases in which this does not hold or in which the implementation type does not define methods (e.g., because it is a basic type), Dos Reis and Stroustrup introduce the `assert` keyword, which indicates that a concrete type models a concept by explicitly establishing a map. Since one of our target optimization is the lowering of an abstraction (an iterator) to a basic type (an `int`), we will require the explicit mapping, which then makes the specification of the use patterns redundant.

In summary, approaches that use subtyping to model a concept, including Java, C#, and Eiffel, do not separate the definition of the model from the language mechanism that establishes its relation to the concept. Therefore, they do not meet the two requirements posed by our environment: they do not allow a non-class-based type (i.e., base types such as `int`) to model a concept, nor do they support retroactive modeling, which is necessary to indicate that a type implemented within legacy code models a concept. These features are supported by Haskell, F^G, and ConceptC++, wherein concepts are not types and models are mapped to them via a language mechanism independent of the model implementation.

5.3 Code Transformation Systems

Examples from `testhydro1` have demonstrated the frequent tension between the expressive power of high-level abstractions and the performance of the resulting code. Gregor *et al.* [30] discovered a similar trend across several object-oriented numerical libraries, finding that expressing mathematical formulae, such as linear equations, via user-defined types and operators often leads to computationally expensive stores to temporary variables. For example, a naive implementation of the vector equation `z = a * x + y` requires three loops: one for the vector scaling, one for the vector addition, and a final loop to copy the vector to the target. Each of the first two operations results in a temporary that must be allocated and destroyed. A more efficient implementation is provided by the semantically-equivalent `AXPY` routine that calculates and assigns the right-hand side of the equation using a single loop and without the use of any temporaries. Fortunately, the translation from high-level algebraic expressions to equivalent fused operations, such as `AXPY`, is often mechanical and may be accomplished by *rewrite systems*. Such systems perform a syntactic match between a subexpression s and the left-hand side l of a conditional rewrite rule $l \rightarrow r \ (if \ c)$, and, in so doing, establish a substitution σ that binds s to l. If the condition c (over, for example, type constraints or alias or side-effect relations) holds, s is replaced by r, after substitution of the free variables in r through use of σ.

Gregor *et al.* [30] described the Simplicissimus system, for performing user-defined conditional rewriting. The authors stressed that the validity of the conditional is not automatically deduced. Rather, a domain expert specifies properties of variables, expressions, and procedure invocations that arise in the context of a domain library and against which the rewrite condition may be automatically

checked. Expressions on either side of a rewrite rule are specified in Simplicissimus through expression templates. A compiler's internal representation (IR) is translated to expression templates, allowing syntactic matching to be effected through partial template specialization. Rewrite rules are then simply listed as partial specializations: once an expression is represented as an expression template, the template processor automatically selects a rewrite rule that best matches the expression [76]. The target right-hand side expression template is then translated back to the compiler's internal representation to complete the transformation. For example, to perform the `AXPY` substitution, a domain expert begins by specifying a default or *primary* (class) template `AXPYMatch` that matches any expression through its type parameter and that defines a static member variable `valid` as `false`. The domain expert then defines partial specializations of `AXPYMatch` that match the scaled addition, when represented as an expression template. This specialization defines `valid` as `true`, so long as the expert-defined conditional holds, and also defines an expression template that yields the right-hand side of the rewrite rule, bound to the appropriate variable names through partial template instantiation. During optimization, an expression represented in the compiler's IR is translated to an expression template, which is then used to instantiate `AXPYMatch`. The `valid` member of the resulting, fully-instantiated template class then describes whether the expression matches the left-hand side of the rule. If so, the bound right-hand side of the rewrite rule is provided by the class as an expression template, which is subsequently translated back to the compiler's IR.

Properties, such as side-effect and alias relations, are associated with type parameters via C++ traits. Traits imbue type parameters with other types, values, and functions through the partial specialization of templates. Properties of the instantiating actual types may then be checked to ensure that the rewrite rule is valid.

In the case of `AXPY`, the validity of the rewrite rule is dependent on the type parameters representing `x` and `y` in `z = a * x + y` not having side effects, since rewriting the expression with the `AXPY` function call may lead to evaluation of `x` and `y` in an alternate order. This condition may be ensured by checking the `has_side_effects` property of the `x` and `y` type parameters. A primary, non-specialized template provides default values for properties, such as `has_side_effects`. Another interesting property of a binary operation is whether it can overflow. A primary template matches any binary expression (after conversion to an expression template) and provides conservative, default properties for a binary operation:

```
template <typename BinaryOp>
class BinaryOpTraits<BinaryOp> {
public:
  typedef __true_type has_side_effects;
  typedef __true_type can_overflow;
};
```

A specialization, such as for an array subscript operation, can then override these default properties. In this case, an array subscript operation does not have side effects and can not result in an overflow:

```
class BinaryOpTraits<Subscript> {
public:
  typedef __false_type has_side_effects;
  typedef __false_type can_overflow;
};
```

As in our approach, Simplicissimus assumes that library designers (i.e., domain

experts) are in a unique position to specify optimizations. In particular, they specify the expressions that should be rewritten and the conditions required to do so, as well as the strategy that controls the application of rewrite rules, such as first-fit or best-fit [76]. Thus, domain experts extend the semantic knowledge available to the compiler in performing optimizations [77]. It is the use of high-level abstractions that make possible the application of this semantic knowledge during optimization. Were a programmer to use a lower-level coding style or to apply compiler techniques such as type lowering or inlining, the connection between the original source code and the semantic knowledge imparted by the domain expert would be obscured.

Simplicissimus facilitates optimizations of numerical libraries through the use of concepts. For example, if the type `T`, the operation `*`, and the value `1` form a monoid (i.e., an algebraic structure with an associative binary operation and an identity element), then `x * 1` may be simplified to `x`. Here the constraints on the type of `x`, the operation `*`, and the element `1` form the condition under which the transformation may be performed. Schupp *et al.* [76] further described distributive transformations on rings as well as rewrite rules for avoiding unnecessary object copying and temporary allocation.

Dinesh *et al.* [21] found that Sophus, a C++ library providing abstractions used in the solution of partial differential equations, induced overheads similar to those encountered by Gregor *et al.* The side-effect free, algebraic coding style encouraged by Sophus is consistent with mathematical notion and improves programmer productivity and code maintainability, but requires significant spatial overhead to hold temporary and intermediate data. Through use of the CodeBoost conditional rewrite system, the authors translated the high-level algebraic

style to self-mutating code that achieved 30% better performance. Bagge *et al.* [6] showed that the Sophus practice of encapsulating the explicit use of loops leads to additional temporaries, as was found in the Simplicissimus work.

CodeBoost consists of a frontend parser, a semantic analyzer, a library of transformations, and a backend that emits transformed source code. A program is parsed via OpenC++ [14] into ATerm format and subsequently into an AST. Each transformation is implemented as a separate module that reads, transforms, and writes the AST. The transformations are applied serially before the final AST is read by the backend and used to emit the transformed program text.

CodeBoost transformations are specified either as Stratego modules [86] or as user-defined rules [5]. Stratego is a transformation language for performing rewrite steps on an AST, which is textually represented with terms, such as applications `C(`$t_1, ..., t_n$`)` of a constructor `C` to terms t_i, lists `[`$t_1, ..., t_n$`]`, strings, and integers. For example, `Plus(Var(‘‘a’’), Int(‘‘10’’))` represents a subtree for the binary expression `a + 10`. Terms may be annotated with program analysis facts that are stored and subsequently queried by a pattern match. A conditional rewrite rule then expresses a transformation on terms. A term pattern is a variable, a nullary constructor `C`, or the application `C(`$p_1, ..., p_n$`)` of an n-ary constructor `C` to term patterns p_i. As it may be cumbersome to specify rewrite rules that manipulate an AST, Stratego allows rule specification using the concrete syntax of the language of the transformed program. For example,

```
EvalPlus : |[i + j]| -> |[k]| where <add>(i,j) => k
```

specifies that `i + j` should be replaced by `k` if the expression bound to `i` and `j` sum to `k`.

Rewrite rules are applied exhaustively until none are valid during *normalization*. Since exhaustive application may not be desirable and may lead to non-termination, Stratego allows for explicit and programmable rewrite strategies, or algorithms that transform one term into another or fail to do so. Stratego provides means of composing strategies, such as sequential composition, deterministic choice, non-deterministic choice, negation, and recursion. The specification of a rule is decoupled from the specification of the strategy used to apply it.

Stratego's scoped dynamic rewrite rules overcome the limitation of purely context-free rewrite rules. Dynamic rules may be generated at run time and make use of context information. For example, Olmos and Visser [63] described how to use dynamic rewrite rules to perform data-flow transformations, such as constant propagation. An assignment of the form `x = c`, for a constant `c`, defines the rewrite rule `x -> c` and undefines any previous rewrite rules for which `x` is the left-hand side. Kalleberg and Visser [42] adapted this mechanism to propagate annotations or *totems* and apply it to propagate matrix dimensions, which may be specified by programmer assertions or may be inferred from variable initialization.

CodeBoost's user-defined rules [5] are specified in stylized C++, thus sharing the benefit of our projection approach of not requiring the domain expert to learn a new transformation language. They are often used to replace a combination of domain-specific functions with a simplified, specialized optimization. For example,

```
int x, y;
simplify: x + y = x, is_zero(y)
```

defines a rule named `simplify` that is applicable to integer variables. When a rule is applied, the structure of an AST expression is compared to the AST structure

of the match pattern, as in our approach. The local variables mentioned in the pattern, `x` and `y`, serve as meta-variables that match any expression of the specified type. If the pattern `x + y` matches, then the conditional `is_zero(y)`, which is separated from the pattern by a comma, is checked. If the conditional passes, the matched expression is replaced by the replacement pattern. Thus, the rule replaces any expression of the form `x + 0` by `x`.

Conditionals check (potentially) domain-specific properties of variables and functions. For example, `is_zero(y)` checks whether `y` holds the value zero. Implementing such conditionals statically requires that language constructs be tagged with domain-specific information and that these tags or totems be propagated throughout the program. The `CB_TAG` directive associates tags with variable names. These are included in the program text, and subsequently available for query by a transformation. A data-flow analysis propagates totems across assignment operators and drops totems when their associated variable is modified. Our attribute propagation mechanism is similar: though it is currently used to infer implicit actual arguments, it can propagate general attributes, such as the shape of a matrix. A significant difference is that the data-flow problem required by our attribute propagation mechanism is defined within the abstraction specification rather than the original program text.

The TAMPR rewrite system [10] has been used to translate numerical algorithms implemented with a functional programming language into a more efficient, imperative style. The functional language specification achieves much of the intent of our use of abstractions: both provide a more human-readable form than the lower-level, more efficient representation and aid in analysis. The high-level semantics of abstractions may be used in lieu of alias and side-effect analysis,

while the functional language specification is side-effect free. As in our approach, the TAMPR system leverages a mapping between the high-level representation and the low-level implementation. The primary difference is that TAMPR is applied to programs written using a high-level representation, while our approach optimizes programs written in a low-level language by first translating it to an abstraction space.

It is possible to define rewrite rules, such as employed in the above systems, using raising and lowering projections. In fact, the lowering of mesh iterators was completely specified in terms of projection: the mesh iteration of the original KOLAH-based implementation was rewritten to an alternate integer-based implementation with an abstraction mediating the two. However, this is an indirect means of performing rewrites. Projection is not intended to directly introduce optimizations (as it does when used for rewriting), but instead provides a common representation for optimizations that target semantics rather than syntax. Each implementation of domain semantics within a domain-specific library may be considered an extension that defines a new language. Projection then reduces different implementations of these semantically-equivalent extensions to a canonical form, which may be targeted by a single per-domain optimization without the need to tailor an optimization separately to each implementation. Therefore, the projection effectively provides a "domain-specific intermediate language" analogous to the intermediate languages of traditional compilers, such as SUIF [90], to which frontends translate syntactically distinct languages, such as C and Fortran. Since the optimization phase targets the intermediate format, optimizations are leveraged across languages.

There is a further philosophical difference between projection and rewrite sys-

tems. Projection is intended to translate collections of invocations on abstraction implementations. The invocations often cooperate to realize some higher-level behavior. For example, mesh iteration and field access are each used in the loops considered above. Projecting one or the other to abstraction space individually does not make sense since they interact with one another through common actual arguments, which must share the same target implementation. Therefore, the projection framework ensures that either all or none of such interacting invocations are raised. Rewrite systems, on the other hand, target and replace compound expressions, such as `z = a * x + y`, independently of other expressions.

A metaobject protocol (MOP) provides an alternate means of introducing transformations: rather than specifying a rewrite rule, a library rewriter associates some semantic action (such as analysis or transformation) with particular types of AST nodes. Metaobjects are responsible for translating some portion of the AST into source code. By allowing metaobjects to be specialized or subtyped, a MOP allows a library writer to interpose the default compilation process to introduce domain-specific behavior.

OpenC++ [14] and OpenJava [84] are source-to-source translation systems based on a metaobject protocol. As the two systems are similar with respect to their design, we focus on OpenC++, which translates OpenC++, an extended version of C++, to C++. The OpenC++ source code is parsed and a metaobject is created for each class and method definition. During the compilation process, OpenC++ traverses the parse tree of the program and invokes metaobjects to emit code for class definitions, member accesses, virtual function invocations, and object creations. By default, a metaobject simply emits ordinary C++ definitions, accesses, invocations, and creations. However, metaobjects may be specialized to

override this default behavior. For example, to add persistent storage to objects, `PersistentClass` may be defined as a subclass of the metaobject `Class`, which provides default translation. The extended C++ syntax then allows a C++ class to be annotated as a `PersistentClass`, such that aspects of its compilation will be dispatched to the `PersistentClass` metaobject rather than the `Class` metaobject. The `PersistentClass` metaobject is then responsible for wrapping persistent object creations to perform bookkeeping and for ensuring that an object has been loaded from stable storage before being accessed.

Polyglot [62] provides a more general approach to base language extension than a metaobject protocol by allowing interposition at arbitrary AST nodes. After the extended Java base language program is parsed into an AST, a pass scheduler selects passes to run over it. Each compilation pass potentially rewrites the AST, which is input to the next pass. The passes may extend the AST by defining new types of nodes that add syntax to the base language. Each pass is implemented by an AST rewriter object that invokes a method associated with that rewriter object at each node.

5.4 Domain-targeted Approaches

With its high-level matrix-oriented syntax, Matlab is an expressive and powerful language for quickly developing algorithm prototypes. Unfortunately, the significant overhead of Matlab with respect to Fortran, for example, is a barrier to its wide-spread use in large-scale production codes. These overheads are attributable to run-time type checking, dynamic matrix resizing, and frequent array bounds checks [54]. Several projects have targeted Matlab's inefficiencies through transla-

tion to C or Fortran [18, 19] or by direct optimization of Matlab scripts [54].

Menon and Pingali [54] noted that type and bounds checks are often redundant because they occur within loops. By vectorizing loops, they were able to eliminate the per-iteration checks. They also found that the algebraic properties of matrices can be exploited for greater efficiency. For example, computing the matrix-vector multiplication $A^T * q$ requires the spatial overhead of a temporary matrix and the execution overhead of matrix element copies for the transpose, while the equivalent $(q^T * A)^T$ expression requires neither.

FALCON [18, 19] is a Matlab to Fortran 90 translator. In order to translate an untyped Matlab script to Fortran, FALCON must perform type and shape inference accounting for intrinsic type (complex, real, integer, or logical), shape (scalar, vector, or matrix), and the size of each dimension. If the type of a variable can not be inferred at some point within the script, FALCON emits code to perform run-time type determination. FALCON type inference utilizes domain semantics, including high-level summary information about built-in functions, such as the knowledge that `lu` returns triangular matrices. It also can derive the types and shapes of procedure return values based on the known types and shapes of its input parameters. Having determined the types of actual arguments, FALCON selects specialized procedures that implement an operator more efficiently than a general-purpose procedure that can not make assumptions about its inputs.

De Rose *et al.* [18] also noted the potential use of algebraic properties, such as associativity, distributivity, and commutativity (when and where they hold), in optimizing matrix operations. For example, judicious choice of ordering of matrix multiplication can reduce the asymptotic complexity. The authors proposed interactive restructuring in which a user guides transformations by selecting ex-

pressions or statements within the application as potential targets for optimization. The FALCON system queries a database of transformation or rewrite rules, such as those exploiting algebraic properties, and applies any matching the user-selected code fragment.

MaJIC presents a Matlab-like interactive frontend for performing just-in-time compilation and optimization and speculative ahead-of-time compilation [2]. Like FALCON, it infers intrinsic type, shape, and range information for callsite specialization. When the frontend encounters a function call, it creates a summary of the function name and inferred argument types and queries a code repository, or database, for the most appropriately specialized procedure implementation. If no valid procedures are stored within the repository, MaJIC uses its just-in-time compilation facility to compile the procedure.

The repository actively snoops source code directories upon update to speculatively compile procedures. To avoid an exponential explosion in procedure specializations, MaJIC performs type speculation to determine the procedure specializations that are likely to be invoked. Type determination need not by exact since it drives speculative code generation rather than procedure invocation. During speculation, type hints derived from the procedure body are propagated back to its input parameters. These hints are based on Matlab common practices, such as the expected, though not required, use of integers as operands to the colon operand used to specify an interval or range.

Each of these approaches to Matlab optimization or translation use some high-level semantic information; for example, algebraic properties suggest transformations and the known semantics of built-in operators dictate type and shape information. However, these domain-targeted schemes are intimately tied to Matlab

and the semantics of linear algebra, which tend to be hard-coded directly within the compiler. Such lack of extensibility prevents application of these approaches to new libraries or toolkits that do not make frequent use of the linear algebra subset of Matlab.

Our use of projection between concrete implementation and abstraction spaces is similar to the approach Menon and Pingali used to optimize numerical codes [55]. The authors converted Matlab or Fortran loops into an Abstract Matrix Form, or AMF. Axioms on AMF express the semantics of matrix, vector, and element-wise operations. By establishing that certain AMF expressions are provably equivalent, the authors allowed for the translation of low-level loop-based scalar code into vector code. After reasoning about transformations in the abstract space, AMF is translated back to Matlab or Fortran.

A high-level intermediate form has also been applied by Mateev *et al.* [51] in the optimization of sparse matrix codes. The authors proposed using two APIs, a high-level interface for expressing generic algorithms and a low-level interface for exposing implementation details necessary to obtain high performance. An algorithm written to the high-level API addresses the matrix using random access. This algorithm is then transformed to an intermediate form in which loops are modeled using a relation algebra, which is subsequently optimized as a set of relational queries. The optimized intermediate form is then output as invocations on the low-level interface, which provides efficient, sequential access to matrix elements.

The above compiler-directed approaches target legacy codes by translating their concrete syntax into an abstract form with known semantics. Alternately, language-level extensions or library routines may be defined to express particular

semantics. For example, Kulkarni *et al.* [49] introduced a set iterator that asserts that iterations may be executed in any order and used it to implement Delaunay mesh refinement. The authors found that loop parallelization requires considering the commutativity of operations on the set accessed across iterations. The internal state of the shared set may differ for different execution orders, though involved operations still may commute in the semantic sense. Therefore, the authors described the set's operations in terms of their semantic effect, rather than their implementation. Doing so allowed them to optimistically execute loop instances in parallel. The commutativity of set operations is verified at runtime, with commutativity conflicts triggering a rollback.

5.5 Broadway

Broadway [33, 35, 34] takes a philosophy similar to our own, viewing a library as a domain-specific language whose procedures often must be treated as black boxes since they are unknown to the compiler. A sophisticated annotation language [33] and abstract interpretation mechanism supplement the compiler's limited understanding of an application. Broadway offers a richer set of analyses and a more expressive data-flow problem description for propagating abstraction properties than currently available in ROSE. However, what the authors perceive as a strength, a transformation scheme utilizing macro-based code substitutions, we view as a limitation. While writing transformations within open or extensible compiler frameworks, such as our own, may be more cumbersome for simple optimizations readily expressed in terms of pattern matching and substitution, we believe that global transformations and analyses require the more flexible approach afforded by direct AST inspection and traversal.

Broadway is a source-to-source translator for C written in C++ [34]. The backbone of Broadway is its annotation language [33]. Low-level analysis information such as variable definitions and uses may be specified via `modify` and `access`. Higher-level, domain-specific information is specified with the `property` keyword, which may introduce a lattice of enumerations or a set [35]. `analyze` annotations are used to define a data-flow problem describing abstract interpretation over properties. Analysis annotations are implications: if the left-hand conditional holds, the right-hand side is evaluated to potentially modify some property, for example, by adding or removing a member from a set. By leveraging `analyze` clauses, procedures act as data-flow transfer functions modifying the abstract properties. Broadway solves the data-flow equations to propagate properties throughout the program, which may then be evaluated within conditionals to trigger optimizations as annotated by `replace-with` or `inline`. Conditions may test for a particular enumeration value, whether an element is a member of a set, if numerical relation such as equality holds over quantities derived through constant propagation, or whether an aliasing or equality exists between variable bindings.

Transformations act on callsites and may either replace a callsite with a user-specified C code fragment or may indicate that the library procedure should be inlined if the source code is available [35, 34]. Earlier work on Broadway allowed simple pattern matching based transformations [36] and a mild variant of the current scheme using the `specialize` keyword to either `replace` or `remove` code fragments [33]. The C-Breeze front-end parses the fragment to ensure that it is valid C code before introducing it into the program.

Optimization within Broadway relies on a domain expert to specify abstract properties, how library routines affect them, and the code transformations predi-

cated on them [34]. Broadway first performs pointer analysis informed by `on_entry` annotations describing the pointer structure of procedure parameters and `on_exit` annotations indicating changes to parameters or the structure of the return value resulting from the procedure's execution. It next solves the data-flow problem instantiated by `analyze` to propagate abstract properties throughout the program. The precision of Broadway's client-driven analysis is tuned to the requirements of a client based on perceived loss of information and is flow- and context-sensitive. Following analysis, Broadway applies a series of enabling transformations such as procedure integration, procedure cloning, loop peeling, and node splitting before specializing callsites according to annotations and finally performing traditional optimizations.

5.6 Telescoping Languages

Similar to our own work, the telescoping languages project [44] focuses on the optimization of libraries and their usage within scientific domains. More specifically, the work understands that high-level scripting languages, such as Matlab or Mathematica, increase programmer productivity by providing language constructs consistent with notations familiar to scientists. Unfortunately, prototypes built in high-level scripting languages often need to be recoded in languages such as C, C++, or Fortran to meet the performance requirements of large-scale production systems. Such tedious, manual translation negates the initial expediency gained from using a high-level language. The telescoping languages project seeks to obviate this recoding step by making the performance of scripting languages commensurate to that of more conventional programming languages.

Performance of scripting languages is degraded because the script interpreter or compiler has no semantic understanding of the invoked libraries. Reasoning that the domain libraries will be re-compiled relatively infrequently, work on telescoping languages [44, 45] invests considerable analysis time and complexity to optimize the libraries. Because this increased *library* compilation time does not burden *script* compilation, the user sees improved performance without undue compilation overhead.

The initial library analysis and preparation phase effectively acts as a language generation phase by creating a recognizer and optimizer for a telescoping language. During this phase a domain library is extensively analyzed by the Palomar translator generator. The domain library may itself be written in a scripting language, in which case it is parsed by a domain script translator and translated into a "base language", such as C or Fortran. Palomar employs powerful interprocedural analyses, informed by annotations supplied by domain experts, that would be prohibitively expensive if applied at script compilation time.

The library-aware optimizer produced by the library analysis and preparation phase treats library entry points as language primitives. Thus it is effectively a recognizer for a new "telescoping language" that is the union of the original base language and library entry points. During the script compilation phase, a domain script is translated into the base language by a domain script translator and presented to the library-aware optimizer, which ultimately produces optimized source code in the base language. This process may be repeated recursively, with a mature script being passed as input to the library analysis and preparation phase; thus the scheme can "telescope" hierarchies of libraries into a single optimizer.

As in Broadway, the most frequent transformation performed by the library-

aware optimizer is callsite specialization informed by calling context. Procedures are most often specialized according to high-level type and shape information. For example, the shape of a matrix, such as tridiagonal, sparse, symmetric, or diagonal, may allow a more efficient implementation of an operator or algorithm: while a typical eigensolver has $O(n^3)$ complexity [26], one acting on a tridiagonal matrix has $O(n^2)$ complexity [20]. The library analysis and preparation phase produces procedures specializations and stores them in a database for use during the script compilation phase.

Because Matlab and S scripts are untyped, performing type-based specialization first requires assigning types to each variable at each statement. The domain script translator employs a sophisticated type analysis that generates a set of valid type configurations [52], each of which potentially admit a unique specialization. During type inference, the library analysis phase creates type jump functions and return type jump functions. The former assign types to local variables within a procedure based on the types of its formal parameters, while the latter assign the return type of a procedure again as a function of the types of its input arguments. Use of these functions allow for the efficient propagation of type information without a need to recompute it; thus the majority of the burden of type analysis is borne by the library analysis phase, whereas the compilation phase simply consults a table.

A data-flow analysis propagates inferred properties or semantics supplied by a domain expert. These properties include the types and values of parameters and are used to specialize routines. *Reverse program analysis* reasons back from potential targets of optimization within a procedure to restrict types at the entry point of a specialized version of the procedure. Having guaranteed the type pre-

conditions required by the optimization, the specialized procedure can implement it. The resulting specialized procedures are stored in a database from which they can be selected and inlined into scripts during script compilation.

As in our work, research in telescoping languages recognizes that the domain expert writing the library is best suited to provide semantic annotations about that library. Envisioned annotations include algebraic identities and inverse operations indicating, for example, that a push onto a stack followed immediately by a pop is a no-op and that both operations may be omitted. Annotations may also assist type analysis by limiting an inferred set of possible types to a more restricted set that occurs in practice. Annotations specify substitutions, as in Broadway, of one code segment for another predicated on a specific calling context. Early implementations of telescoping languages [13] used XML to specify code to replace any code sequence matching a query pattern. This substitution is triggered within the specified context assuming that no dependences are violated. In addition, a query pattern may contain variable, constant, and statement wild cards. Both the substitute and the match sequences are comprised of simple statements, loops, and two-way and multi-way branches. In addition, the annotation may specify the profitability, in either a qualitative or quantitative sense, of applying a transformation.

5.7 ROSE-related Abstraction Optimization

From its inception, automatic generation of domain-specific grammars has been a goal of the ROSE project [66, 68, 72, 73]. In early work, ROSETTA, currently used to specify the C++ grammar, automatically generated grammars used to rec-

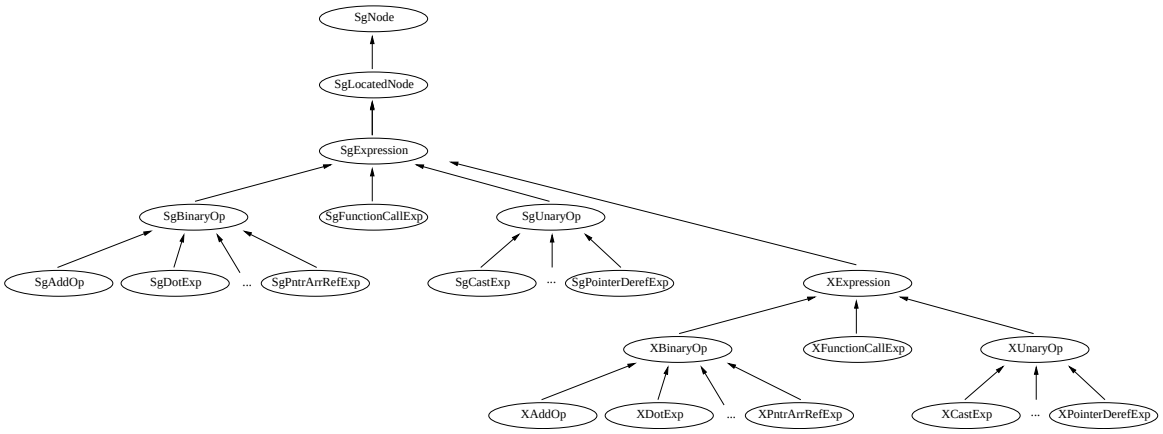

Figure 5.1: ROSETTA abstraction inheritance.

ognize abstraction defined within libraries [68]. Library header files were parsed to discover library class definitions and functions. The resulting abstraction grammar contained a shadow set of grammar variants; for example, for a library `X` ROSETTA would generate an `XNode` for every `SgNode` [66]. This simplified the recognition of abstractions within the AST because each was clearly labeled as belonging to a library or to the base language; a function call represented by a `SgFunctionCallExp` invokes a function defined within the base language, while a `XFunctionCallExp` invokes a function or method defined within the `X` library. A subset of the shadow variants that would be generated by ROSETTA are shown in Figure 5.1.

A recognizer generated by ROSETTA propagates abstraction references up to their enclosing expression or statement. For all `SgExpressions`, such as `SgFunctionCallExp` and `SgBinaryOp`, ROSETTA defines a corresponding `XExpression`. An AST node is labeled with an `XExpression` rather than a `SgExpression` only if the types of all references within the expression are abstraction types. Similarly, an `XStatement` is so labeled only when all of its expressions are `XExpressions`. For example, the node representing the statement `A = B +`

`foo();` is marked as library specific if the types of `A` and `B` are a user-defined abstraction and `operator=`, `operator+`, and `foo` are procedures acting on these or some other user-defined type [73]. In this regard, this approach is similar to the raising projection: an abstraction implementation is raised to abstraction space only if all of its actual arguments can be similarly raised or if their type is invariant across implementation spaces.

The higher-level abstraction grammar is defined relative to the base-language grammar according to a set of constraints, which are expressed as C++ code strings [66]. For example, Quinlan and Philip [68] describe a higher-level grammar for array abstractions, in which an abstraction array type within the grammar is defined as a class type in the base-language grammar with the additional constraint that the class type is named "doubleArray." The authors also add terminals corresponding to each public member function of the array class, thereby introducing these methods to the language definition.

A subset of the array abstraction grammar is defined programmatically in Figure 5.2 [67]. The new `doubleArrayType` type is included in the abstraction grammar by adding a corresponding terminal to the base-language grammar. Abstraction grammars may also remove terminals from the base grammar. This abstraction grammar definition can be automatically generating by parsing the `doubleArray` class definition.

Line 4 constructs a representation of the C++ grammar, which may be parsed by the recognizer built in line 8. The `X_Grammar` for array abstractions is specified as a child grammar of the base-language grammar by line 13, which indicates that the AST nodes of `X_Grammar` should be prefixed with X. The grammar adds a single terminal to the base-language grammar in line 23. The terminal is a copy of

```
    // Build the C++ grammar.
 2
    // For base-language grammar use prefix "Sg"
 4  Grammar sageGrammar("Cxx_Grammar","Sg","ROSE_BaseGrammar");

 6  // Build the header files and source files representing the
    // grammar's implementation.
 8  sageGrammar.buildCode();

10  // For the higher-level grammar use any prefix but "Sg" to
    // avoid namespace collision. Specify the parent grammar if
12  // it exists (this defines the hierarchy of grammars).
    Grammar X_Grammar("X_Grammar","X_","ROSE_BaseGrammar",&sageGrammar);
14
    // Build a new terminal as a copy of an existing terminal,
16  // giving it a new name. The copy is then a child of the
    // copied terminal: parsing the parent triggers the parsing of the
18  // children (constraints are tested and a child is built if a
    // constraint test passes, else the parent is built). In the tree
20  // hierarchy the new terminal is DERIVED from the parent (thus the
    // doubleArrayType is derived from the ClassType).
22  Terminal &doubleArrayType =
       X_Grammar.getTerminal("ClassType").copy("doubleArrayType");
24
    // Build a constraint and add it to the new terminal.
26  char* constraintString = "isSgClassDeclaration() &&
        isSgClassDeclaration()->getName() == \"doubleArray\"";
28  doubleArrayType.addConstraint("declaration",constraintString);

30  // Add Terminal to Grammar (to the X_Type branch)
    // ("OR" the new terminal with the existing terminals)
32  // X_Grammar.getNonTerminal("X_Type") |= doubleArrayType;

34  // Adding a terminal to the grammar will automatically place the
    // terminal in the correct location within the tree hierarchy.
36  X_Grammar.addNewTerminal(doubleArrayType);

38  // Build the header files and source files representing the
    // grammar's implementation.
40  X_Grammar.buildCode();
```

Figure 5.2: ROSETTA-based definition of abstraction grammar.

`SgClassType`, which is used to represent types introduced as C++ classes, and is given the name doubleArrayType. Line 28 adds a constraint to the new terminal, defining it as a `SgClassDeclaration` whose name is `doubleArray`. The grammar specification is completed by adding the terminal to the AST type hierarchy. This may be done explicitly as in line 32, which adds the new terminal to the right-hand side of the `X_Type` non-terminal, thus making it a subtype of `X_Type`. Alternately, the new terminal may be automatically situated in the type hierarchy as in line 36. Finally, line 40 builds the recognizer for the newly defined abstraction grammar.

A hierarchy of grammars results from defining an abstraction grammar relative to another grammar through the addition or removal of terminals. Parsing of a parent node recursively checks any constraints that were added to a copy of that node to define a child node in a sub-grammar. If a child node's constraints pass, a child node is built in place of the parent node. Because a child node is derived from a parent node, an optimization may act on any level of the AST hierarchy. Therefore, modifications at one conceptual level of the AST hierarchy are automatically visible at all levels.

The primary drawback of the ROSETTA-based approach is its scalability: for each new abstraction, it introduces not only a complete new set of variants, but also the hundred or so lines of code required to recognize each variant. Another drawback is the relationship it imposes on Sage variants and abstraction variants. The ROSETTA approach creates `XExpression` as a subtype of `SgExpression`. Therefore, any optimization or analysis targeting a `SgExpression` applies equally well to a `XExpression`. Unfortunately, this is not true of the subtypes of `XExpression`: since `XBinaryOp` is not a subtype of `SgBinaryOp`, a transformation targeting binary operators would have to be implemented redundantly for Sage and abstraction

nodes, even when the semantics for binary operators over abstractions are consistent with those in the base language. `XBinaryOp` could conceivably multiply inherit from `XExpression` and `SgBinaryOp`, though this would sacrifice the simplicity of an inheritance tree for an inheritance graph.

During the course of this work, we considered defining a shared pseudo-base class of both `SgExpression` and `XExpression`, `SgExpressionAbstraction`, to be used whenever one would like to ambiguously refer to a base-language or abstract expression. In point of fact, `SgExpressionAbstraction` subsumed `SgExpression` and `XExpression`, as defined in the ROSETTA approach, and had no explicit language-level inheritance relation with `SgExpression`.

These approaches extend the base language to include abstraction-level constructs. This is beneficial since it allows expressions on abstractions, e.g., the matrix-vector equation `A * x + b`, to be treated like semantically-similar expressions in the base language, e.g., the integer equation `a * x + b`. This would facilitate, for example, the transformation of matrix-vector equations using traditional optimizations intended for scalars, including partial redundancy elimination and algebraic simplification. Therefore, these approaches should be of value to mathematical abstractions, which can be readily mapped to language-level expressions such as addition and multiplication.

Operations on abstractions outside the linear algebra domain, e.g., mesh iteration, do not correspond well to language-level constructs. While `KOLAH`-based mesh iteration was lowered to an integer-based implementation using only language-level expressions, those low-level constructs do not reflect the semantics of iteration in the same manner in which a base language's addition operator generalizes to matrices. Therefore, representing operations on arbitrary abstractions within an

extended grammar seems to be of little value. For example, mapping mesh abstractions and the operations on them to new AST nodes would not aid mesh precomputation in determining those expressions that access mesh connectivity metadata. We believe that expressing abstraction semantics that do not correspond to the semantics of base language constructs is better achieved through the projection approach. This framework presents abstractions to analyses and optimizations in a canonical, function-based form without attempting to force an artificial correspondence to base language constructs.

Work within ROSE leveraging domain semantics to guide transformations has proceeded independently of the above work on ROSETTA-based abstraction recognition [70, 71, 69, 91]. This work relies on semantics as specified by a domain expert or library writer through an increasingly sophisticated and maturing annotation interface, which has been informed by our earlier work [89].

Quinlan *et al.* [70] introduced OpenMP pragmas into serial code to parallelize loops over user-defined containers. To ensure the correctness of this transformation, the authors needed to guarantee that no dependences exist between loop iterations. For their examples, this guarantee is provided by a developer-specified assertion that containers obey "Fortran array semantics", i.e., their elements are unique. Traditional dependence-based approaches would be insufficient to navigate user-defined abstractions to derive this container property.

Yi and Quinlan [91] and Quinlan *et al.* [71] presented an extended loop transformation framework that applies interchange, fusion, and blocking to user-defined containers. They extended a traditional dependence analysis with an array abstraction interface, through which they communicate container semantics including the uniqueness of container elements. By widening the interface to dependence anal-

```
class floatArray :
inheritable is-array { dim = 6;
    len(i) = this.getLength(i);
    elem(i$x:0:dim-1) = this(i$x);
    reshape(i$x:0:dim-1) = this.resize(i$x); };
operator floatArray::operator()(int index) :
inline { this.elem(index) };
restrict-value { this = { dim = 1; }; };
```

Figure 5.3: Container annotation language.

ysis, they bring the community's significant investment in loop optimization for Fortran to bear on user-defined containers.

Their approach is consistent with Figure 1.2 because it maps a concrete implementation to an abstract intermediate form, which is analyzed without recourse to the implementation. For example, the annotation for the `floatArray` class in Figure 5.3 [91] declares that it has array semantics, which are pre-defined by the annotation language to include the `dim` attribute and the `len`, `elem`, and `reshape` methods. The class annotation maps the abstract method `elem` to the concrete method implementation `operator()`. This relation, along with all others supplied in the class annotation, are inherited by any subclasses of `floatArray` because of the `inheritable` keyword. A reciprocating annotation for `operator()` establishes its semantic equivalence to the abstract `elem` via the `inline` keyword. `restrict-value` assigns a value to a property, previously specified via `has-value`, based on the procedure context. For example, since a `floatArray` is accessed via a single dimension by `operator()(int index)`, it must have dimensionality one.

Our approach to abstraction optimization, particularly establishing and exploiting the projection between spaces, was inspired by the work of Yi and Quinlan [91]. However, their annotation language was defined specifically to describe

```
  class Node : has_value { id = this.id(); }
 2 class Edge : has_value {
    n1 = this.node1(); n2 = this.node2();
 4 };
  class Mesh : has_value {
 6  nsize = this.node_size(); esize = this.edge_size();
    nodes(i:0:nsize) = this.get_node(i);
 8  edges(i:0:esize) = this.get_edge(i);
  };
10 restrict_value { nodes(i).id != nodes(j).id; }
  never_alias (edges(i).n1) = edges(i).n2;
12 never_alias (edges(i)) = edges(j) : j != i;
  never_alias (nodes(i)) = nodes(j) : j != i;
14 must_alias(nodes(j)) = edges(i).n1 or edges(i).n2;
  restrict_value { esize >= nsize * k1 : esize <= nsize * k2 }
```

Figure 5.4: Mesh annotation language.

container semantics and mandates its interface, whereas our approach allows a domain expert to define the interfaces of any number of abstractions. Further, the authors translate an AST (i.e., the original implementation) to an intermediate form in order to optimize it, whereas our approach allows for distinct original and target implementations. The first step in their optimization process replaces low-level implementations with abstract procedures through semantic inlining. This results, for example, in arrays for which all dimensions are explicitly specified as opposed to C pointers whose single "dimension" would hide array semantics and prevent optimization. An adapted constant propagation algorithm, similar to our data-flow-based attribute propagation, computes properties, introduced by `has-value` and refined by `restrict-value`, to inform loop transformations. The final step generates low-level code by translating the abstract procedures back to concrete implementations.

Later work [69] generalized this container annotation language to accommodate unstructured mesh semantics, motivated in part by our earlier findings [89]. These semantics indicate that a node has an identifier, an edge is comprised of two

nodes, and a mesh has fixed numbers of nodes and edges as well as means of accessing them, as shown in Figure 5.4. This specification assumes that the mesh provides random access to mesh elements, which facilitates making assertions over a collection. For example, line 12 states that no two edges in a collection are the same. The annotation implicitly uses universal quantification and formalizes the high-level concept of a set, that is, a collection with unique elements, through a low-level analysis statement about aliases. The latter use of compiler terminology will allow semantics to be readily incorporated into existing analysis frameworks.

Our current work embeds semantics within abstraction documentation, as done in STL, rather than explicitly codifying them with an annotation language. Just as STL programmers must respect the semantics of the routines they use, optimizations must respect the semantics of abstractions. In future work, we anticipate explicitly annotating semantics and, as described above, could easily use existing facilities to tag abstraction procedures with simple side-effect and alias annotations. However, the present work focuses primarily on the importance of several traditional and novel optimizations, the semantics that enable them, and a framework for their automation, rather than on the representation of semantics. Originally, we envisioned that semantics, such as those of Figure 5.4, might lead a compiler framework to *infer* that lowering was possible. Instead, by providing two alternate implementations of mesh iteration, the domain expert obviates the need for many of these annotations.

Chapter 6

Conclusion

High-level abstractions, such as matrices and fields, improve programmer productivity because they more closely resemble the mathematical notation familiar to scientists than low-level constructs that expose implementation details. While the implementations of individual operations acting on these abstractions are generally very efficient, achieving good performance across such operations is challenging. Whereas the compiler can infer behavior of low-level constructs through side effect and alias analyses, because it does not recognize abstraction invocations, it must often conservatively treat them as black boxes. This can hinder optimization and lead to poor performance. For example, we have found that a representative mesh benchmark executes more than eight loads or stores and more than two branches per floating point operation.

Following the lead of pioneering work on the optimization of high-level abstractions by Guyer and Lin [33, 35, 34] and the research groups of Kennedy [44, 45], Pingali [1, 48, 51, 54, 55], and Quinlan [91, 71], we propose a framework for *representation-independent* optimization that targets an abstraction's semantics, rather than its implementation. These approaches treat the constructs within the input program as abstractions to be optimized or recognize specific constructs (e.g., matrix accesses) and translate them to a higher-level intermediate representation (IR) (e.g., database queries) that is subsequently optimized. Our approach differs in that the abstraction space, which defines the IR, is not fixed by the optimization framework, but is specified by a domain expert who also provides optimizations acting within that abstraction space. Thus, while we instantiate several unstructured mesh optimizations within this framework, it is effectively an open compiler

infrastructure that exposes for optimization a view of the program expressed in a developer-defined IR. As such, it may be applied to other domains leveraging abstractions.

6.1 Contributions

This thesis has made specification contributions to the optimization of unstructured mesh codes by quantifying overhead in a benchmark implemented within the `KOLAH` framework and by discussing mesh semantics that we expect generalize beyond `KOLAH`. We anticipate that this characterization will help developers and compiler groups to better understand and optimize mesh applications. We have used these findings to motivate several domain-specific optimizations, including mesh precomputation and lowering. Though the `testhydro1` benchmark implemented within `KOLAH` served as our specific testbed, we believe our findings and domain-specific optimizations generalize beyond each. For example, though mesh precomputation targeted gradient and divergence operators, it is amenable to operators likely to be employed in other dynamical simulations. These include curl and averaging operators, which are implemented in the production library from which `KOLAH` was derived and which share the loop nesting structures and loop exit conditions of the gradient and divergence operators. Though other dynamical solvers may incorporate theories much different from the Lagrangrian formalism used by `testhydro1` in solving the Euler equations, we expect them to employ similar programming motifs. In particular, solving a system of equations over a volume or surface necessarily entails iterating over the mesh. Simple loops that access fields will therefore benefit from lowering, while nested loops will additionally benefit from mesh precomputation.

This thesis has made a general contribution to semantics-based optimization of abstractions by proposing a framework that projects abstractions from an implementation space, where optimization is limited by the bounds of traditional compiler analysis, to an abstraction space, where optimization and analysis are informed by semantics. As the projection, or mapping, between the two spaces is defined by a domain expert, the system is not restricted to the unstructured mesh domain considered here. We described a data-flow analysis for assigning, propagating, and querying variable attributes and applied it to the inference of actual parameters. Because of this analysis, implementations of an abstraction need not conform to a single interface. Rather, parameter inference can be used to bridge the interface differences between an implementation within which a parameter is implicit (e.g., the container implicitly associated with an STL iterator) and a second within which it is required (e.g., the vector associated with an integer index).

6.2 Future Work

The current approach follows the STL convection of providing abstraction semantics within their specification documentation and expects that optimizations written by domain experts respect them. These semantics could be more explicitly encoded within the specification via annotations, such as those used to indicate that a procedure abstraction is a conversion operator. Once semantics are made explicit, it may be possible to verify their correctness—i.e., to ensure that the implementation of an abstraction adheres to its declared semantics. Doing so is more practical at the analysis level, rather than at the algorithmic level. However, verifying properties such as alias and side-effect relations may be infeasible given

the state of compiler analysis technology. The ability to exploit domain expertise to overcome this limitation of traditional analysis should therefore be considered a strength, rather than a shortcoming, of this approach. As such, related transformation systems typically rely on the validity of communicated domain expertise [76, 6].

Nevertheless, it is possible to check that abstractions defined in libraries are used in accordance with their specifications. STLlint [29] uses symbolic execution to check uses of STL (e.g., to verify that an invalidated iterator is not accessed and to ensure a binary search is not applied to an unordered sequence). Violation of abstraction semantics are not language-level errors, since the program is syntactically correct. Therefore, STLlint analyzes source code and replaces any types and functions that have specifications with an executable form of the specification. These executable (and analyzable) abstractions are then analyzed to ensure proper use. For example, an executable specification for a heap sort routine might tag the sequence as being sorted. A binary search routine would then assert that its sequence argument has a sorted tag, which might subsequently be removed (e.g., by a routine that appends an element onto the sequence). STLlint uses fixed-point iteration and symbolic differencing in performing loop analysis. The authors do not attempt to prove the correctness of the specifications passed to STLlint. Though we perform no such abstraction use checking (and lack the sophisticated loop analysis to do so), the projection mechanism could be used as a necessary first step in replacing implementations with their executable specifications.

The projection framework can be applied to alternate implementations of mesh abstractions and to alternate domains. Porting the framework to an alternate implementation requires mapping it to the mesh abstractions used in this thesis

through appropriate lowering operators. Doing so will allow mesh precomputation and lowering to target the new implementations. Applying the projection framework to new domains involves defining an abstraction interface and elucidating the semantics that span implementations. The availability of an open compiler framework that targets the commonalities of abstraction implementations will allow coordination of these and other optimization efforts across research groups.

BIBLIOGRAPHY

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. 14th ACM International Conference on Supercomputing*, pages 141–152, May 2000.

[2] G. Almasi and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, May 2002.

[3] W. Anderson, W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 1999.

[4] M. H. Austern. *Generic Programming and the STL: Using and Extending C++ Standard Template Library.* Addison Wesley, 1999. ISBN: 0201309564.

[5] O. S. Bagge and M. Haveraaen. Domain-specific optimisation with user-defined rules in codeboost. In *Workshop on Rule-Based Programming (RULE 2003)*, June 2003.

[6] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, September 2003.

[7] F. Bassetti, K. Davis, and D. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. In *Proc. International Symposium on Computing in Object-Oriented Parallel Environments*, pages 107–118, Dec. 1998.

[8] M. W. Beall and M. S. Shephard. An object-oriented framework for reliable numerical simulations. *Engineering with Computers*, 15(1):61–72, Apr. 1999.

[9] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and c++ restructuring tools. In *Proc. Second Annual Object-Oriented Numerics Conference*, pages 122–138, Apr. 1994.

[10] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, 1997.

[11] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *Proc. 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 183–200, 1998.

[12] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. SIGPLAN Symposium on Compiler Construction*, pages 98–101, June 1982.

[13] A. Chauhan and K. Kennedy. A source-level matlab transformer for DSP applications. In *Proc. 6th IASTED International Conference on Signal and Image Processing*, pages 92–101, Aug. 2004.

[14] S. Chiba. A metaobject protocol for C++. In *Proc. 10th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 285–299, 1995.

[15] T. Chilimbi, B. Davidson, and J. Larus. Cache-conscious structure definition. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, May 1999.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, 1991.

[17] R. Das, M. Uysal, J. H. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, 1994.

[18] L. DeRose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. Falcon: A matlab interactive restructuring compiler. In *Proc. 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288, Aug. 1995.

[19] L. DeRose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. Prog. Lang. Syst.*, 21(2):286–323, 1999.

[20] I. Dhillon. A new $o(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem. Technical Report UCB/CSD-97-971, University of California at Berkeley, May 1997.

[21] T. B. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*, 8(4):247–259, 2000.

[22] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proc. ACM SIGPLAN*

Conference on Programming Language Design and Implementation, pages 229–241, May 1999.

[23] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM Symposium on Principles of Programming Languages*, pages 295–308, 2006.

[24] Edison Design Group. C++ front end internal documentation (excerpt). `http:www.edg.com/docs/edg_cpp.pdf`, Jan. 2006.

[25] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proc. 18th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 115–134, Nov. 2003.

[26] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1996. ISBN: 0801854148.

[27] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 2005. ISBN: 0321246780.

[28] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in c++. In *Proc. 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 291–310, Oct. 2006.

[29] D. Gregor and S. Schupp. STLlint: Lifting static checking from languages to libraries. *Software – Practice and Experience*, 36(3):225–254, 2006.

[30] D. Gregor, S. Schupp, and D. R. Musser. Design patterns for library optimizations. In K. Davis and J.Strieglitz, editors, *Proc. International Conf. on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'01)*, pages 309–320, 2001.

[31] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit cfd codes. In *Proc. of Parallel CFD'99*, May 1999.

[32] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Performance modeling and tuning of an unstructured mesh cfd application. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2000.

[33] S. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proc. 2nd USENIX Conference on Domain-Specific Languages*, pages 39–52, Oct. 1999.

[34] S. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proc. IEEE*, 93(2), Feb. 2005. Special Issue on Program Generation, Optimization, and Platform Adaption.

[35] S. Z. Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, University of Texas at Austin, 2003.

[36] S. Z. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *Proc. 13th International Workshop on Languages and Compilers for Parallel Computing*, pages 227–243, Aug. 2000.

[37] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *Proc. of Workshop on Languages, Compilers, and Runtime-Systems for Scalable Computers*, pages 70–84, 2000.

[38] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, Apr. 2002.

[39] IBM Corporation. HPM Tool Kit – Home Page. http://www.alphaworks.bim.com/tech/hpmtoolkit, Jan. 2005.

[40] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming - challenges of constrained generics in C++. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 272–282, June 2006.

[41] G. Jin and J. Mellor-Crummey. Experiences tuning smg98–a semicoarsening multigrid benchmark based on the *hypre* library. In *Proc. 16th ACM International Conference on Supercomputing*, pages 305–314, June 2002.

[42] K. T. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. In *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, pages 18–31, March 2007.

[43] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *PLDI01*, pages 1–12, 2001.

[44] K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proc. 14th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 297–304, May 2000.

[45] K. Kennedy, B. Broom, A. Chauhan, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proc. of the IEEE*, 93(2):387–408, Feb. 2005.

[46] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, Dec. 2001.

[47] B. S. Kirk. Computational meshes–an overview and unstructured mesh data structures. `http://www.cfdlab.ae.utexas.edu/~benkirk/`, Apr. 2003.

[48] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.

[49] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, June 2007.

[50] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[51] N. Mateev, K. Pingali, and V. Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proc. 14th ACM International Conference on Supercomputing*, May 2000.

[52] C. McCosh, A. Chauhan, and K. Kennedy. Domain-specific type inference for library generation in a telescoping compiler. Number TR04-434, 2004.

[53] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 28(3), June 2001.

[54] V. Menon and K. Pingali. A case for source-level transformations in matlab. In *Proc. of 2nd Conference on Domain-specific Languages*, pages 53–65, Oct. 1999.

[55] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proc. 13th ACM International Conference on Supercomputing*, pages 434–443, June 1999.

[56] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991. ISBN: 0132479257.

[57] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. ISBN: 0262631814.

[58] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proc.IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 192–202, 1999.

[59] T. Mohan, B. de Supinski, S. McKee, F. Mueller, A. Yoo, and M. Schulz. Identifying and exploiting spatial regularity in data memory references. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, page 49, Nov. 2003.

[60] P. J. Moran. Field model: An object-oriented data model for fields. Technical Report NAS-01-005, NASA Ames Research Center, 2001.

[61] D. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software – Practice and Experience*, 24(7):623–642, 1994.

[62] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 138–152, Apr. 2003.

[63] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 204–220, 2005.

[64] J. Owen. An open-source project for modeling hydrodynamics in astrophysical systems. *Computing in Science and Engineering*, 3(4):54–59, Nov/Dec 2001.

[65] S. Peyton Jones. *Haskell 98 Language and Libraries (The Revised Report)*. Cambridge University Press, 2002. ISBN: 0521826144.

[66] D. Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proc. of Conference on Parallel Compilers (CPC2000)*, volume 10 of *Parallel Processing Letters*. Sringer Verlag, Jan. 2000.

[67] D. Quinlan. ROSE Source Distribution: src/ROSETTA/array.C.two_grammars, 2006.

[68] D. Quinlan and B. Philip. Rosetta: The compile-time recognition of object-oriented library abstractions and their use within applications. In *Proc. 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2001.

[69] D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi. Annotating user-defined abstractions for optimizations. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL'06)*, Apr. 2006.

[70] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *Proc. 16th International Workshop on Languages and Compilers for Parallel Computing*, Oct. 2003.

[71] D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen. Classification and utilization of abstractions for optimization. In *Proc. of First International Symposium on Leveraging Applications of Formal Methods (ISoLA'04)*, Oct. 2004.

[72] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan. Treating a user-defined parallel library as a domain-specific language. In *Proc. 16th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 105–114, Apr. 2002.

[73] D. J. Quinlan, M. Schordan, B. Miller, and M. Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 16(2-3):293–302, Feb. 2004.

[74] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *Proc. 12th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 81–87, Mar. 1998.

[75] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference (JMLC'03), Lecture Notes in Computer Science*, pages 214–223, Aug. 2003. vol. 2789, Springer-Verlag.

[76] S. Schupp, D. Gregor, D. Musser, and S. Liu. Library transformations. In M. Hamann, editor, *Proc. IEEE International Conf. Source Code Analysis and Manipulation*, pages 109–121, 2001.

[77] S. Schupp, D. Gregor, D. Musser, and S. Liu. User-extensible simplification—type-based optimizer generators. In R. Wilhelm, editor, *International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 86–101, 2001.

[78] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, May 1997.

[79] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, pages 59–70, 1998.

[80] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 73–84, June 2005.

[81] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, Nov. 1995.

[82] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–102, June 2003.

[83] M. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[84] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. Openjava: A class-based macro system for java. In *Proc. of 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, 1999.

[85] J. S. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2002.

[86] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[87] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[88] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–27, Nov. 1998.

[89] B. White, S. McKee, B. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the computational intensity of unstructured mesh applications. In *Proc. 19th ACM International Conference on Supercomputing*, pages 341–350, June 2005.

[90] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, C. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[91] Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *Proc. 17th International Workshop on Languages and Compilers for Parallel Computing*, Sept. 2004.