UNIVERSITY OF CALIFORNIA, SAN DIEGO

Compile-time Optimization of a Scientific Library through Domain-Specific Source-to-Source Translation

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Alden King

Committee in charge:

Professor Scott B. Baden, Chair
Professor Ranjit Jhala
Professor Sorin Lerner
Professor Sutanu Sarkar
Professor Daniel Tartakovsky

2012

The dissertation of Alden King is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2012

DEDICATION

To myself, fifty years from now.

May this be the least of your accomplishments.

EPIGRAPH

The Jews were amazed and asked,
"How did this man get such learning without having studied?"
— The Bible, John 7:15, New International Version (1984)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thank you, Morgan Parker, for all your love, support, and tasty meals. I still don't know how you managed to read through my papers and thesis.

Thank you to all my friends at the University of California San Diego, the University of Washington, other friends I've met along the way, and those who have stuck with me since childhood. Their support and advice have been invaluable. Thanks for all the beer and scotch. Thank you to my family. All puns are dedicated to my father, Douglas King.

Thank you to my labmates, Pietro Cicotti, Didem Unat, Tan Nguyen Thanh Nhat, Mark Gahagan, Alex Breslow, Tatenda Chipeperekwa, Yajaira Gonzalez, and Mohammed Sourouri for all their help and support with reading and editing papers and working in the lab. Thank you Professor Allan Snavely for all your support and excitement, I am sorry you could not be here today. I would like to extend a special thank-you to Eric Arobone, who has been my collaborator and counterpart in the Mechanical and Aerospace Engineering department at UCSD. His insights into physical matters and, more importantly, his explanations of exactly what they mean to a laymen have been incredibly valueable and helpful.

And of course, a thank you to my advisor, Scott Baden, for his direction, guidance, and assistance through these five years. Thank you to my committee for all your advice and guidance, not just on this thesis, but on grad school in general.

Thank you to the fantastic beer bars in San Diego for deliciousness and keeping me sane, particularly my favorite local watering hole, Blind Lady Ale House. Thank you to all the coffee shops in San Diego, especially to Le Stats for being there 24/7 whenever inspiration struck, and especially when it didn't.

This research was supported in part by NSF contract OCE-0835839 and in part by the Advanced Scientific Computing Research Petascale Tools Program of the U.S. Department of Energy, Office of Science, contract No. DE-ER08-191010356-46564-95715. This research used Teragrid and other resources provided by the San Diego Supercomputer Center (SDSC), in particular, the Trestles and Triton machines. Computational resources housed in the CSE Department at UCSD were supported by the NSF DMS/MRI contract 0821816.

VITA

2007	B. S. in Computer Science, University of Washington, Seattle Washington
2007	B. A. in Philosophy with a minor in Mathematics, University of Washington, Seattle Washington
2006-2007	Undergraduate Teaching Assistant, University of Washington, Seattle Washington
2007	Graduate Teaching Assistant, University of California San Diego, La Jolla California
2010	Master in Computer Science, University of California San Diego, La Jolla California
2012	Ph. D. in Computer Science, University of California San Diego, La Jolla California

PUBLICATIONS

Alden King, Eric Arobone, Sutanu Sarkar, and Scott Baden; "The Saaz Framework for Turbulent Flow Queries", *7th IEEE International Conference on e-Science*, November 2011.

Alden King, and Scott Baden; "Reducing Library Overheads through Source-to-Source Translation", *International Conference on Computational Science*, June 2012.

ABSTRACT OF THE DISSERTATION

Compile-time Optimization of a Scientific Library through Domain-Specific Source-to-Source Translation

by

Alden King

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Scott B. Baden, Chair

In recent years, scientific exploration has become more reliant upon computers. Certain scientific frontiers, such as the study of fluid dynamics, are difficult and costly to study empirically, as measurement devices interfere with what they are measuring. By using computers to simulate known low-level physical interactions, scientists can reproduce higher-level phenomena than they can through physical experiments, and at a much lower cost. With the growth of high-performance computing, computer simulations are able to generate enormous amounts of data.

Analyzing large-scale scientific data is expensive both in terms of computational power and time, and in terms of programming effort. For some fields, such as Computational Fluid Dynamics (CFD), scientists are still searching for

mathematical models to explain observations. Such models are part of the scientific inquiry and so are continually under development. As a consequence, data analysis in these fields is largely *ad hoc*. Many different queries are asked of the data, queries which are not known ahead of time. Current libraries do a poor job optimizing the execution of such ad hoc queries.

In this thesis we introduce *Saaz*, a C++ library for analyzing turbulent flow. While Saaz makes analysis codes easier to write, maintain, and share, it significantly harms performance. Compared to *plain C++* (C++ with no user-defined abstractions), Saaz code can perform as poorly as 97 times slower. To address these issues we present *Tettnang*, a source-to-source translator which uses semantic knowledge of the Saaz library to track the data schemas being used. Saaz library calls are then re-written at the call-site to remove the abstraction overheads of the library. After being optimized by Tettnang, ad hoc queries written in Saaz perform comparably to the plain C++ implementation. Of the queries we tested, the slowest was 17% slower than the plain C++ implementation, while the fastest was 16% faster.

Our success with Tettnang demonstrates the power and effectiveness of custom translation in creating an Embedded Domain-Specific Language (EDSL) from an application library. The EDSL technique allows powerful transformations that are not available through existing techniques such as Expression Templates [Vel96] [VJ02], or existing library annotation systems such as Broadway [GL00].

Chapter 1

Introduction

In recent years, scientific exploration has become more reliant upon computers. Certain scientific frontiers, such as the study of quantum dynamics or fluids, are difficult and costly to study empirically, as measurement devices interfere with what they are measuring. By using computers to simulate known low-level physical interactions, scientists can reproduce higher-level phenomena. For example, quantum and atomic interactions, which are well-founded on quantum mechanics, can be used to simulate the actions of a nuclear reactor. By studying these higher-level phenomena under different conditions, scientists can develop higher-level models.

In computational science, high-level physical models are built through computational analysis of large amounts of data. This data can come either from empirical measurements (such as at the Large Hadron Collider), or from computer simulations. Analyzing large-scale scientific data is expensive both in terms of computational power and time, and in terms of programming effort. Even an inexpensive computation, performed trillions of times, can be expensive. Some analyses simply need to repeat a few different computations many, many times; given efficient implementations of each computation, these can be straight-forward to program. Other analyses require more programming resources. These include analyses that need to explore different computational operations and their effectiveness in describing the higher-level phenomena scientists seek. Because the operations being computed change over the course of analysis, we refer to them as *ad hoc queries*. Significant programming effort is spent maintaining analysis tools

that use ad hoc computations.

Large-scale data analysis requires complex analysis tools. A typical approach for managing software complexity is modularity, typically through the use of libraries. Libraries, however, are not traditionally well suited to support ad hoc, user-defined operations. Typically, efficient scientific libraries (e.g. BLAS [DCHH88b] or LAPACK [ABD+90]) provide highly-optimized implementations of a few operations. However, it is difficult to encapsulate an optimized implementation for a computation which has not yet been decided. The support of ad hoc queries prevents a library from optimizing computation, forcing it instead to focus on data-structures and data-access.

Computational Fluid Dynamics (CFD) is one area of computational science in which ad hoc queries are especially important. In this thesis we look at the study of turbulence within CFD. We have developed *Saaz*, a C++ scientific library for the analysis of turbulent flow data. Saaz supports ad hoc queries by providing an abstraction of the data schema used by turbulence simulators. The abstraction of data schema reduces the maintenance burden of turbulence analysis codes, letting a single analysis code be used with different simulators and simulation conditions. This encourages interoperability and the sharing of tools.

Modern languages offer objects as a means of encapsulating data and related operations within a single data-structure. Unfortunately, the object-oriented capabilities of languages such as C++ segment operations in a way that makes optimizations difficult for the compiler. Compared to language-primitive data-types, traditional compilers do a poor job generating efficient code for user-defined (or library-defined) data-structures. While a library providing convenient data-structures may succeed in reducing the cost of programming, it often does so at the cost of a decrease in performance. Compared to analysis written in *plain C++* (C++ with no user-defined abstractions), Saaz codes exhibit a significant performance penalty (9-97 times slowdown).

In our case, general-purpose compilers are unable to optimize code written against the abstractions of Saaz. In this thesis we categorize the overheads and performance penalties introduced by libraries in general, and Saaz in particular.

We introduce *Tettnang*, a source-to-source translator that addresses these overheads in the Saaz library. Similarly to how compilers like GNU's gcc are able to optimize calls to common library functions such as `memcpy` or `malloc`, Tettnang incorporates semantic knowledge of its input. Tettnang uses built-in knowledge of Saaz to identify and track data schema. Through a series of localized transformations, Tettnang uses this information about data schema to rewrite Saaz queries to use cheaper, lower-level primitives. The general-purpose compiler running on Tettnang's output is able to optimize these lower-level primitives to achieve performance comparable to that of *plain C++*.

Dissertation Contributions

- We have developed Saaz, a C++ library for analysis of CFD data. Saaz abstracts data schemas, simplifying queries and making them more interoperable across simulators and groups.

- We have identified three categories of library overheads: *Encapsulation*, *Isolation*, and *Generalization*. We give a small toy example of these overheads before demonstrating them in an example computation that utilizes the Saaz library.

- We have built the Tettnang translator to address the identified overheads in the Saaz library. By utilizing knowledge of the semantics of the Saaz library, Tettnang can gather information which had previously been isolated from general-purpose compilers. Tettnang eliminates Saaz overheads that stem from *Encapsulation*, *Isolation*, and *Generalization*.

- We have demonstrated Tettnang's effectiveness in greatly reducing the overheads of the Saaz library in real queries. Performance with Tettnang-processed code is at least 83% of the performance of implementations written in plain C++, and sometimes even exceeds plain C++ performance.

- We have developed a means of extending primitive data-types (particularly arrays) to support new operations and behavior with little to no performance

costs.

- Finally, we discuss future directions and extensions to Tettnang and the techniques it uses.

Dissertation Outline

Chapter 2 provides motivation and background for this thesis. It describes the physics of the turbulent flows analysis we perform with Saaz, as well as properties of the analysis.

Chapter 3 delves into Saaz itself. It begins with background on the types of analysis CFD performs and what a library for turbulent flow analysis will have to support. It continues with a description of the primitives in Saaz, before introducing example queries. Saaz supports ad hoc queries; abstractions of data schema increase the interoperability of queries written with Saaz.

Chapter 4 covers the overheads that exist in object-oriented libraries, classifying them as *Encapsulation*, *Isolation*, or *Generalization*. It finishes with some example techniques which have been developed to address some of these overheads in libraries at large.

Chapter 5 introduces Tettnang. Tettnang utilizes *type-refinement* to track the data schema of Saaz objects. Support for various schema contributes to various overheads introduced by Saaz. We describe the different modules within Tettnang and demonstrate their transformations.

Chapter 6 presents results achieved by Tettnang. We look at how Saaz queries without Tettnang's transformations suffer a significant slowdown compared to their implementation in plain C++. We show how the various transformations performed by Tettnang positively impact performance.

Chapter 7 discusses the limitations of Tettnang, future directions, and concludes the dissertation.

Chapter 2

Motivation and Background

Computational Fluid Dynamics (CFD) studies fluid dynamics using computer simulations. By using Direct Numerical Simulation (DNS), a CFD methodology, scientists can simulate fluid flows by solving the low-level Navier-Stokes equations. Computer simulations are able to take exact "measurements" of the entire flow field. Conversely, physical studies require inserting probes or sensors to measure fluid flows. Probes and sensors, however, are limited: they can take few measurements and interfere with the flows they are measuring. The completeness of the measurements available in computer simulations reduce the need for parameter tuning and eliminate the potential for bias in probe-placement. By analyzing the measurements taken from these flow fields, scientists can construct high-level models of fluid behaviors. High-level models describe not how small bodies of fluid move and interact, but how general trends influence large-scale behavior.

2.1 Questions in Computational Fluid Dynamics

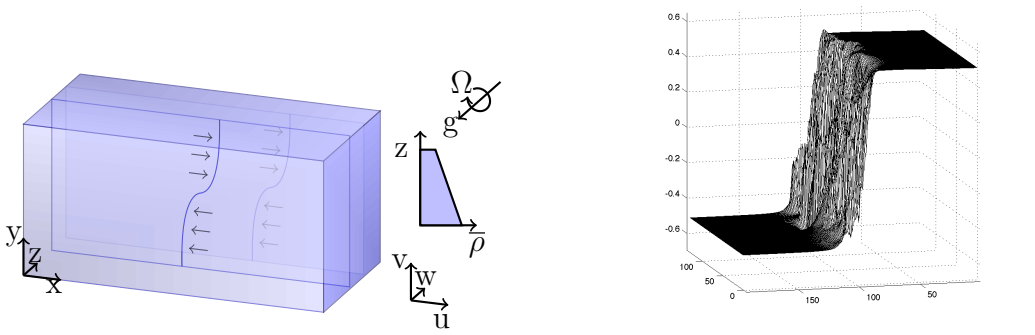
One interesting high-level fluid behavior is that of vortical structures. Such structures are ubiquitous in fluid flows, particularly in turbulent flows, such as in an internal combustion engine, or around a submarine. Formally defining turbulence is not trivial, but several features are universal in turbulent flows. Turbulence is dissipative, meaning that significant amounts of kinetic energy are converted continuously into internal energy. Therefore, turbulent flows tend to decay rather

rapidly when no energy source is present. Turbulent flows are also highly nonlinear, resulting in flow that appears random and is highly sensitive to initial conditions. Turbulence, however, is not truly random and has statistics that are highly reproducible when appropriate averaging (ensemble, spatial, or temporal) is implemented. Efficient mixing is another feature of turbulent flows. For example, efficient mixing of momentum leads to increased drag on solid bodies such as airplane wings. The adverse health effects of tail-pipe emissions on pedestrians are minimized through efficient dispersion of particulates. Turbulence is three-dimensional with velocity and vorticity fluctuations of comparable magnitude in all three directions [TL72].

Turbulence is affected by variations in density (e.g. *stratification*, where mean density varies in space). Stratification can modify vertical motion and mixing significantly. In the ocean, stratification results from gradients of temperature and salinity. The atmosphere is also a stratified system, where density variations result primarily from temperature and water-vapor gradients. When stratification is such that density is a function of elevation alone, with lighter fluid positioned above heavier fluid, then stratification acts to stabilize turbulence, in turn inhibiting vertical motion. If stratification differs from this base state then energy can be extracted from the density field, providing an energy source for turbulent mixing and internal wave emission.

Coherent vortical structures play an important roll in geophysical flows, such as those in the ocean. Oceanic flow with scales much larger than those at which energy is dissipated and much smaller than geostrophic scales are poorly understood. These scales are influenced by stable stratification and moderate effects of the Earth's rotation [RL08]. Ocean models require better mathematical descriptions of physical processes to become an accurate predictive tool.

It is our objective to provide better mathematical descriptions of physical processes, in particular, the roll *vortical structures* play in environmental mixing and transport. We are currently assisting the investigation of *barotropic instabilities*, which are a particular class of *shear instabilities* in stratified flow. These barotropic instabilities are known to form coherent vortical structures. To under-



(a): Configuration of the temporally evolving shear layer.

(b): Simulation data for u (the velocity in the x direction) in the yz plane.

Figure 2.1: In (a), two streams of fluid are shown, one moving in the positive x direction and the other in the negative x direction. The streams are separated by a y mid-plane. Stratification (mean density variation) is in the z direction, which is shown by the constant density gradient, $d\bar{\rho}/dz$.

stand these structures, we examine how they differ from the surrounding flow field. By comparing different physical quantities within and without these structures, we can better understand the part these structures play in the environment.

The turbulent flow we are helping to investigate (Figure 2.1) is that of a temporally evolving shear layer, representing two streams with velocity difference ΔU oriented horizontally, subjected to uniform vertical stratification. The streamwise (x) and vertical (z) directions are assumed to be infinite and homogeneous, and are thus appropriate directions over which to perform statistical averaging. We apply fluctuations of various sizes to the flow velocities in the region between the two streams to accelerate the transition to turbulence.

2.2 Simulations of Turbulent Flow

The approach we use to simulate turbulent flow is called Direct Numerical Simulation (DNS). DNS consists of solving the *Navier-Stokes* and *density* equations for incompressible flow (Figure 2.2). Simulations consist of an outer loop which marches through time, solving the equations each time step to update flow variables for the next. While DNS is not tractable for many engineering problems (due to the wide range of spatial and temporal scales that must be resolved) its lack of

$$\rho\left(\frac{\partial \vec{u}}{\partial t} + \vec{u}\cdot\nabla\vec{u}\right) = -\nabla p + \rho\vec{g} - 2\vec{\Omega}\times\vec{u} + \nu\nabla^2\vec{u} \qquad \frac{\partial \rho}{\partial t} + \vec{u}\cdot\nabla\rho = \kappa\nabla^2\rho$$

(a): The Navier-Stokes Equations. (b): The Density Equation.

Figure 2.2: The Navier-Stokes and Density equations. $\vec{u}$ is the vector-valued velocity, with components u, v, and w. ρ is the fluid density. p is pressure. $\vec{g}$ is the gravity vector. $\vec{\Omega}$ is rotation. κ is diffusivity.

empiricism has made it a successful research tool.

By simulating low-level physics, DNS eliminates the dependence on empirical models. Empirical models often have many degrees of freedom that must be resolved via tuning, and thus become intractable.

The Navier-Stokes equations are solved using a pseudospectral method with colocated fourth-order compact differencing computation of derivatives in the *inhomogeneous* direction and spectral collocation in the homogeneous directions. Conservation of mass is enforced using a projection method, which is solved using a parallel Thomas algorithm. The pressure solve reduces to a tridiagonal system of equations which is also solved using a parallel Thomas algorithm. Density advances in time via an advection-diffusion equation (Figure 2.2b). Time marching is accomplished using a low storage third order mixed Runge-Kutta Williamson scheme with viscous terms treated implicitly [Bew10].

Different simulations are required to examine and explore a particular model. Each simulation may have different *physical parameters*, where quantities such as the *Reynolds Number*, *compressibility*, *rotation*, or *stratification* vary. Varying these physical parameters may lead to the use of different *configuration parameters* in the simulation's configuration. Configuration parameters reflect implementation choices by the particular program that runs the simulation. This includes, for example, the mapping of physical coordinates to Cartesian indices of an array, and the layout of that array in memory.

2.3 Turbulent Flow Analysis

To develop new turbulence models, we must extract information about physical behavior from the data generated by our simulations. Unfortunately,

without a model to guide us, the way in which physical quantities relate to physical behavior is unknown. We must experiment not only with different simulations and configurations, but also with our analyses.

Analysis consists of executing a number of *queries* over data generated by a simulation. These queries compute different physical quantities. Ultimately, we analyze data from a number of different timesteps of a number of different simulations.

The normal scientific process guides our analysis: we develop hypotheses about (i.e. models of) observed behavior, use this to measure (by computing physical quantities), and then refine our hypothesis. Our model gives us an idea of which simulations and configurations might be interesting, letting us generate large amounts of data. Our model also tells us what kind of physical quantities in this data might be interesting. Once computed, these quantities help us refine our model. We then repeat the process.

2.4 Comp. Fluid Dynamics Analysis Software

The most common approach to analyzing CFD data is to use application-specific tools. These tools are often constructed purely for the specific case at hand with little care regarding their future use or maintainability. This state of affairs inhibits the dissemination and sharing of ideas and tools. While there are some commercial systems available, most are for well-established engineering applications and take short-cuts that are not physically accurate, making them inappropriate for scientific discovery. There is a need for software which can be customized to support both the specific case at hand as well as future cases as simulators and query choices change and new models are developed. Our Saaz software makes it easier to write interoperable, sharable code for CFD analysis.

Chapter 3

Saaz

In this chapter we will present Saaz. Saaz is an array-class library for C++ designed for post-processing *Computational Fluid Dynamics* (CFD) data, that is, the data generated by CFD simulations. We will present a description of design considerations for Saaz and the problems it aims to tackle. The primitives in Saaz will be discussed briefly before presenting examples of Saaz code. More details regarding the implementation of Saaz can be found in Appendix A, with a more complete set of examples in Appendix C. Extensions to Saaz appear in Appendix B.

3.1 Target Problem

Saaz is designed to aid in the analysis of large datasets generated from CFD simulations. Particularly, the data-discovery codes which assist in the development of new physical models. Saaz presents an abstraction of arrays. While the computations we address are from CFD, Saaz includes no physical concepts directly. Arrays are indexed at Cartesian coordinates, but there is no notion of distance outside of the index space. All physical context and restrictions (such as the Reynold's number) are imposed upon computations by the programmer, not by Saaz. This should make Saaz useful for domains outside of CFD, but we do not address that use in this thesis.

3.1.1 Maintenance

One of the difficulties in CFD analysis is the dependence between simulation and analysis. Analysis codes are tailored to simulators, which are in turn tailored to the specific fluid conditions being explored. It is difficult to avoid the coupling of problem and simulator because of the wide variety of conditions and flows that could be explored; tailoring a simulator to the subject at hand makes sense.

The necessity of coupling simulator and analysis tool, however, is less clear. Developing new physical models requires exploring the efficacy of different analyses. The analyses can change dramatically over the use of one simulator, and are often similar if not identical to the analyses performed over different simulators. If we can decouple the simulator and analysis tool, we can reduce the code-maintenance burden on the scientist.

Maintaining an analysis tool requires attention to several issues as conditions change:

1. Physical and problem-specific assumptions change. If the data changes physical characteristics, code which relies on some of those characteristics may need to be rewritten.

2. Performance Robustness (hardware portability with performance). If the underlying hardware is changed and has different performance trade-offs, then a tool may need to be rewritten to avoid slowdown and take advantage of the changes.

3. Storage schema changes. A new simulator may have a different native format for storing and managing data. Tools which are unable to accommodate a different way of organizing data must be recoded for each new organization.

4. Analysis itself changes. When exploring what analyses are informative, some queries or computations will naturally become less interesting while others become more interesting. Tools which are flexible in accommodating changes to data analysis algorithms will require fewer changes.

The ability to maintain an analysis tool across these changes will be of great help to the CFD community [Moi10]. Most physical assumptions are simple enough to be factored out into variables, and so recoding is trivial or nonexistent. Hardware changes usually affect optimizations such as cache blocking which require tuning to optimize. There has been much work already in this area, and so Saaz does not address it. Presently, most analysis tools are tightly coupled to a simulator via its storage schema. New analyses are added and removed ad hoc as needed: reimplemented for each tool and its simulator's specific storage schema. By decoupling a storage schema and analysis, Saaz reduces the need to reimplement these tools.

Physical Assumptions Physical assumptions might include the Reynolds number, the mapping between Cartesian coordinates and the index space (coordinate mapping), compressibility vs. incompressibility in the simulation, the effects of stratification, the effects of planetary rotation, or even which physical values (velocity, density, pressure, etc.) are simulated. Generally these assumptions are frequently able to be parameterized into a few variables. Most queries that rely on these assumptions only rely on them to the extent that they require them as inputs. The structure of a query is not usually dependent upon these assumptions. For example, the mapping of a coordinate system can parameterize a query by specifying which axes are inhomogeneous and which axes a loop covers, but it will not normally affect how many loops are used.

Performance Robustness Changes in hardware can greatly affect the performance (though generally not correctness) of scientific applications. Such changes include changes in the memory hierarchy (cache size, main memory size, use of flash, etc.), memory bandwidth, and computational power relative to memory accesses. Increased data sizes, whether for higher physical or temporal resolution, often necessitate an upgrade in hardware. Code-generators such as Spiral [PMJ+05] or FFTW [FJ05] make it easy to port an algorithm implementation to a new hardware system. These systems rely on an understanding of the sensitivity of the computation to certain hardware parameters. If, for example, cache sizes change, a loop nest can be blocked differently to avoid false sharing and take advantage of

the cache.

Unfortunately, the effect of hardware changes on arbitrary, ad hoc, computations is still an open area of research [OTCS09] [OSMC10] [WSO+09]. It is difficult to find the optimal tuning parameters for specific hardware changes. Saaz does not attempt to tackle this problem, instead it addresses the remaining two rewrite conditions.

Storage Schema Analysis tools must be compatible with the storage schema used by the simulators with which they interact. Factors such as the physical properties being explored, or the techniques to do so, may affect the optimal way for a simulator to format data in memory. In some cases a particular axis ordering or data layout (such as row-major) may be preferable. Just like a simulator may have an ideal data arrangement, so too may an analysis tool. We would prefer to have this ideal data arrangement depend on the operations being performed, and not on the way the tool is written. By abstracting the organization of data away from the analysis tool, we can elide these details and make it easier to write more interoperable code.

When a significant amount of computation must be performed over simulator-generated data, it may be cost-effective to reorganize it (such as with a transpose) before analysis. These transformations, however, may be expensive both in terms of time and memory usage. It may not be reasonable, or even necessary to reorganize the data. The primitives in Saaz are designed to make it simple to express computations in such a way as to be robust to changes in data organization. In particular, computations using a Saaz array need not know if the array is stored in memory using a row-major or a column-major layout.

Saaz arrays include metadata that is maintained when the array is stored on disk. A single array type can hold arrays of differing layouts, and will choose the appropriate one based on how each instance is created. Forcing users to maintain metadata themselves (e.g. array layout) can be distracting. When metadata is incorporated into a query's implementation, it can become confusing and a source of errors, especially as code is maintained, updated, or ported. Abstracting how data is organized isolates metadata from computations. This lets a program be

written without being concerned with data layout, allowing code written with Saaz to run over different datasets. Thus, Saaz programs are robust to changes in storage schema.

Analyses Many existing tools and libraries provide fixed sets of primitive computations. These tools are effective because they restrict operations to a few specific instances which are highly tuned. This can yield high performance, but it also restricts flexibility.

In circumstances where analyses are changing more frequently than the library itself, the library will be unable to adapt to new analyses directly. Analyses are expressed as code, not as data, and are therefore harder for a library to manipulate. When a new query needs to be computed, code must be rewritten and recompiled. In the scientific realm, particularly the exploratory parts of CFD, analyses change quickly as new venues of possible causes and correlations are explored. This makes the use of a few specific, but highly-tuned implementations ineffective.

Instead of providing a set of fixed queries (possibly parameterized by data), Saaz makes it easy to express new queries. We refer to these queries as *ad hoc* because they are not known prior to analysis and change as scientists refine their models. An interface which is natural to domain scientists is important in this regard. Domain scientists don't want to talk to computer scientists every time they want to compute something new, especially if they do not do so already. Many domain scientists make use of Fortran and Matlab for these sorts of new computations. The Saaz interface uses similar concepts. Queries are expressed procedurally via imperative computations.

3.1.2 Kinds of Abstractions

In terms of abstractions, libraries and languages are not significantly different. Both can have abstractions which encourage or discourage compatibility with different systems. The primary difference is that libraries are embedded in an existing language. By operating within an existing language, libraries enjoy a greater

opportunity for compatibility with both old and new systems. This encourages adoption.

A library may sometimes be referred to as an Embedded Domain-Specific Language (EDSL). In this case, either the language's compiler or another compiler is able to treat operations on a library as language primitives. This affords opportunities for optimization of the library which may not be available to a more general-purpose compiler. Certain functions in the C standard library, for example, are treated as built-ins by some compilers (such as `memcopy` by GNU's gcc). The semantics of these functions are known by the compiler and calls to them may be rewritten using an optimized, more-efficient implementation.

Software libraries offer a convenient way of managing program complexity as data analysis becomes increasingly complicated and expensive. Libraries offer the programmer modularity: a few experts are able to provide an optimized implementation for many different users. What abstractions a library offers can vary significantly depending on the library's purpose. Some uses are amenable to declarative abstractions, others merely abstractions of algorithms or specific operations, and still others permit only the slimmest of data abstractions. Figure 3.1 shows different levels of abstraction.

Declarative Ideally all libraries would be able to operate at the highest levels of abstraction, offering a declarative interface. At the declarative level, users provide a high-level schema for data organization. Users may then merely declare what they want but not how to get it (e.g. SQL SELECT). At this level, the library can automatically arrange data in a way optimized for the retrieval patterns and computations users require. Operations presented to the user have certain high-level mathematical properties of which the library is aware (e.g. identity values, composability, associativity, etc.). By incorporating rules about these properties, the library can pick the best implementation of a specific query (computation) over data as well as the best way to organize data for that query. Libraries presenting declarative-level abstractions can thus achieve very high performance.

SQL databases have been successful because the relational model for organizing data works very well for a large class of problems. Not all problems are

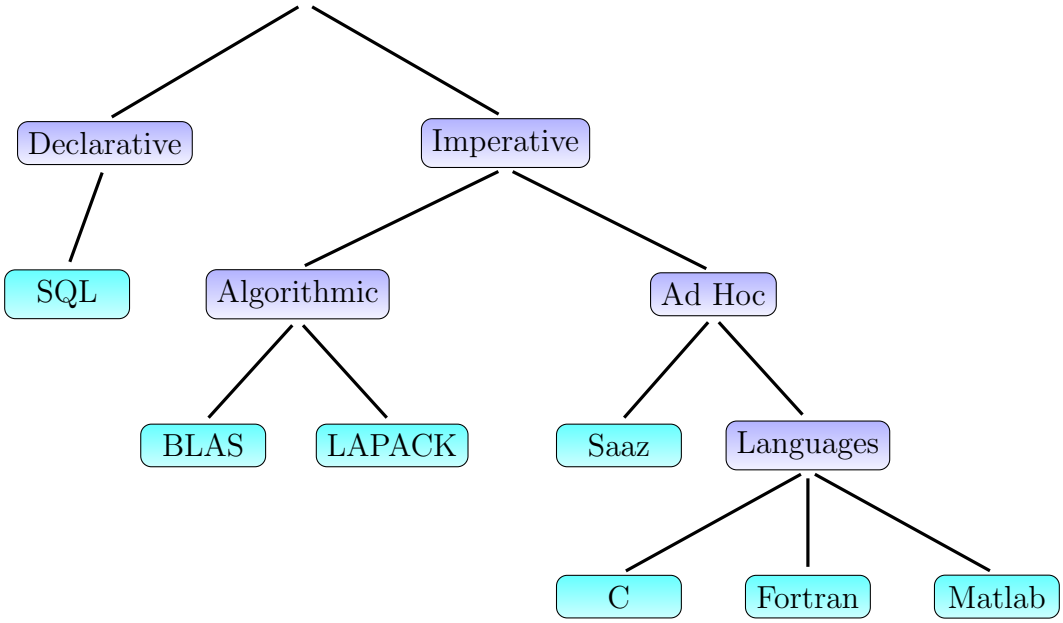
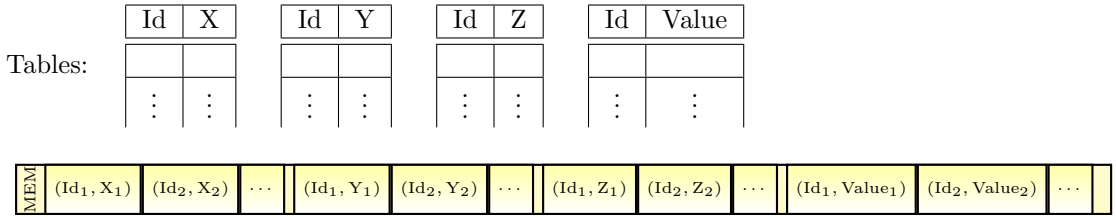


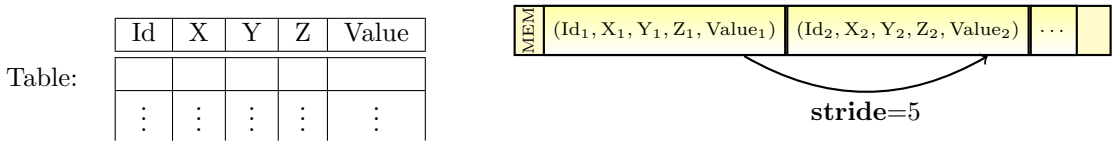
Figure 3.1: Differing levels of abstraction. Internal nodes represent an abstraction category. Leaf nodes are examples of libraries or languages within that category.

amenable to declarative abstractions, however. Compatibility with other systems or storage constraints may require a lower-level schema than that which a declarative interface uses, preventing it from re-arranging data. Furthermore, even if data can be reorganized, sometimes the computations to be performed over the data are not expensive enough to justify the cost of reorganizing the data. The most effective way to organize data tends to be very problem-specific.

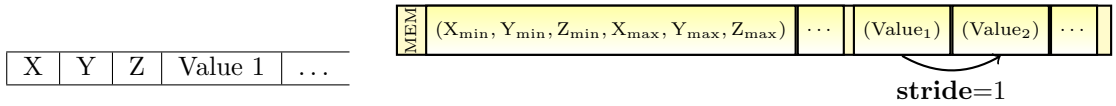
The Sloan Digital Sky Server [GST+02] is one of the few projects in the scientific community which has successfully and effectively used a SQL database to analyze data. The Relational Model [Cod70] used by SQL is not well suited to handling intensive computations involving arrays as it cannot preserve memory locality in tightly-bound loop nests. The direct, bi-directional mapping between an array's indices and memory location preserve locality much better. Figure 3.2 illustrates storage of an array inside a relational database and a regular imperative language. One way of storing an array relationally is shown in Figure 3.2a: all the values may be stored adjacent in memory. Unfortunately, the mapping from the indices is far away, and so accessing adjacent values requires non-local access. Alternatively, Figure 3.2b organizes all the values in adjacent tuples in memory.



(a): One way of storing array data in a relational model. Separate tables hold coordinate information and values. Each table is stored as a list of rows.



(b): Another way of storing array data in a relational model. A single table holds coordinate information and data.



(c): Regular arrays hold bounds information, then all values adjacent in memory.

Figure 3.2: Different ways of storing data can cause poor locality in the relational model as in (a) and (b), compared to the imperative model used by (c).

Using this method, the index records in between values are not needed for computation, merely for retrieving the values, and they take up space in the cache. Using non-relational storage of raw array data (Figure 3.2c), values are stored next to each other, and can often be retrieved from the cache. Because of CFD's heavy reliance upon large arrays of data, relational databases have failed to gain general acceptance in the CFD community [LPW+08].

Imperative In cases where operations and data cannot be constrained to a formal logic, a declarative approach may be inappropriate. By allowing the user to specify *how* to perform computations, as well as which computations to perform, more general problems and considerations may be addressed. Unfortunately, this leaves fewer opportunities for the system to optimize executions and data retrievals.

Algorithmic An important application design pattern is to glue together a fixed set of high-level operations. In these cases, abstractions at the algorithmic level may be ideal. By devoting efforts to a few specific operations, a few efficient implementations can significantly increase performance of a large number of programs.

The scientific realm is especially sympathetic to this approach. In scientific analyses, operations largely appear as a consequence of mathematical properties or explanations, and as such do not specify an implementation. Which implementation is optimal will depend heavily on data properties and organization. For example, matrix-matrix multiply is a high-level operation which can be implemented in many different ways. Considerations affecting performance include: what kind of data (sparse vs. dense), the layout of data (row-major vs. column-major), and the properties of the data (e.g. triangular or diagonal matrices). Libraries such as BLAS [DCHH88b] and LAPACK [ABD+90] provide many different implementations for certain algorithms. By specifying how their data is organized and particular data properties, users of these libraries can benefit from tuned implementations. Because there are relatively few ways that data can be organized, it is feasible to tailor each of the important high-level algorithms to the data.

Ad Hoc In domains that explore the efficacy of different models or computations (such as CFD), however, it is not possible to provide an efficient implementation of a few high-level algorithms. Construction of a new physical model requires constantly re-examining and changing what is being computed. A high-level primitive which performs a specific canned operation is not useful here. Rather, exploration requires a continuous ad hoc re-construction of operations. As a consequence, only the simplest of primitives for accessing data, and not operating on data, are used. The subjects of optimization are limited to data access, not algorithms.[1]

In CFD, for example, it is not known what physical properties (such as vorticity) can best define a physical phenomena (such as a vortex). Different physical properties must be computed and their efficacy for elucidation of dynamics

[1] In some cases, an implementation of a known algorithm may be identified from uses of primitives, and may then be optimized according to the semantics of the known algorithm (such as replacing it with a more efficient implementation). Identifying these known algorithms is difficult, fragile, and not the subject of this thesis.

evaluated. These properties must be evaluated and deemed effective over different experimental conditions, such as the *Reynolds Number*, *compressibility*, *rotation*, or *stratification* conditions. As new experiments are performed, the problem's data configurations often change. Low-level details relating to data organization may not be relevant to the properties being explored, but they affect the implementation of a computation. Using a library to abstract these details about data organization lets the user focus on the exploration of the data properties.

Because Saaz is designed to help develop novel physical models, it cannot manage the implementation of specific algorithms. Instead, it supports the expression of ad hoc computations and abstracts the organization of data.

Interestingly, during the course of developing Saaz, we found it useful to introduce an algorithmic library on top of it. Saaz only abstracts data, thereby allowing ad hoc queries. This abstraction, however, is too low-level for some uses who perform only a few common analysis operations. These users prefer a smaller API and built-in operations. There are a few operations common to several queries which can be implemented in Saaz and packaged. To that end, we developed the *Cascade* front-end to Saaz. Cascade simplifies the implementation of particular queries compared to regular Saaz. Many queries, however, cannot be efficiently expressed in Cascade. Dissipation, for example, cannot be easily computed with Cascade's primitives. For these more complicated queries, and for queries which do not fit Cascade's pre-defined operations, Saaz itself must be used. Cascade is discussed in more depth in Appendix B.2.

3.1.3 Data Organization

Saaz abstracts data schemas for regular grids. In CFD, simulation data is usually organized in one of three different ways:

1. Regular Grids
2. Irregular Meshes
3. Adaptively Refined Meshes (AMR)

Regular Grids operate on a dense multidimensional array of data where each coordinate in space is represented by a specific index point in a domain. While the physical space is not necessarily uniform, the grid over it uses a domain with a uniform index space. Data is arranged regularly in memory using a closed (reversible) formula. Regular Grids also allow the use of implicit indices: the location in memory is a function of an index's value. Regular grids, however use a single resolution, and thus do not allow very much flexibility in how precise a space is handled.

Irregular Meshes cover a particular volume or surface of interest such as an airplane wing. Their irregularity allows them to be very fine-grained at small scales such as edges or where behavior of great interest is expected. Irregular meshes are usually stored as a list of triangles, and thus do not preserve locality as well as dense arrays.

While iterating over an irregular mesh can be just as convenient as iterating over a regular grid, it is much more difficult to access values at specific locations. Even when triangles are organized hierarchically, operating on values which are adjacent in physical space is expensive because they are not adjacent in memory and must be found. Certain operations require accessing values by physical coordinates. Planar averages, for example, require accessing all the values in a plane. Differential operators require accessing values from adjacent physical coordinates. Depending on the particular analysis needs and the tools available, an irregular mesh may be less appropriate than a regular grid. Care must be taken with the conversion, since if the conversion is too coarse accuracy may suffer, but if it is too fine, it will require much more storage space (and thus compute time).

Adaptive Mesh Refinement (AMR) [BC89] tries to offer the best of both regular grids and irregular meshes. AMR uses a set of hierarchically-ordered grids at varying resolutions. Each grid is represented densely. In this way, storage requirements are reduced while ensuring important or small components can be examined at high resolution, and large components can use more coarse approximations. Data can still be accessed by physical coordinate, although some interpolation may be required.

Saaz supports only regular grids. Although constructing an adaptive mesh would be possible, it is not natively supported, and would require extensions to domains and arrays, though the Saaz computational interface would not be altered. In cases where data is stored in an irregular mesh, it would have to be rearranged as one or more regular grids before Saaz can use it. However, because Saaz arrays maintain their own opaque mapping between physical coordinates and memory location, Saaz is free to implement different storage types. In the future, adaptive meshes may be implemented natively behind an array's indexing operation.

Saaz hides details specific to a simulation's data schema. For example, arrays in Saaz are indexed in such a way that their memory layout is transparent, that is, the mapping from a point in index space to a memory location is opaque. The mapping from physical space to index space is handled in a preamble, freeing the rest of the code from being aware of its specifics. Other configuration details are also able to be moved into the preamble. By using these kinds of abstractions, Saaz allows the user to express ever-changing computations while also making it easy to change simulations and schemas. It becomes easy to compute different physical quantities with code that is more interoperable across different physical experiments, parameters, and tests. Saaz's abstractions facilitate the construction of new physical models.

3.2 Saaz Primitives

As scientists continually refine their physical models they add, remove, or otherwise change the queries they use for analysis. This precludes a solution which focusses on a few high-performing computations.[2] To support the *ad hoc* nature of scientific exploration, Saaz simplifies the authoring of new queries instead of canned operations. Saaz abstracts data schemas and uses an imperative model for computational queries. This type of programming is familiar to domain scientists.

In this section we will present the primitive types in Saaz, as well as a

[2] There is a higher-level interface to Saaz called Cascade, which is discussed in more detail in Appendix B.2. Cascade abstracts some high-level operations which are common in CFD, but because it is not ad hoc, it is very inflexible, and not for general use.

```
1 Domain3 dmn(x_lo,y_lo,z_lo, x_hi,y_hi,z_hi);
2 Domain1 pencil = dmn.Collapse(Y_AXIS);
```

Figure 3.3: The `Collapse` operation for a domain selects a subspace according to the axes specified. If `Y_AXIS` is 1, `pencil` will cover the points $(y\_lo : y\_hi)$.

few operations. These types are: *Points*, *Domains*, and *Arrays*. The operations are: *Iterators* for traversing domains, *Predicates* for restricting computations, and *Aggregators* to summarize data.

3.2.1 Points

Point objects in this thesis will generally be named `p` or `q`. The point's value along a specific axis will be denoted `p.`*`axis`*, e.g. `p.x`.

Points represent an element of $\mathbb{Z}^n$ and are used to index n-dimensional arrays. They are elements of n-dimensional *domains* (Section 3.2.2, below). The use of points allows the user to write dimension-independent array-accesses: instead of listing a coordinate along each dimension, a single object contains each coordinate.

3.2.2 Domains

Domain objects in this thesis will generally be named `domain` or `dmn`.

Domains in Saaz represent a rectilinear subset of $\mathbb{Z}^n$. Domains are bound by two n-dimensional points *lo* and *hi* (Figure 3.5a). They contain all values within the Cartesian-product of n integer ranges: $(x\_lo : x\_hi) \times ... \times (z\_lo : z\_hi)$. We call the bounds in a particular dimension the *extents*. Axes are numbered starting from zero.

A lower-dimensional subspace of a domain may be extracted by using the `Collapse` method, which returns a hyper-plane (Figure 3.3). For example, a three-dimensional domain over $(x\_lo : x\_hi) \times (y\_lo : y\_hi) \times (z\_lo : z\_hi)$ can be collapsed to the y axis, to isolate two domains: one over $(y\_lo : y\_hi)$ and one over $(x\_lo : x\_hi) \times (z\_lo : z\_hi)$.

The extents of a domain `dmn` can be accessed via the `Min` and `Max` functions: `dmn.Min(0)` returns the first coordinate of the smallest point in the domain. For

some computations, such as those that work with neighbors, an interior domain is necessary. This is created using the `Inset` method. For example, for `dmn` over $(x\_lo : x\_hi) \times (y\_lo : y\_hi) \times (z\_lo : z\_hi)$, `dmn.Inset(1)` will create a domain over $(x\_lo + 1 : x\_hi - 1) \times (y\_lo + 1 : y\_hi - 1) \times (z\_lo + 1 : z\_hi - 1)$.

3.2.3 Arrays

Array objects in this thesis will generally be named A, B, or C, when referring to an array in general; arrays in the context of CFD include the three different components of velocity: `u`, `v`, and `w`.

An array is a mapping from the points of a domain to a value: $\mathbb{Z}^n \rightarrow \mathbb{R}$. That is, they hold a value corresponding to each point in a particular domain. Each array has a domain as an index set. Arrays may only be defined over dense domains. The domain of an array A can be retrieved via the `Domain` member call: A`.Domain()`.

We refer to arrays over a single dimension as *vectors*. Scalars (such as a single integer) can be thought of as arrays over zero dimensions.

Arrays are indexed with multidimensional points rather than tuples (i.e. A`[p]` instead of A`[i,j,k]`). This capability was pioneered in the Fidil programming language [HC88] and later employed in the KeLP system [Fin98] [FBK98] and the Titanium programming language [YSP+98]. Unlike many languages and libraries, which treat array-layout (whether an array is row-major or column-major) as a convention, Saaz allows the user to specify the layout as a property of each array instance. When a language chooses a particular layout, it simplifies the computations which must be done to index an array, but sacrifices compatibility. Saaz abstracts the organization of data from the indexing operation, making such code robust to changes in data schema.

3.2.4 Iterators

Iterators in this thesis are generally named `i`, `j`, or `k`, depending on which axis (or axes) they are covering (such as `ik` for the x and z axes). `i` will also

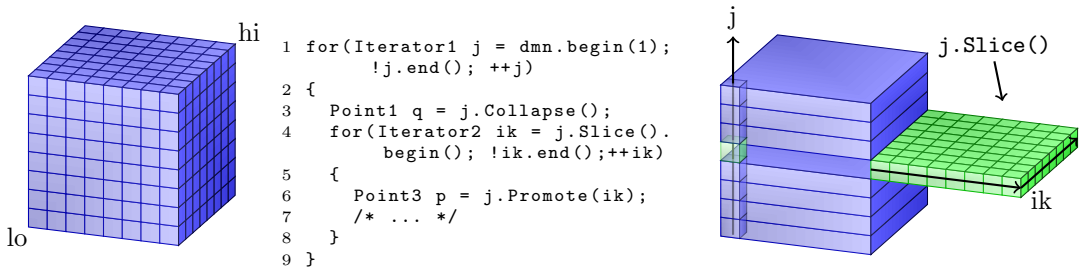
```
1 for (Iterator i = A.Domain().begin(); !i.end(); ++i)
2 {
3   A[i] = B[i];
4 }
```

Figure 3.4: Copying an array.

sometimes be used for an iterator over an unspecified number of dimensions.

To access arrays which are defined over a domain, we iterate over that domain. Iterator objects perform this task, taking on the value of all points within a domain. Figure 3.4 shows how to copy array B into array A. Because iterators are multidimensional and cover points along several axes, they allow a user to write a single for-loop that traverses a multidimensional index space instead of writing individual for-loops over each dimension. This allows the user to write dimension-independent code. The order in which iterators cover points is not set, although the order in which dimensions are covered can be forced. When not forcing a particular order, Saaz may choose a particular order to improve some aspect of performance such as locality.

Iterators can also cover subspaces of a domain. For example, an iterator can traverse a line existing in three-space by traveling along the leading point of planes in a three-dimensional domain. In this way, iterator `j` can cover just the y axis of a three-dimensional domain, `dmn` over $(x\_lo : x\_hi) \times (y\_lo : y\_hi) \times (z\_lo : z\_hi)$. In this case, the iterator will traverse the points between (x_lo, y_lo, z_lo) to (x_lo, y_hi, z_lo). To obtain the one-dimensional point, the iterator must first be *collapsed*. `j.Collapse()` will yield one-dimensional points between (y_lo) and (y_hi). When an iterator does not cover all the axes of a domain, it provides access to the remaining points of the domain using the `Slice` method. `j.Slice()` will provide the part of `dmn` which exists perpendicular to the axis traversed by the iterator `j`. In this case that is $(x\_lo : x\_hi) \times (z\_lo : z\_hi)$. See Figure 3.5. The corresponding 3-dimensional point, `p`, in the original domain can be obtained by *promoting* the iterators from the subdomains: `j.Promote(ik)`. In the case where the iteration space is sparse (due to *Predication*, Section 3.2.5, below), `j.Slice()` might change for each value of `j`.



(a): The Domain `dmn`.

(b): Traversing `dmn` with two iterators, `j`, and `ik`.

(c): The spaces covered by `j` and `ik`.

Figure 3.5: Two-loop iteration.

3.2.5 Predicates

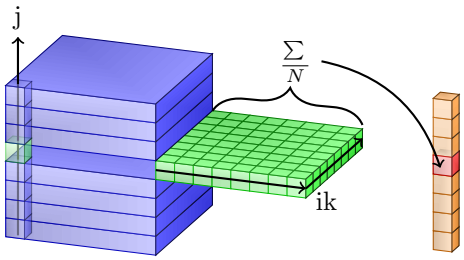
A predicate in Saaz constrains execution to a particular subset of a domain for arbitrary conditions. Predicate objects must be able to specify for each point whether or not execution should occur for that point. Because Saaz does not presently support sparse arrays, not executing the loop body for some points will result in dense arrays having default values. Predicates allow Saaz to use user-defined criteria to constrain computation. CFD researchers can deduce properties of coherent structures by isolating them from the rest of the flow.

Saaz does not support sparse domains directly. Instead, predicates are used to specify sparse *iteration spaces* and affect only the iterator. Saaz does not currently support sparse arrays.

3.2.6 Aggregators

Aggregators are built on top of the other Saaz primitives. Aggregators are objects which perform a generalized reduction operation, taking arrays of n dimensions and producing arrays of m dimensions, where $0 \leq m < n$. The reductions supported by Saaz are different from those of other systems such as Matlab which reduce arrays by aggregating along a single axis. Figure 3.6 illustrates a common reduction operation which reduces along two axes: the *planar average*.

Aggregator objects execute reductions by implementing six functions, each corresponding to different parts of a loop nest. These functions are: `Initialize`, `PreProcess`, `Process`, `PostProcess`, `Finalize`, and `Return`. Figure 3.7 shows the



(a)

```
1  for (Iterator1 j = dmn.begin(YAXIS);
       !j.end(); ++j)
2  {
3    Point1 q = j.Collapse();
4    avg[q] = 0;
5    for (Iterator2 ik = j.Slice().begin();
         !ik.end(); ++ik)
6    {
7      Point3 p = j.Promote(ik);
8      avg[q] += arr[p];
9    }
10   avg[q] /= j.Slice().Size();
11 }
```

(b)

Figure 3.6: A reduction to compute the average value in a plane with N=`j.Slice().Size()` elements.

```
1  DomainM collapsed = dom.Collapse(agg->GetOutputDimensions());
2  agg->Initialize(collapsed);
3  for (IteratorM i = dom.begin(agg->GetOutputDimensions()); !i.end(); ++i)
4  {
5    PointM q = i.Collapse();
6    agg->PreProcess(q);
7    for (IteratorL j = i.Slice().begin(); !j.end(); ++j)
8    {
9      PointL p = i.Promote(j);
10     agg->Process(q, p);
11   }
12   agg->PostProcess(q);
13 }
14 agg->Finalize();
15 return agg->Return();
```

Figure 3.7: Aggregator objects implement six functions which are called at various points of a loop nest. In this example, the aggregator object, `agg` *reduces* the N-dimensional domain, `dom`, to M dimensions, where $0 \leq M < N$ and $L = N - M$. `GetOutputDimensions` returns a list of the dimensions along which the aggregator should reduce, that is, the M dimensions which the outer-loop should cover.

call-sites of these functions within a loop nest. The order in which an aggregator covers points of a domain is not defined, so ordering assumptions should not be present in these function bodies. This allows Saaz the ability to select an order for the iterations for performance or other reasons. Saaz may even parallelize the computation.

3.3 Examples

Saaz factors out configuration parameters (i.e. the data schema) so they do not liter the expression of a query. By avoiding problem-specific assumptions and configuration details, Saaz makes query implementations more interoperable across

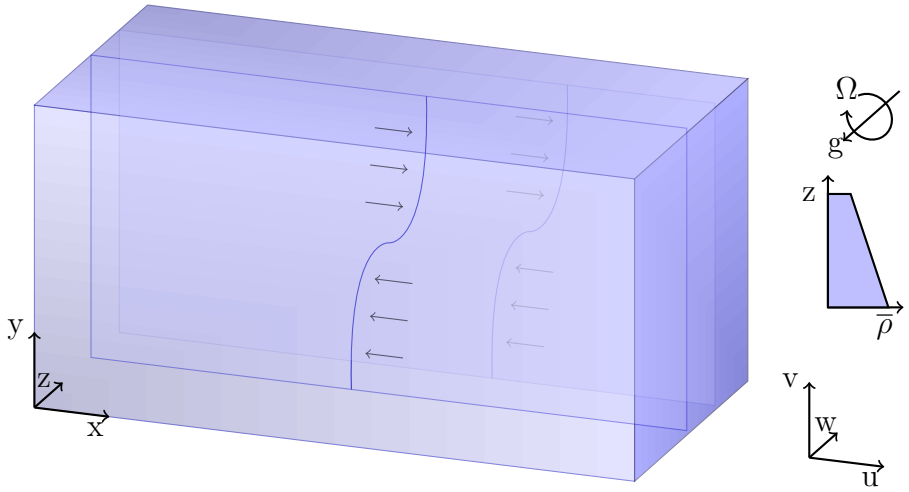


Figure 3.8: A shear layer. Fluid flows in the positive and the negative x directions, with an interface at the y midplane. u, v, and w are the components of the velocity along the x, y, and z axes, respectively. ρ (density) decreases as z increases. Gravity ($\vec{g}$) acts against the z axis, while rotation (Ω) acts clockwise around the z axis.

simulators. Configurations specify how a physical problem maps onto program data structures. Certain individual configuration details can be used to parameterize code. By moving configuration details out of queries, the queries can be used by different groups over data that comes from different simulators.

To introduce Saaz's interface, we will present a number of representative queries. We choose these queries because they are physically meaningful, and demonstrate a number of useful properties of the Saaz library. The particular problem being explored is the turbulent and coherent structures which arise in shear layers. Here, two bodies of water are streaming past each other at different relative velocities. At the interface, these different relative velocities introduce rotating flow and thus coherent structures form. This process is illustrated in Figure 3.8.

3.3.1 Query Syntax

Our datasets consist of uniform arrays of field variables as generated by a turbulent-flow simulator. We look at five primitive variables, although others can be added. u, v, and w represent the x, y, and z components of vector-valued velocity, respectively. ρ represents density, and p represents pressure. From these

primitive variables we derive other quantities by executing user-defined queries.

The most basic query is the *planar average*. The planar average performs the appropriate averaging to reduce the seeming randomness of turbulent flows. For 3D data, the planar average returns a 1D array containing the average of all the points of the corresponding planes in the 3D input array. The planes are perpendicular to the *inhomogeneous* axis (usually y, see Figure 2.1 and Section 2.1). In this thesis we do not use any axis other than y and an abuse of notation will mean the y axis when one is not mentioned. The planar average of array u along the y axis is represented by $\langle u \rangle_y$.

A *normalized* version of the primitive variables is required by many queries. Normalization measures deviations from the average by subtracting from each element, the average value of the points in the plane perpendicular to the inhomogeneous axis. The normalized value of array u is represented by u'. The value at point p is calculated by $u'[p] \leftarrow u[p] - \langle u \rangle_y [p.y]$ for an inhomogeneous direction along the y axis.

For those queries which involve derivatives, we compute derivatives using finite-difference approximations implemented as second-order stencil operations in physical space. Derivatives are represented using the standard $\frac{\partial}{\partial x}(u)$ or $\frac{\partial u}{\partial x}$.

Finally (for now), some queries involve correlations. A *correlation* is the planar-average of a point-wise product, for example: $\langle uv \rangle_y$. We use adjacency to indicate point-wise multiplication. uv means $u[p] * v[p]$ for all points p in the domain.

The remainder of this section will give example queries involving planar averages, correlations, and derivative correlations.

3.3.2 Planar Average

The planar average query (see Figure 3.6) forms the basis of a number of other queries. Because the planar average can be implemented with a simple Saaz reduction (perhaps by a Saaz aggregator), it is useful in demonstrating the common control-flow structure of reductions.

Certain problems exhibit characteristics of uniform physics. For example,

the surface parallel to and above an infinite plane should be uniform. These idealized problems simplify considerations to allow exploration of specific effects. In our fluid dynamic problems, there are two dimensions which exhibit uniformity for all flow variables: the streamwise direction (x), and the vertical (z). The y direction is inhomogeneous because it crosses the boundary between the two flowing fluids. For these problems we normalize queries with respect to the mean in these two homogeneous directions. This lets us see how the query changes with respect to the inhomogeneous direction.

The planar average is used in queries that require normalized field variables. In mathematical notation (not in code), normalized arrays are denoted with a tick: $\mathtt{v}'$, and defined such that:

$$\mathtt{v'[p.y] = v[p] - v\_avg[p.y]}$$

where `v_avg` is the planar average of `v`.

Figure 3.9 shows how the planar average would be implemented in plain C++ and in Saaz, respectively. In the main body of this thesis we have taken some liberties with typenames to ease clarity (see Appendix C for additional code examples). By comparing these two code examples, we can see how the abstractions in Saaz come into play.

The plain C++ implementation (Figure 3.9a) is littered with problem-specific assumptions and configuration parameters/data-schema details. To make matters worse, this implementation has certain *implicit* assumptions which limit performance, interoperability, or both.

1. The coordinates x, y, z are mapped to dimensions $0, 1, 2$ of the arrays (Lines a.7, a.8, a.11, a.13, a.15-17, and a.22).

2. The inhomogeneous dimension (the dimension along which the planar average is taken) is the y dimension (Lines a.7, a.8, a.11, a.13, and a.22)

3. Array bounds can have an arbitrary origin, as in Fortran (Lines a.7, a.8, a.10, a.11, a.13, a.15-17, a.18, a.21, and a.22).

4. The arrays store data in row-major order (Lines a.15-17).

```
1  double* PlanarAverage(double* arr,
2      int x_min, int x_max, int y_min,
3      int y_max, int z_min, int z_max)
4  {
5    int i,j,k;
6    unsigned long long idx;
7    double* avg = new double[y_max-y_min+1];
8    for (j = y_min; j < y_max + 1; ++j)
9    {
10     avg[j-y_min] = 0;
11     for (i = x_min; i < x_max + 1; ++i)
12     {
13       for (k = z_min; k < z_max + 1; ++k)
14       {
15         idx = (k-z_min) + ((j-y_min) +
16               (i-x_min) * (y_max-y_min+1))
17                         * (z_max-z_min+1);
18         avg[j-y_min] += arr[idx];
19       }
20     }
21     avg[j-y_min] /=
22           (z_max-z_min+1)*(x_max-x_min+1);
23   }
24   return avg;
25 }
```

(a): Plain C++.

```
1  Array1 PlanarAverage(Array3 arr,
2                       unsigned int Y_AXIS)
3  {
4    Domain3 dmn = arr.Domain();
5    Array1 avg(dmn.Collapse(Y_AXIS));
6    for (Iterator1 j = dmn.begin(Y_AXIS);
7         !j.end(); ++j)
8    {
9      Point1 q = j.Collapse();
10     avg[q] = 0;
11     for (Iterator2 ik = j.Slice().begin();
12          !ik.end(); ++ik)
13     {
14       Point3 p = j.Promote(ik);
15       avg[q] += arr[p];
16     }
17     avg[q] /= j.Slice().Size();
18   }
19   return avg;
20 }
```

(b): Saaz.

Figure 3.9: Planar Average implementations. In the plain C++ case, three-dimensional arrays are stored as single-dimensional C-style arrays because we wish to accommodate column-major layouts without changing data-types, and multidimensional C-style arrays are row-major.

5. The highest performance for the inner loop is attained when iterating in the z dimension, and then the x dimension (Lines a.13 and a.11).

The loops that processes the data are nested to reflect the axis labeling convention, and to improve re-use in cache. These implicit assumptions inhibit interoperability. This is problematic when queries need to be run under a different data schema.

The highest cost of different schemas comes not from the increased duration of queries (though, this is definitely a cost), rather, it comes from the need to rewrite analysis codes. It is this maintenance burden caused by the integration of problem-specific assumptions and data schema parameters throughout queries that is the real cost. For example, if our data source takes a different convention for coordinate axes (say $x = 2$, $y = 0$, and $z = 1$), the loops will have to be rewritten, the allocation size changed, the array linearization arithmetic changed, and the size calculations changed. If we don't do this, we would have to perform an expensive coordinate transformation prior to calling the routine, reorganizing the data in memory to ensure the planar average was taken across the correct axis and to avoid unfavorable memory access strides.

Re-implementing queries for different configurations is time-consuming and prone to error. Saaz's goal is to abstract away built-in assumptions about a data schema so that queries can be interoperable across different schemas and domain scientists can remain focused on the science of their application. In other words, Saaz does not remove these assumptions, but rather removes the responsibility from the user's consideration (Figure 3.9b). Instead of playing a tacit part in the implementation the user must write, these assumptions are hidden in parameters or in the data objects. This makes the salient features of the query, the mathematical operations, more clear.

1. The coordinate mapping from x, y, z to axes $0, 1, 2$ is specified using either the axis name or a basis point (basis vector) (Lines b.5 and b.6).

2. The inhomogeneous dimension is factored out as the `Y_AXIS` variable (Lines b.5 and b.6).

3. The bounds are an attribute of an array's domain object, which can report its size or the length along a given axis (Line b.17).

4. Layout is an array attribute. As a consequence, the arrays in a query may have identical layouts, but may also have differing layouts (Line b.15).

5. The domain object, rather than the programmer, determines the order taken by the inner loop iteration (Line b.11).

This code is interoperable across different array layouts as well as different coordinate mappings. Except for the coordinate mapping from x, y, z to axes $0, 1, 2$ and the choice of inhomogeneous dimension, all the other configuration details are encoded in the way the array, arr, is constructed. When we load a Saaz array from disk, all the configuration details (i.e. the schema) come with it, and are not set via user code. This makes Saaz code much simpler to write and much more interoperable across simulators and groups, and even versions of simulators, thus facilitating the sharing of code and data.

3.3.3 Point-wise Correlations

Correlations are the continuous sum (i.e. definite integral) of point-wise products of n-dimensional fields. In our case, this integration is represented by the planar average of the point-wise product of two arrays. Point-wise correlations are calculated by, for every point, multiplying the corresponding values of the two arrays. In our 3-dimensional cases, the array which holds this product is reduced via a planar average to a 1-dimensional array.[3]

Note that we need not compute the product and then reduce it, but can do both together. The *vw*-Correlation, for example, can be computed by modifying the Planar Average query. We can replace Line 15 from Figure 3.9b with

```
14 avg[q] = vn[p] * wn[p]
```

`vn` and `wn` are the normalized values of the `v` and `w` arrays, respectively. Again, `vn` is not necessarily instantiated, but could be computed inline as `v[p] - v_avg[q]`. One of the interesting characteristics of the correlation queries is that they either reuse an array's value (auto-correlations) or they access values for the same point in two different arrays (cross-correlations). This can have effects on caching and computational overheads.

3.3.4 Derivative Correlations

Derivative correlations are correlations that involve a derivative as one of their components. In Saaz, these are implemented as finite difference calculations via stencil operations. In our examples we use a second-order approximation ($O(h^2)$). See Figure 3.10.

A good example of a query which makes use of derivative correlations is dissipation. Dissipation measures the loss of energy, and is thus important for understanding turbulent systems [TL72]. Regions with more dissipation tend to have stronger turbulence. Dissipation has three sub-queries, each of which are

[3] In Matlab notation, for example, for 3-dimensional arrays A and B defined over a domain D, the correlation along axis 1 will produce a 1D array (vector), C. $size(C) == size(D, 1)$, and $\forall\ i \in D(1)\ .\ C(i) = mean(A(:, i, :) * B(:, i, :))$. Where $A(:, i, :) * B(:, i, :)$ is the point-wise product of the i planes of A and B.

```
1 Array1 YZDissipation(Array3 v, Array3 w,
2     Array1 v_avg, Array1 w_avg,
3     unsigned int Y_AXIS, Point3 PY, Point3 PZ,
4     double dz, double dy)
5 {
6   Domain3 dom = v.Domain().Inset(1);
7   Array1 yzdiss(v.Domain().Collapse(Y_AXIS));
8   double tmp1,tmp2,cur;
9   for (Iterator1 j = dom.begin(Y_AXIS); !j.end(); ++j)
10  {
11    Point1 q = j.Collapse();
12    yzdiss[q] = 0;
13    for (Iterator2 ik = j.Slice().begin(); !ik.end() ++ik)
14    {
15      Point3 p = j.Promote(ik);
16      tmp1 = (((v[p + PZ] - v_avg[q]) -
17               (v[p - PZ] - v_avg[q])) / dz);
18      tmp2 = (((w[p + PY] - w_avg[q+1]) -
19               (w[p - PY] - w_avg[q-1])) / dy);
20      cur  = tmp1*tmp1 + tmp2*tmp2;
21      cur /= 4.0;
22      yzdiss[q] += cur;
23    }
24    yzdiss[q] /= j.Slice().Size();
25  }
26  return yzdiss;
27 }
```

Figure 3.10: A Saaz implementation of the yz-Dissipation query. `Inset` selects the interior of a domain, with padding of the specified width (Line 6). `Collapse` of a domain selects a subdomain along a domain's axis (Line 7). `Collapse` of an iterator demotes it to the 1-dimensional point on the line it is traversing (Line 11). `Slice` selects the domain perpendicular to an iterator which traverses another subdomain (Line 13). `Promote` combines 1-dimensional and 2-dimensional points to produce a 3-dimensional point (Line 15).

interesting in their own right. Figure 3.10 shows how yz-Dissipation (named for the velocity components which it uses, `v` and `w`) is implemented in Saaz.[4]

The yz-Dissipation query differs from the vw-Correlation primarily in its use of adjacent points. This requires that it use the interior domain: it pads the iteration space to stay within bounds. The points $\vec{Z}$ and $\vec{Y}$ are unit-vectors, added or subtracted according to vector-addition and vector-subtraction. At each iteration, we do not access values at point p, but rather values at $p \pm \vec{Z}$ and $p \pm \vec{Y}$ (as well as points in the one-dimensional averages at q and $q \pm 1$). Which dimensions these are is factored out of the actual query through the parameters `Y_AXIS`, `PZ`, and `PY`. Thus, $p + \vec{Z}$ (`p + PZ`) for $p = (p_1, p_2, p_3)$ may be $(p_1, p_2, p_3 + 1)$ for $\vec{Z} = (0, 0, 1)$, or it could be $(p_1 + 1, p_2, p_3)$ for $\vec{Z} = (1, 0, 0)$. This allows the query to be valid for schemas that differ in their coordinate mappings.

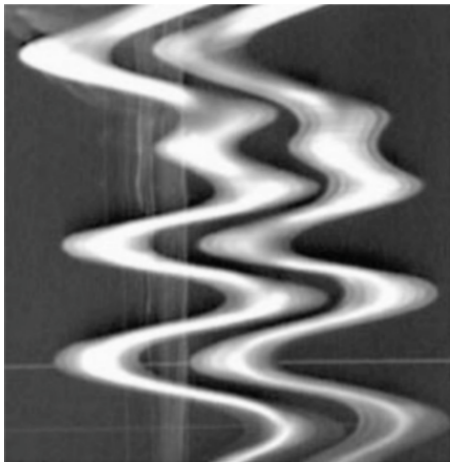
[4]This is actually a component of pseudo-dissipation, which is a common approximation of the full dissipation.

This code is compact and intuitive. We can readily see which data-points are being accessed and what computation is being performed on them. In the case of a plain C++ implementation, the code is much more complicated. Saaz supports arithmetic on point-valued objects, so expressing offsets from a given point is straight-forward. In plain C++, however, the arithmetic must be included. As before, we do not use multidimensional C-arrays because we want to be able to support column-major layouts. The arithmetic to linearize points can make the code much harder to read, especially with complicated expressions involving many different point locations.

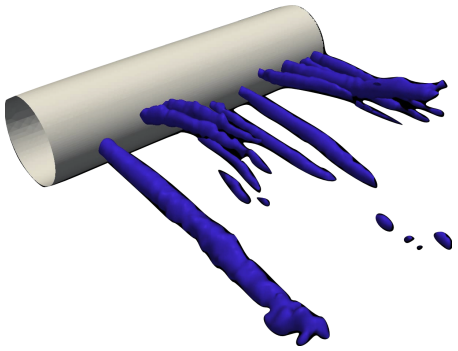
One approach which is conventionally used to mitigate this clutter is to define macros (or inline functions) which perform the linearization operation. $p+\vec{Z}$ can then be written as `OFFSET(i,j,k+1)` (for $p = (i, j, k)$). The use of macros in this fashion will introduce many redundant computations between linearization operations. It can be expected that the compiler will be able to eliminate these in a plain C++ implementation. This approach, while it may hide the linearization computations, does not hide the mapping of coordinates to axes. If $\vec{Z}$ were to change from $(0, 0, 1)$ to $(1, 0, 0)$, then all of the macro arguments would need to be updated.

3.3.5 Eduction Criteria

When running physical experiments, scientists use an inert dye so that the flow can be visualized. As the flow moves the dye around, visual inspection can identify vortical structures. Figure 3.11a shows this for a zig-zag instability. Computational simulations can simulate dye, although keeping track of it can take non-trivial amounts (20%) of additional space in memory. Unfortunately, it is often not clear where the dye should be placed to begin with. The experiment or simulation may have to be re-run several times to find good locations. When examining a different region or property of the flow, the dye needs to be moved to a new location, possibly over several attempts. Instead of looking at a flow that includes some dye, simulations can use an *eduction criterion* to computationally define the regions of interest.



(a): A zig-zag instability [BC00]. This flow has lots of dissipation but still has coherent structures.



(b): λ_2 for a flow over a cylinder [HJ11].

Figure 3.11: Interesting components of fluid flow.

Eduction criteria are functions which attempt to quantify the location of rather qualitative phenomena. In our case, we focus on "vortices". Several criteria have been used, including Q [HWM88], Δ [CPC90], *Lagrangian Coherent Structures* (LCS) [HY00], and λ_2*(Lambda 2)* [JH95]. LCS is the most expensive, and provides more detail than is necessary to identify vortical structures. λ_2 identifies vortical structures well, but is still expensive, involving not just many floating-point operations (141), but also several trigonometric operations and square-roots (8). The visualization of these eduction criteria is useful in visualizing vortices. Instead of watching the dye evolve, vortices can be identified immediately. For example, Figure 3.11b shows λ_2 resulting from the flow over a cylinder. Figure 3.12 shows λ_2 resulting from flow over the landing gear of an airplane.. Because the eduction criteria is computationally defined it can also be used to isolate computations. By thresholding one of these criteria, we can conditionally compute other queries. This lets scientists examine how vortical regions are different from the rest of the flow. This approach will help build a new model of flows dominated by coherent vortices.

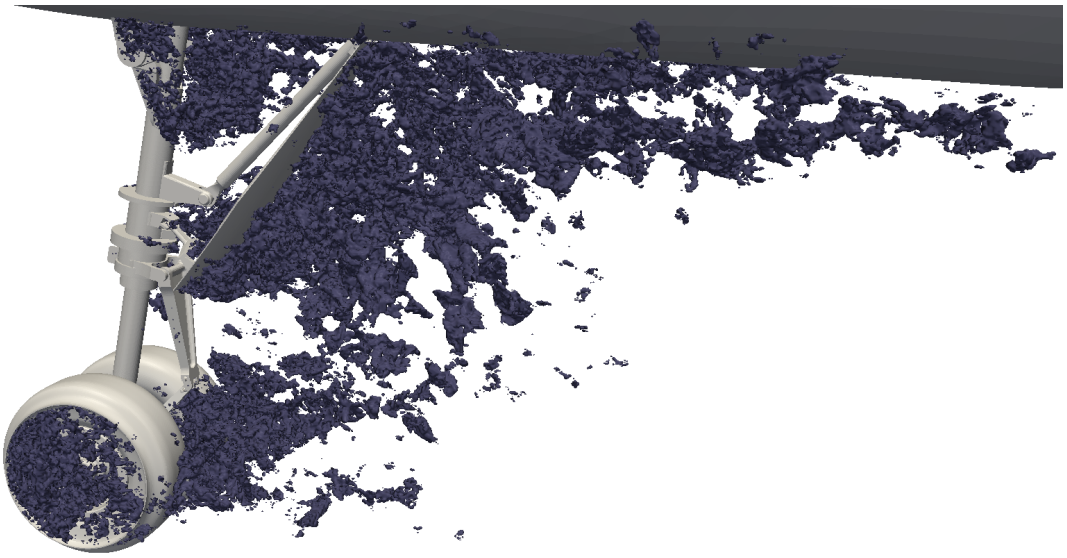


Figure 3.12: λ_2 computed around the landing gear of a Gulfstream airplane [dAJH12].

Chapter 4

Library Overheads

As discussed in Section 3.1.2, there are different levels at which languages and libraries build abstractions (reproduced in Figure 4.1). One is the abstraction of algorithms (the *Algorithmic* level of abstraction). Libraries of this sort provide highly tuned implementations of particular algorithms, such as matrix-matrix multiply. Such libraries may provide these implementations via a number of different functions and leave it up to the user to select the optimal one. Alternatively, they may provide a single function that, perhaps automatically or with some user guidance, is able to select the optimal implementation. For problems that perform only a few known computations and when these computations are expensive relative to data access, algorithmic libraries work well.

Sometimes, however, a set of fixed algorithms is inappropriate. When users need support for a variety of computations, and these computations are not known a priori, there are not yet algorithms for which an optimized implementation can be developed. Libraries supporting unknown, *ad hoc* computations provide abstractions for something other than computation, frequently data-organization.

Languages provide some primitive data-structures, but most applications interact with data in a more complicated way. One particularly common data-structure is the *array*. Arrays are collections of data, all of the same type. Different languages provide different abstractions of arrays. Fortran, for example, supports multidimensional arrays, while C and C++ support only single-dimensional arrays (they are just contiguous memory regions; multidimensional arrays in C and C++

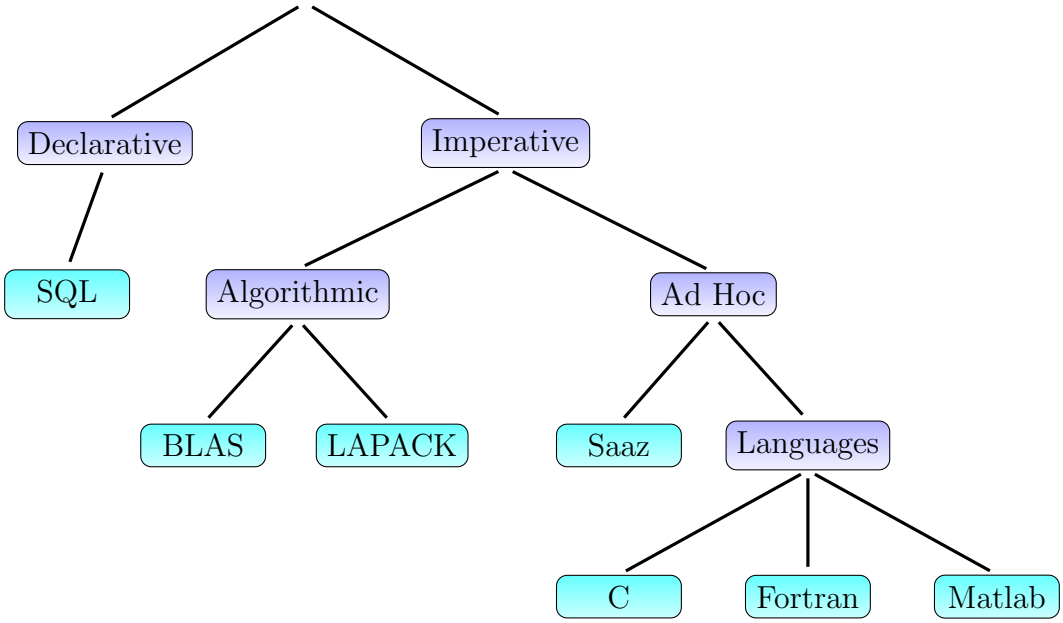


Figure 4.1: Differing levels of abstraction. Internal nodes represent an abstraction category. Leaf nodes are examples of libraries or languages within that category.

are a kludge).

Languages vary in what kinds of dynamic properties they allow for arrays. Fortran and Matlab support multidimensional arrays that can dynamically change their extents (size and shape). C arrays can dynamically change their size through `realloc`, although doing so is fragile and often ineffective. The Saaz library is an array-class library for C++. Its primary abstraction is the multidimensional array. Saaz provides the ability not for arrays to change their extents, but to have different layouts: to be either row-major or column-major, a capability not present in Fortran, Matlab, or C/C++.

The choice of abstractions is largely determined by the application requirements. SQL for example, targets business data which conforms to the relational data-model. The properties of the relational calculus provide many opportunities to reorganize queries and data. As a result, SQL can provide a high-level abstraction of both data and computation. Although SQL is a language, it is made available to conventional languages through library support. SQL-bridge libraries achieve the same performance benefits as the SQLlanguage directly because they merely forward requests to the same SQL execution engine. Recent work has added

support for SQL to existing languages [MEW08] [TTS+08].

The kinds of and depth of abstractions which a language or library provides can significantly affect its performance. SQL is able to deliver very strong performance because it operates within a constrained computational model that is amenable to optimizing performance by reorganizing data and computations. SQL queries are declarative, and thus do not specify a particular order of operations. In fact, few SQL queries are executed without being rearranged. Because SQL execution engines understand all of the semantics of their high-level language, they are able to provide a very efficient implementation.

Imperative languages, on the other hand, are more general-purpose. While the compiler may know a language's semantics, the language's generality may prohibit heavy optimizations. Maintaining support for general computations requires forgoing the ability to perform certain optimizations. *Algorithmic* libraries address certain computations by providing highly-optimized implementations. General-purpose languages are not so lucky. While compilers can do some analysis to fit some computations into certain patterns, this usually requires very simple or no user-defined data-structures. Datasets with dynamic attributes generally require data-structures and methods of data-access that obfuscate the computations. The myriad fine-grained functions which control or modify these data-structures are easily confused with functions that perform the computations that are sine qua non for the program. It becomes difficult for the compiler to disambiguate which code is working to retrieve data and which is doing meaningful computations. The complexity of library interfaces in modern object-oriented languages makes it difficult, if not impossible, for compilers to optimize general computations which make heavy use of a library. Compilers operate at the level of a language, not a library. One solution is to enable compilers to reason about libraries and computations which use them, in effect, turning the library into an *Embeded Domain-Specific Language*.

When general-purpose compilers are discussed we will be referencing Intel's icc and ifort (version 12.0) and GNU's g++ and gfortran compilers (version 4.7). We use the term *general-purpose* to mean compilers that are designed for a

particular imperative language, but without specific knowledge of a library which is implemented in that language. *Domain-specific* compiler, on the other hand, refers to a compiler which has been granted special knowledge about a particular domain (of information), and so is able to draw on more information than just a language's specification. In the case of a library with a tuned domain-specific compiler, the pair may be called an *Embedded Domain-Specific Language* (EDSL) [Hud96] [MHS05]. Our source-to-source translator, Tettnang, which we introduce in Chapter 5 is a domain-specific compiler incorporating knowledge of the Saaz library. Tettnang treats Saaz arrays as a primitive data-type in a domain-specific language embedded in C++.

4.1 Overhead Categories

Before presenting solutions to library overhead, we first want to understand the kind of overheads that a library can introduce: overheads are from more than just function calls. We first present these overheads in the context of a toy program, and then the Saaz library. More generally, these overheads are typical of object-oriented libraries. For example, TooN [Ros] is an object-oriented numerical library that uses LAPACK and BLAS as backends and also suffers from these overheads. Analysis tools at the Large Hadron Collider [FJHL08] also exhibit the kinds of overheads which we document in this chapter.

In order for domain scientists to develop scientific models, they must be able to explore the efficacy of different physical quantities/statistics. These statistics are not known *a priori*, and as a consequence, old ones are discarded and new ones are attempted continually. To support these ad hoc queries, Saaz's interface abstracts only data-structures, but not algorithms. In plain C++, the code is littered with details about the data schema such as the coordinate system and the in-memory (or on-disk) layout of arrays. Saaz allows the programmer to factor out the data schema as configuration settings, rendering the expressions of computational queries much more interoperable than plain C++.

Configuration details are factored out through the use of class layers. These

class layers introduce three categories of overhead which move the performance of Saaz far away from that of plain C++. *Encapsulation* of code and data introduces duplicate metadata and indirection overhead. *Isolation* of concerns restricts the compiler's ability to perform optimizations (e.g. vectorization, common subexpression elimination). *Generalizations* introduce superfluous computations and control flow changes.

The remainder of this section will describe these categories of overhead in more detail. Figure 4.2 provides an example program, which deals with people living in houses in a typical object-oriented way. We will use this example to illustrate the overhead categories and how general-purpose compilers respond to them. In this example, `Person` objects are associated with `House` objects. Multiple people may point to the same `House`, or no house at all. Handling these circumstances introduces overheads that fall within our categories.

4.1.1 Encapsulation

Encapsulation manages complexity by hiding data in objects and structures and hiding code in functions and methods. It is considered good practice to not make object data directly accessible, but rather to use accessor methods (e.g. Line 20). These methods can check invariants and arguments, or log accesses for debugging purposes. They also add function-call overhead, as well as overhead from consistency checks such as `assert` statements. These methods are typically short and usually something a compiler can inline. Even when the use of an accessor method can be avoided, there are overheads from accessing object datamembers directly. An extra addition operation is required to generate the memory offset from the object pointer.[1]

We refer to data which is needed to control computations or behavior, but is not at the essence of the program, as *metadata*. Metadata in Saaz, for example,

[1] Although the exact consequences are architecture or compiler dependent, our tests with Intel and GNU compilers on an x86-64 processor revealed that both compilers incurred the additional add overhead when dealing with objects on the heap. With later versions of `g++`, the results were less consistent. There are times in which `g++-4.7` is able to avoid this overhead by storing a read-only datamember in a register.

```
#include <iostream>
#include <string>
class House
{
public:
  House (const std::string& number_, const std::string& street_)
    : number(number_), street(street_)
  { }
  const std::string GetAddress() const
  { return number + " " + street; }
private:
  std::string number, street;
};
class Person
{
public:
  Person(const std::string& first_, const std::string& last_, House* home_ = NULL)
    : first(first_), last(last_), home(home_)
  { }
  const std::string& GetFirst() const
  { return first; }
  const House* GetHome() const
  { return home; }
  const std::string GetAddress() const
  {
    if (home)
      return home->GetAddress();
    else
      return "Unknown Address";
  }
private:
  std::string first, last;
  House* home;
};
int main()
{
  House lakefront("221 B", "Baker Street");
  Person lyra("Lyra", "Silvertongue", &lakefront);
  Person will("William", "Parry", &lakefront);

  std::cout << lyra.GetFirst() << " lives at " << lyra.GetAddress() << std::endl;
  std::cout << will.GetFirst() << " lives at " << will.GetAddress() << std::endl;
};
```

Figure 4.2: An example of many library overheads.

controls how arrays are formatted in memory or on disk, but is separate from the actual data in the array. Encapsulating code into objects frequently requires duplicating metadata or accessing it with another level of indirection. Metadata must be duplicated when each object instance needs access to descriptive state independent of other instances. When separate instances store metadata in a common object, they access it through indirection. Furthermore, there may be duplication of work when multiple objects use the shared object for the same task.

Example

The program in Figure 4.2 creates two people, `lyra` and `will`, and gives them a `lakefront` house to live in. The `GetFirst` accessor method retrieves the first

name for a person, and incurs overhead in the function call and the retrieval of the name. `GetHome` incurs similar overhead. Each person maintains duplicate metadata by storing `home` in a datamember. This duplication is application-specific, but the library does not predict how it will be used, and each object stores what ends up being the same pointer. This is typical overhead associated with data encapsulation.

Encapsulation of code has additional overhead. The `GetAddress` method computes the concatenation of the `number` and `street` components of the house. Because each person lives at the same house, they end up having the same address, but because the `GetAddress` method is specific to each person, this address has to be computed twice.

Compiler Response

When common code gets moved into functions or methods, function call overhead is introduced. Most modern general-purpose compilers are able to inline such operations (`GetFirst` and `GetHome`). The Intel and GNU C++ compilers, for instance, are able to perform this optimization under a wide variety of conditions.

4.1.2 Isolation

One of the primary goals of encapsulation is separating different subsystems. This separation frees the programmer from being concerned with the details of implementation, but also thwarts compiler analysis. Saaz, for example, hides from the programmer array-layout and thus how data is retrieved. While the programmer can access this information, he does not need to know it to author queries.

The features which separate a user from implementation details can also prevent the general-purpose compiler from performing optimizations. Encapsulation of operations into objects requires repeated operations and their requisite metadata to be moved into object methods and members (e.g. `lakefront` is moved into `Person::home`). Object methods are isolated, operating only on their particular object instance (e.g. `Person::GetAddress`). When multiple objects share the

same metadata, calling certain methods on each object can be redundant.

When multiple objects require identical metadata, the compiler cannot reason about their common values, and thus cannot perform optimizations such as common sub-expression elimination. To do so would require something like Global Value Numbering (GVN) [AWZ88], which tracks many of the values and equivalences in the program. While some modern compilers such as GNU's `g++` include GVN, applying it to datamembers requires it to work across object constructors and methods. Tests indicate `g++`'s reasoning to be insufficient when dealing with datamembers. When indexing a Saaz array, the array's bounds are used to linearize the point into a memory offset. The array's bounds are specified as parameters to the array constructor. Multiple arrays with the same bounds individually store these in their respective datamembers (or store a pointer to it in different datamembers). Identifying that two arrays have identical bounds and thus, when indexed, share subexpressions requires significant compiler analysis. Because of the required inter-procedural analysis, general-purpose compilers are unable in the general case (or common case) to make the determination that the arrays have identical bounds. In the case of plain C++, arrays are not self-describing, and so the variables which hold their bounds are usually shared among arrays, and thus it is easier for a general-purpose compiler to identify common subexpressions.

Example

In our example, both `lyra` and `will` live at the same house. While it is Encapsulation overhead which causes their `GetAddress` methods to duplicate work, it is the Isolation of `GetAddress` that prevents the compiler from optimizing it. In order to identify that `lyra.GetAddress()` and `will.GetAddress()` return the same value, the compiler must know that both `lyra` and `will` store the same object in their `home` datamember. This requires tracking that value through the object's constructor and into its datamember. This sort of alias analysis can be expensive. In this case, both Intel and GNU compilers failed to use this analysis to save a call.

Compiler Response

Inlining can expose some computations within object methods to be similar (e.g. the null-check for the `home` datamember on Line 26, or the concatenation of strings in `GetAddress` on Line 10). However, since each object contains its own copy of the metadata (or its own pointer to the metadata), the computations will appear to be over different data. It takes significant alias and pointer analysis to identify that these computations are over the same data and do indeed share common subexpressions. This sort of analysis is not solvable in the general case, and general-purpose compilers cannot know a priori in which situations it is beneficial, let alone possible. As a consequence, general-purpose compilers elect to not perform this analysis. In our example, persons hold metadata in their `house` datamember. The Intel and GNU compilers do not recognize that `lyra.house` holds the same value as `will.house`, and thus cannot see that the two calls to `GetAddress` yield the same result. These compilers generate code that duplicates the work. If a compiler had domain-specific knowledge, it could know in which situations common subexpressions would arise, and not need to perform the analysis.

Indeed, it is this ability to know a priori where certain optimization opportunities lie that allows domain-specific compilers to overcome Isolation overheads. By incorporating information about how a library works (its semantics), a compiler can avoid having to follow use chains throughout the program. By avoiding this expensive (and often untenable) analysis, domain-specific compilers can better track object metadata and identify which computations are being performed and if they are redundant.

4.1.3 Generalization

Generalizations support code reusability and interoperability, thus facilitating sharing of software and tools. When designing a library, the designers strive to enable it to accommodate problems similar to, yet different from, the ones available at design time. Plain C++ implementations will typically specialize code to the particular circumstances of the problem at hand, while libraries must function for other problems where these implicit assumptions do not hold.

Few programs use all the features of the libraries they invoke. These features are not useless, they are merely superfluous in some particular instances. Nonetheless, they can have costs. Overheads occur in the form of new control flow to decide between features or generalized arithmetic operations (such as multiplying by a value that happens to be one or adding what happens to be zero).

Example

Our example supports two possibilities which we do not exercise. First, persons are allowed to live in different houses, yet both `lyra` and `will` live at the `lakefront` house. Second, persons are allowed to live at no house. If people live at no house, then the `GetAddress` method returns "`Unknown Address`". In this program, this branch is not taken, but to identify this, the compiler needs to know the value of the `home` datamember, or at least that it is non-null.

Compiler Response

Overheads from Generality can be especially hard for general-purpose compilers to identify. Identifying them usually requires non-local knowledge with which general-purpose compilers have trouble. Operations for a particular circumstance may not be strictly necessary for a given use of the library, but the determining factors can be data-dependent, preventing compilers from reasoning about them. General-purpose compilers use branch prediction and code profiling to gain some application-specific knowledge about which code-paths a program is using. This can allow code to be implemented to support general operations but with a lower cost. Domain-specific compilers do not need runtime support to get this information. By incorporating knowledge of how code-paths are related to each other and to other parameters, domain-specific compilers can identify the important code-paths and optimize for them.

Oftentimes, however, instead of implementing a general solution with unused code-paths or operations, the user will specialize their code to the particular problem at hand. This is a major contributing factor to the need to constantly recode analysis tools. Problem sizes, for example, may be constrained to a power

of two. When these constraints need to be changed, entire modules or programs will need to be rewritten.

Software systems such as operating systems use the C-macro preprocessor for conditional compilation (`#if`). This practice can be effective, allowing code for different circumstances to be interleaved with general-purpose code. Conditional-compilation, however, can be messy and hinder the readability of code. Furthermore, it requires recompilation, often with significant configuration changes, thus restricting the flexibility of programs. If the compilation conditions are not independent, a single program can have a combinatoric explosion of compilation paths, complicating testing and code readability.

4.2 CFD Examples

We have presented three categories of overhead: *Encapsulation*, *Isolation*, and *Generalization*. These overheads were introduced with a toy example. In this section, we look at how our plain C++ and Saaz examples from Section 3.3 contribute overhead. In particular, we will focus on two queries, the simpler Planar Average query (Figure 4.3), and the more complicated yz-Dissipation query (Figure 4.5). Code blocks for plain C++ are colored identically to comparable code blocks in Saaz.

The declaration of the Saaz array in Figure 4.3b Line 7 (b.7), incurs Encapsulation overhead as the domain object (`dmn`) must be collapsed (Section 3.2.2) to get the subdomain along a particular axis (in this case, the y axis). This performs the simple arithmetic that is done with stack-local variables at Line a.7 in the plain C++ version. The `Collapse` call exhibits Encapsulation overhead as it hides the member accesses and encapsulations behind function calls.

Similar Encapsulation overhead occurs in the loops starting on Lines b.9 and b.14 that use iterator objects to cover the points within their respective 1-dimensional and 2-dimensional domains. The use of these objects also prevents the compiler from understanding exactly how the loop operates: where it starts, where it ends, and how it advances. This lack of clarity introduces Isolation overheads,

```
1  double* PlanarAverage(double* arr,
2      int x_min, int x_max, int y_min, int y_max,
3      int z_min, int z_max)
4  {
5    int i,j,k;
6    unsigned long long idx, planesize;
7    double* avg = new double[y_max - y_min + 1];
8    planesize = ((z_max-z_min+1) * (x_max-x_min+1));
9    for (j = y_min; j < y_max + 1; ++j)
10   {
11
12
13     avg[j-y_min] = 0;
14     for (i = x_min; i < x_max + 1; ++i)
15     {
16       for (k = z_min; k < z_max + 1; ++k)
17       {
18         idx = (k - z_min) + ((j - y_min) +
19               (i - x_min) * (y_max - y_min+1))
20                           * (z_max - z_min+1);
21         avg[j-y_min] += arr[idx];
22       }
23     }
24     avg[j-y_min] /= planesize;
25   }
26   return avg;
27 }
```

(a): Plain C++

```
1  Array1 PlanarAverage(
2      Array3 arr,
3      unsigned int Y_AXIS)
4  {
5
6    Domain3 dmn = arr.Domain();
7    Array1 avg(dmn.Collapse(Y_AXIS));
8
9    for(Iterator1 j = dmn.begin(Y_AXIS);
10       !j.end(); ++j)
11   {
12     Point1 q = j.Collapse();
13     avg[q] = 0;
14     for(Iterator2 ik=j.Slice().begin();
15         !ik.end(); ++ik)
16     {
17
18       Point3 p = j.Promote(ik);
19
20
21       avg[q] += arr[p];
22     }
23
24     avg[q] /= j.Slice().Size();
25   }
26   return avg;
27 }
```

(b): Saaz

Figure 4.3: The Planar Average query.

Table 4.1: Overheads from Saaz abstractions in the Planar Average and yz-Dissipation queries.

			Encapsulation	Isolation	Generalization
yz-Dissipation	Planar Avg.	Domain Objects	✓		
		Iterators	✓	✓	
		Abstracted Axis Mapping (`Y_AXIS`)			✓
		Multiple Array Layouts		✓	✓
		Point Arithmetic			✓
		Multiple Arrays		✓	✓

preventing the compiler from vectorizing or parallelizing the loops. Furthermore, the iterator introduces Encapsulation overheads not just from the function calls which are its conditional and increment operations, but also inside them as it accesses domain bounds through the `dmn` and `j.Slice()` domain objects.

`Collapse` (Line b.12, Section 3.2.4) converts the iterator from the 3-dimensional point along the leading edge of the domain into a 1-dimensional point. Its overheads fit primarily into the Encapsulation category.

`Promote` (Line b.18, Section 3.2.4) combines the 1-dimensional and 2-dimensional points from `j` and `ik` to form a 3-dimensional point. These overheads fit primarily into the Generalization category. It is because of the particular di-

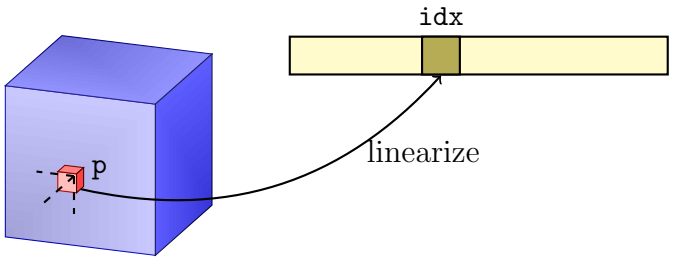


Figure 4.4: The linearization operation converts a 3-dimensional point (`p`) to a linear offset in memory (`idx`).

mensions over which `j` (and thus `ik`) iterate that this operation must be able to recombine them in a particular fashion.

The indexing of the array (Line b.21) requires that the point `p` be converted from a multidimensional point-valued object into a linear offset in memory. This *linearization* operation (Figure 4.4) is different for each different layout. Picking the right computation for a particular layout can be expensive. The plain C++ implementation performs the linearization inline (Lines a.18-20), and is able to access the bounds directly instead of querying the domain object (which would incur Encapsulation overheads). Because of Generalization overheads in Saaz (the layout is not known), the Saaz implementation cannot be inlined, and thus neither can the accesses to domain bounds.

Finally, Line b.24 divides the running sum by the number of points in the domain. In the plain C++ version, this can be factored out of the loop (Line a.8), but in the Saaz version, it is dependent upon the iterator variable, `j`, and so cannot be factored out.

Table 4.1 shows how Saaz features impose different overheads. Many of these overheads are illustrated by our Planar Average query, but some require a more complicated query such as yz-Dissipation (Figure 4.5). The yz-Dissipation query has the same form as the Planar Average query, with two main differences: it uses multiple arrays, and it computes derivatives using finite differences.

Derivatives in Saaz are usually expressed as a finite difference calculation, using adjacent points. Point arithmetic in Saaz makes accessing neighboring (including adjacent) points simple; it is relatively cheap on its own (just a call to the overloaded `operator+` function, which can be inlined). However, when multiple dif-

```
1  double* YZDissipation(double* v, double* w,
2      double* v_avg, double* w_avg,
3      int x_min, int x_max, int y_min, int y_max,
4      int z_min, int z_max,
5      double dz, double dy)
6  {
7
8    double* yzdiss =
9        new double[y_max - y_min + 1];
10   yzdiss[0] = 0;
11   yzdiss[y_max-y_min] = 0;
12   double tmp1,tmp2,cur;
13   unsigned long long offset_pz, offset_mz;
14   unsigned long long offset_py, offset_my;
15   unsigned long long planesize;
16   planesize = (z_max-z_min-1) * (x_max-x_min-1);
17   for (int j = y_min + 1; j < y_max; ++j)
18   {
19
20
21     yzdiss[j-y_min] = 0;
22     for (int i = x_min + 1; i < x_max; ++i)
23     {
24       for (int k = z_min + 1; k < z_max; ++k)
25       {
26         offset_pz =
27             (k+1 - z_min) + ((j - y_min) +
28             (i - x_min) * (y_max - y_min +1))
29             * (z_max - z_min + 1);
30         offset_mz =
31             (k-1 - z_min) + ((j - y_min) +
32             (i - x_min) * (y_max - y_min +1))
33             * (z_max - z_min + 1);
34         offset_py =
35             (k - z_min) + ((j+1 - y_min) +
36             (i - x_min) * (y_max - y_min +1))
37             * (z_max - z_min + 1);
38         offset_my =
39             (k - z_min) + ((j-1 - y_min) +
40             (i - x_min) * (y_max - y_min +1))
41             * (z_max - z_min + 1);
42         tmp1 = (((v[offset_pz]-v_avg[j-y_min]) -
43               (v[offset_mz]-v_avg[j-y_min]))/dz);
44         tmp2=(((w[offset_py]-w_avg[j-y_min+1]) -
45             (w[offset_my]-w_avg[j-y_min-1]))/dy);
46         cur  = tmp1*tmp1 + tmp2*tmp2;
47         cur /= 4.0;
48         yzdiss[j-y_min] += cur;
49       }
50     }
51     yzdiss[j-y_min] /= planesize;
52   }
53   return yzdiss;
54 }
```

(a): Plain C++

```
1  Array1 YZDissipation(Array3 v,Array3 w,
2      Array1 v_avg, Array1 w_avg,
3      unsigned int Y_AXIS,
4      Point3 PY, Point3 PZ,
5      double dz, double dy)
6  {
7    Domain3 dom = v.Domain().Inset(1);
8    Array1 yzdiss(
9        v.Domain().Collapse(Y_AXIS));
10
11
12   double tmp1,tmp2,cur;
13
14
15
16
17   for (Iterator1 j = dom.begin(Y_AXIS);
18        !j.end(); ++j)
19   {
20     Point1 q = j.Collapse();
21     yzdiss[q] = 0;
22     for(Iterator2 ik=j.Slice().begin();
23         !ik.end(); ++ik)
24     {
25       Point3 p = j.Promote(ik);
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42       tmp1=(((v[p + PZ]) - v_avg[q]) -
43           (v[p - PZ]) - v_avg[q])) / dz);
44       tmp2=(((w[p + PY]) - w_avg[q+1]) -
45           (w[p - PY]) - w_avg[q-1]))/dy);
46       cur  = tmp1*tmp1 + tmp2*tmp2;
47       cur /= 4.0;
48       yzdiss[q] += cur;
49     }
50
51     yzdiss[q] /= j.Slice().Size();
52   }
53   return yzdiss;
54 }
```

(b): Saaz

Figure 4.5: The yz-Dissipation query.

ferent points are accessed, each individual point must be linearized. These are the calculations in Figure 4.5, Lines a.27-41. Using different arrays makes the matter even worse: if they have different layouts or domains, then this linearization must be done separately for each layout/domain/point combination.

While Saaz makes queries simpler, more compact, and more interoperable, it also introduces lots of overheads. We quantify the costs of these overheads in

Chapter 6, after mitigating them in Chapter 5.

4.3 Related Work

We have placed library overheads into three categories: *Encapsulation*, *Isolation*, and *Generalization*. While library overheads are not always placed into our categories, many individual overheads expressed by these categories have been identified before and have motivated attempts to reduce or eliminate them. In this section we show how two existing techniques are not effective in treating the compiler optimization issues present in Saaz.

4.3.1 Templates

A well-known approach to address many concerns about library overheads lies in the use of templates. Expression Templates [Vel96] [VJ02] make use of compile-time computations and inlining to minimize the overheads associated with function composition, Encapsulation of operations, and function pointers. Templates are a powerful tool and can address a significant portion of these overheads, particularly those from Encapsulation. There are several reasons we avoid them: (1) templates are poor tools for optimizing across objects, (2) they require compile-time values, and (3) they have poor error message support.

4.3.1.1 Optimizing Across Objects

Templates are not helpful in optimizing across objects: they cannot eliminate duplicate metadata, resolve aliases, or optimize call sequences. Consider the overheads in Figure 4.5, which makes use of multiple array objects, in particular, `v` and `w`. Under most circumstances, these arrays will hold duplicate metadata (their domains).

Most general-purpose compilers can inline small functions, such as those accessing the bounds of domain objects; template-based libraries rely heavily on this inlining capability. However, templates cannot keep track of the fact that the domain objects used by different array objects are identical. Domain objects are

pointers, and thus runtime values. Pointers (except for function pointers) cannot be used as template arguments: `Array<dmn>` is invalid. The bounds of the domains could be used as template parameters, but then they must be compile-time constants, and not runtime values as required for data that lives on disk. Domain bounds cannot be used as template arguments: `Array<x_lo, x_hi>` is not acceptable. If the domains are not known to be identical, then the common operations cannot be eliminated. Furthermore, the duplicated metadata remains. The Expression Templates paradigm can be a good solution for function composition, but it is not very good for optimizing across sequences of calls. This makes Expression Templates inappropriate for addressing the Isolation overheads which cause redundant computations.

Saaz's support for multiple array layouts adds significant overhead to array-indexing operations. Each array must linearize the multidimensional point used as an index. Multiple indexing operations will frequently share many common subexpressions, but because of Isolation overheads, these are not identified by a general-purpose compiler. These common subexpressions are a frequent occurrence in structured grid problems, since they frequently access multiple arrays at identical or nearby locations. For example, in Figure 4.5a, computations are shared between Lines 27, 31, 35, and 39, and between Lines 28, 32, 36, and 40, and from one line to the next (reproduced below).

```
25 offset_pz =
26   (k+1 - z_min) + ((j - y_min) +
27   (i - x_min) * (y_max - y_min +1))
28   * (z_max - z_min + 1);
29 offset_mz =
30   (k-1 - z_min) + ((j - y_min) +
31   (i - x_min) * (y_max - y_min +1))
32   * (z_max - z_min + 1);
33 offset_py =
34   (k - z_min) + ((j+1 - y_min) +
35   (i - x_min) * (y_max - y_min +1))
36   * (z_max - z_min + 1);
37 offset_my =
38   (k - z_min) + ((j-1 - y_min) +
39   (i - x_min) * (y_max - y_min +1))
40   * (z_max - z_min + 1);
```

To optimize these calculations with templates, the templates would have to be able to recognize when arrays with the same layout and defined over the same domain are indexed at similar locations. Unfortunately, template support for reasoning about objects is tenuous at best; templates are better at optimizing expression

trees than accesses to data.

If layout were fixed (or even a template argument) then templates might be able to identify common index expressions, but they would have to symbolically reason about the point being passed as an argument. A fixed layout would help a compiler inline the linearization calculation. The compiler could then optimize the common subexpressions within the linearization calculations. However, once inlined, these calculations would still be using their original object's datamembers. Even if these datamembers hold common metadata, a general-purpose compiler is unable, for example, to identify as common the expressions stemming from the indexing's linearization calculations in Figure 4.5b Lines 42-45, reproduced below.[2]

```
41 tmp1=(((v[p + PZ] - v_avg[j]) -
42    (v[p - PZ] - v_avg[j])) / dz);
43 tmp2=(((w[p + PY] - w_avg[j+1]) -
44    (w[p - PY] - w_avg[j-1]))/dy);
```

Templates still will not help significantly.

More complex expressions are even harder. Templates cannot recognize the common index expressions that arise in complicated operations, such as those of `v` and `w` in Figure 4.5 (Lines b.42-45: `p+PZ`, `p-PZ`, `p+PY`, and `p-PY`).[3] These index expressions share the point `p` and index arrays which share a domain and layout. Therefore, the linearizations of these index expressions have common subexpressions. If the yz-Dissipation calculation (Figure 4.5) is converted into an expression tree (such as with Expression Templates), it will be at least five levels deep and involve at least eleven different operations. Even if all $11^5 = 161,051$ were enumerated (via a code-generator, perhaps), the compiler would be slowed to the point of uselessness. Enumerating all the template patterns to match and optimize these indexing expressions (via specialization) is unfeasible.

Templates operate on expressions, not objects. It is possible to extend the scope of templates by converting statements to expressions. Overloading the comma operator (`operator,`) and replacing statement-terminators with commas will convert statements into expressions, and a template implementation will be able to handle them. Overloading the comma operator, however, significantly

[2]Our tests showed that even for relatively simple examples, both GNU and Intel's compilers failed to identify the data as common, and thus did not eliminate common subexpressions.

[3]Technically, templates are Turing-complete, so these are practical, not theoretical limits.

changes the semantics of the C++ language, raising interoperability issues. Commas are *sequence points*: they define a strict ordering of operations. When the operator is overloaded, however, the comma is no longer a sequence point: both sides become mere function arguments, and as such, are evaluated in an undefined order [Dew03]. For example, for integers `i` and `j`, incrementing and then selecting it will ensure the increment happens first.

```
1   int i = 1;
2   ( ++i, i ); // result = 2
3   int j = 1;
4   ( j++, j ); // result = 2
```

Consider an `Object` which holds an integer and forwards the increment operators on itself to that datamember. If we overload the comma operator, the increment and non-increment are treated as function arguments to the comma operator and evaluated in an undefined order.

```
1   class Object
2   {
3   public:
4     Object(int data_) : data(data_) { }
5     Object& operator++()   //prefix
6     { ++data; return *this; }
7     Object operator++(int) //postfix
8     { Object tmp = *this; ++*this; return tmp; }
9     Object& operator,(Object& other)
10    { return other; }
11  private:
12    int data;
13  };
14
15  Object A(1);
16  ( ++A, A ); // result = 2
17  Object B(1)
18  ( B++, B ); // result = 1
```

This can lead to different behavior between the overloaded version and the non-overloaded version.

`boost::phoenix` [dGMH] is an interesting case. The Phoenix library implements a lazy version of C++ from within C++, making heavy use of templates, inlining, and overloading of the comma operator. By using lazy evaluation, large expressions can be built which make use of multiple objects in multiple ways, and transformations can be done using template evaluation operators. Unfortunately, coding in this style is exceedingly complicated. Since the capability is implemented in C++, the C++ compiler gives error messages not for the embedded version of C++ the programmer is thinking of, but the C++ language in which it is embedded. These errors are difficult to understand, even for a seasoned C++ programmer. It would be very difficult both to write a library in this style and to

use it.

4.3.1.2 Compile-Time Values

We must distinguish between compile-time values and runtime parameters. Certain parts of C++ require a compile-time value. These include template parameters and array sizes (e.g. `sz` in `char foo[sz]`).[4] Because Saaz is intended to analyze data coming from disk, array layout is a configuration parameter determined at runtime when the file is loaded. While it would be possible to specify layout as a template argument, doing so would commit layout to being a compile-time value. Such a constraint does not meet Saaz's design requirements.

Compile-time values are not the only ones which can be determined at compile-time. Under certain conditions runtime parameters can also be identified. Where runtime conditions can be identified, a compiler, unlike templates, is able to use that information to perform optimizations. For example, the `new[]` operator takes an integer that can be a constant:

```
1 const int sz = 512;
2 char* buffer = new char[sz];}.
```

In this case, the size of `buffer` is a runtime parameter, but can be identified at compile-time. If it can be determined that the pointer is not leaked, a compiler could make the allocation on the stack instead of the heap. Templates cannot perform evaluations based on runtime conditions.

Compile-time optimization (by a customized compiler or templates) cannot be based on runtime conditions which cannot be identified (without checks). Where a property cannot be verified, the compiler has three options: the compiler could either do nothing; the compiler could ensure it's assumption is met, perhaps by inserting conversion code; or the compiler could specialize the code and put a conditional test to determine which path to take. A conditional test for the constant-size array might look like, for example:

```
1 int sz = ... ;
2 if (sz == 512)
3 { /*allocate on stack*/ }
4 else
5 { char* buffer = new char[sz]; }
```

[4] Array sizes are allowed to be variable after C99 or when using extensions such as Variable Length Arrays (VLA).

We could utilize different types for configuration parameters (e.g. array layout as row-major or column-major) determined at runtime (such as from a file) vs. compile-time. We could then optimize where we can for compile-time parameters that are known, while also retaining flexibility when they are not. Doing so, however, would require implementing multiple variants of the library. Preserving efficient interoperability between different configurations would lead to a combinatoric explosion for different configuration sets, not all of which are presently known. This would make Saaz very difficult to maintain and extend for different configurations.

4.3.1.3 Error Messages

Template error messages are complicated and require experience to understand. This is a very sensitive point for our users. While other systems can depend upon having experienced programmers who have a good grasp of templates, we cannot. Saaz is targeted to computational scientists, many of whom have only rudimentary programming experience gained informally in Matlab and Fortran. Even experienced programmers can have difficulty understanding how templates are expanded on demand, and how this causes error messages to be context-dependent (see Appendix D).

4.3.2 Virtual Function Calls

Virtual functions came into use with the introduction of object-oriented programming [DMN68]. While function pointers existed before, they had not been used so readily or commonly, and were not part of such an ubiquitous design pattern. Both virtual function calls and calls through function pointers have overheads associated with indirection; how expensive they are can vary. Both also introduce significant *Isolation* overheads, preventing the compiler from knowing what code is going to execute.

There are limitations on a compiler's ability to inline functions. Functions that are in a different compilation unit cannot be inlined. Virtual methods or function pointers are determined at run time and thus cannot always be inlined.

While a human programmer may be able to identify which function body would be used, general-purpose compilers do not have the heuristics or the domain-specific knowledge available to discover which function is being called.

The cost of virtual function calls is a well-known problem, and there has been some work in eliminating or reducing the overhead. Calder and Grunwald [CG94] examine a dynamic profiling technique to identify the targets of virtual function calls. Bacon and Sweeney [BS96] look at the ability of three static analysis techniques to identify virtual function calls: Unique Name, Class Hierarchy Analysis, and Rapid Type Analysis (RTA). The most effective static technique is RTA. RTA identifies which types are actually instantiated and resolves virtual function calls where there is no ambiguity. This approach is inapplicable to Saaz, which uses a function pointer instead of a virtual function, although Saaz could be rewritten to support it.

Saaz does not use virtual function calls because function pointers are cheaper. RTA's principles would, however, still be applicable. RTA, however, is still limited in the overheads it can address. RTA can identify virtual function calls but cannot, for example, eliminate redundant subexpressions which use different datamembers.

Consider the case in Figure 4.6. RTA performs analysis of source code and tracks which types are actually instantiated. In this case, only `Child1` ever gets instantiated. RTA can see that `Child1` is the only subclass of `Base` which ever gets instantiated. Thus, all calls to `DoWork` will be to `Child1::DoWork`, and can be called without using the virtual function table. If, however, `kid2` were assigned a `new Child2()` (Line 19), then it would be unclear to RTA as to what function `DoWork` referred.

RTA relies upon identifying what objects are created. If objects with heterogeneous subtypes are created, then RTA is not able to resolve virtual function calls. In particular, if arrays of different layouts have been created in the program, RTA would not be able to optimize the layout calls at all. This will prevent it from optimizing something like: `A(x,y) = B(x,y) + C(x,y)` when `A` and `B` have the same layout, but `C` does not. Even `A(x,y) = B(x,y)` would not be able to be

```
1  struct Base
2  {
3    virtual void DoWork() = 0;
4  };
5
6  struct Child1 : public Base
7  {
8    virtual void DoWork() { return 1; }
9  };
10
11 struct Child2 : public Base
12 {
13   virtual void DoWork() { return 2; }
14 };
15
16 int main()
17 {
18   Base* kid1 = new Child1();
19   Base* kid2 = new Child1();
20   Factory.Recruit(kid1, kid2);
21   kid1->DoWork();
22   kid2->DoWork();
23   return 0;
24 }
```

Figure 4.6: An example of RTA at work.

optimized.

Chapter 5

Tettnang

To some degree, all libraries will exhibit overheads in our three categories: *Encapsulation*, *Isolation*, and *Generalization*. Saaz is no exception. Saaz brings to C++ true multidimensional arrays which can have differing layouts. The dynamism required for different layouts dramatically decreases performance, both through Generalization overheads, and through Isolation overheads (stemming from the compiler's inability to optimize dynamic code).

There have been several strategies for dealing with library overheads. Some approaches are more general than others. Expression Templates [Vel96] [VJ02], for example, are a programming methodology which makes use of existing compiler abilities, particularly C++ templates and inlining, to generate efficient code. Rapid Type Analysis (RTA) examines which subtypes are present in a program, eliminating virtual function calls where there is only one which could be called. Telescoping Languages [KBC+05] analyze libraries and generate a compiler to optimize them, focussing primarily on statically-typing dynamically-typed languages. Broadway [GL00] provides pragmas which can be used to annotate libraries and specify transformations which can be applied to code which uses those libraries.

These strategies, while they may be useful for certain kinds of overheads, are incapable of addressing all the overheads introduced by libraries such as Saaz (see Section 5.4). The incompleteness of these generalized tools suggests a different approach. Instead of building a general tool, we first work to understand and cater to a specific case.

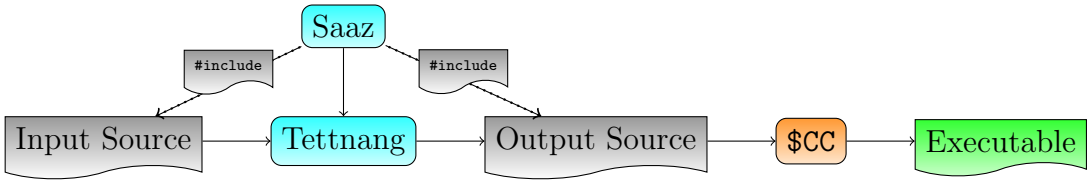


Figure 5.1: The workflow for Tettnang.

Instead of developing a general way of addressing library overheads (such as library annotation) we tailor our translator, Tettnang, to a particular library, building information about Saaz and its semantics into the Tettnang translator. We think it important to see how fully a specialized implementation can address such overheads. We want to fully understand specific cases before we generalize. Had we generalized too early, then we may have confused limitations on a library's ability to be analyzed with limitations of the technique or of the translator implementation. Because we have control over both the library (Saaz) and the translator (Tettnang), we can tailor each to the other. This should enable us to achieve the maximum possible performance this technique can provide.

The overheads we are addressing come from specific uses of language facilities. To address these overheads we have developed Tettnang as a source-to-source translator. Compared to machine operations, source-code uses higher-level language primitives, making available a greater understanding of the user-level operations being performed. Tettnang works with a representation of the users's source-code instead of a lower-level intermediate or machine language. This lets it address the overheads at the source[1].

By generating source-code we simplify verification of the translator's output and its differences from plain C++, that is, C++ with no user-defined abstractions. Source-code generation also avoids the nuances of assembly-code generation, which can have a significant impact on performance.

To identify overheads, we examine the differences between queries written in Saaz and those written in plain C++. These comparisons provide us with overheads to target. Because we are performing source-to-source translation, we have an easy map from our observations to the transformations we want to support.

[1]Haha!

By applying new transformations incrementally, we move the two implementations closer and closer.

Following a discussion of the framework (Rose) we used to build our translator, we will present a high-level view of how we use type-refinements to manage identified configuration parameters, as well as how those refinements are conceptually identified. We will then delve into the internals of Tettnang. Finally, we conclude with a discussion of related work.

5.1 The Rose Compiler Framework

Rose [QMPS02] is an open-source source-to-source translation framework developed and maintained at Lawrence Livermore National Laboratory. Rose provides a number of analysis tools and operates on multiple source languages including C, C++, Fortran, and x86 Assembly. Support for Python is currently in development. Rose is released under a BSD license.

The framework uses the C++ parser from the Edison Design Group (EDG). Source code is parsed and stored in an intermediate representation (IR) called Sage III. The IR is a (mostly) language-independent representation of source-code and formed into an Abstract Syntax Tree (AST). All transformations and rewrites are performed on the AST. Sage III has 782 nodes to represent different source-code concepts (280 of which are for assembly instructions). The Rose backend traverses the AST and outputs a new source-file containing original comments and control structure. This generated source-file can then be run through a conventional backend compiler (such as Intel's `icc` or GNU's `g++`).

Rose provides many analysis tools on its own, including loop unrolling, loop normalization, def-use analysis, and constant propagation. For the most part, we do not make use of these tools. Rose's built-in inlining tool was modified slightly to support some of the needs for handling the Cascade (Appendix B.2) interface to Saaz. By inlining Cascade's wrappers, we do not need to include their semantics in Tettnang.

There are other compiler frameworks for code-analysis. Low Level Virtual

Machine (LLVM) [LA04] is an open-source compiler framework. LLVM, however, is designed for binary optimization and provides code at a very low-level IR, outputting binary object code. The Rose AST is very close to the source-code, letting us target library-specific overheads more easily. Our goal is to generate source-code similar to plain C++.

5.2 Tracking Configurations

Queries written with Saaz hide the data schema by factoring it out of query implementations and into configuration parameters behind Saaz's interface. This introduces overheads. As a result, code written with Saaz is more general and more interoperable than code written in plain C++ or Fortran. Because it is the ability to work with multiple data schemas that introduces the overhead we are trying to eliminate, Tettnang must identify the schema by identifying the values of the configuration parameters. Once the parameter values are identified, Tettnang can examine each query's implementation and replace those parts which use Saaz's generality with a version tailored to the specific data schema. Thus, Tettnang transforms general query implementations by specializing them, moving their implementations and performance closer and closer to plain C++.

Saaz configuration parameters are stored as immutable properties of objects. As a result of not needing to determine which and how object properties are modified, Tettnang can perform a simplified side-effect analysis. Because Tettnang has semantic knowledge about how Saaz objects are constructed, it can extract configuration details from parameters to object constructors. Alternatively, when Saaz array objects are loaded from disk, Tettnang could open the files and check their data schema. Inserting a call to verify these settings would let Tettnang ensure that these configurations are accurate at runtime.

The plain C++ implementations serve as a baseline for assessing the effectiveness of Tettnang. Domain-scientists are, in general, not experienced with aggressive optimization techniques. We feel it is valid to compare to these unoptimized implementations because even if domain-scientists did heavily optimize their

```
1 for (int i = 0; i < i_max; ++i)
2 {
3   for (int j = 0; j < j_max; ++j)
4   {
5     int idx = i * j_max + j;
6     A[idx] = B[idx]
7   }
8 }
```

(a): Before optimizations.

```
1 #pragma omp parallel for
2 for (int iblock = 0; iblock < i_max; iblock +=
      I_BLOCK_SIZE)
3 {
4   for (int jblock = 0; jblock < j_max; jblock +=
       J_BLOCK_SIZE)
5   {
6     int ilim = std::min(iblock+I_BLOCK_SIZE,i_max);
7     int jlim = std::min(jblock+J_BLOCK_SIZE,j_max);
8     for (int i = iblock; i < ilim; ++i)
9     {
10      #pragma vector always
11      for (int j = jblock; j < jlim; ++j)
12      {
13        A[i][j] = B[i][j]
14      }
15    }
16  }
17 }
```

(b): After optimizations.

Figure 5.2: Example of expert transformations including blocking-for-cache, parallelization via OpenMP, and vectorization.

code, many optimization techniques that an expert could apply to the code are orthogonal to the transformations that Tettnang performs. Blocking loops for cache, for example, can significantly increase the effectiveness of the cache, but does not decrease library overheads. Other orthogonal optimizations include parallelization, vectorization, and loop skewing. Figure 5.2 shows how the addition of two arrays may be optimized within plain C++. While Tettnang's analysis may make transformations such as blocking-for-cache easier to implement, these transformations can be applied after those performed by Tettnang, or even implemented by Saaz itself. Since Saaz iterators cover their iteration space in an undefined order, to improve performance Saaz could transparently change the order in which the iteration space is covered. We consider these sorts of optimizing transformations outside of the scope of this thesis, but a possible venue for future work.

5.2.1 Type Refinements

Dependent types [ML85] enhance the expressiveness of type systems by annotating types with a predicate (that is, dependent types *refine* the type system's types). Instead of a variable being restricted to merely holding integer values, for example, it can be said to be holding only integer values between 1 and 99. A dependent type system represents this restriction with a refinement of the types

of variables. This idea can be expressed for a variable `i` by the following type:

$$\texttt{i} : \{v \text{ of } \texttt{int} : 1 \leq v \wedge v \leq 99\}$$

The predicate above can assist bounds checking for array indexing, constant propagation, or elimination of conditionals. Liquid Types [RKJ08] infers these predicates with minimal annotations to OCaml [KRJ10] and C [RBKJ12] programs.

We use a similar idea. Tettnang keeps track of configuration parameters through a series of simple type-refinements on Saaz objects. Unlike other refined-types, these type-refinements do not express predicates over what values an expression can take, but rather express properties of the expression that relate to configuration parameters and data schema. Instead of logical predicates, we track and maintain a listing of particular properties of variables; instead of performing safety-checks, we optimize code through specialization.

Saaz guarantees that the configuration parameters of an object are immutable, meaning that Tettnang usually identifies them when an object is first constructed. Configuration parameters may also be identified when a file is loaded from disk or when it is assigned to. Configuration immutability also grants Tettnang the flexibility to identify configuration parameters later in the program, and then use them before the point of identification. Immutability enables Tettnang to avoid performing significant interprocedural analysis to identify and track how properties change. While such analysis is a practical matter and not fundamental, in practice immutability makes a significant difference. Designers of libraries other than Saaz should keep in mind the effect of mutability in stifling analysis. While making all data items immutable may be untenable, there are many circumstances in which mutability of certain object parameters and properties is not required (as is the case with Saaz). Indeed, mutability of object parameters may only be required for a few programs or a few objects out of many.[2]

Our full refinement-type grammar can be found in Figure 5.3. We specify base types before the colons: `Array`, `Domain`, `Iterator`, and `Aggregator`. Array variables have type *array_type*, domain variables have type *domain_type*,

[2]Some languages, such as OCaml, only support mutability reluctantly and many programs in OCaml don't make use of it at all.

iterator variables have type *iterator_type*, and aggregator variables have type *aggregator_type*. `Unknown`, `RowMaj`, and `ColMaj` are terminals representing an array's layout. The terminal `domain_var` is a reference to an in-scope object (that is, an object with a lifetime at least that of the array variable) with type *domain_type*, possibly the array variable's member variable, which holds the array's domain. The terminal `new_var` is a reference to an in-scope variable (with a lifetime matching that of the domain) created by Tettnang. This variable will be constant, and as a translator-generated variable, is never assigned to or aliased. The terminal `array_var` is a reference to an in-scope object with type *array_type*. Array objects are tracked so equivalences between arrays can be maintained and additional array objects can have their types refined. The non-terminal *bounds* is an n-tuple of integer constants or `URT`, where n is the number of dimensions in the domain, and the terminal `URT` represents an unknown (runtime) value which is the default for domain bounds. The non-terminal *dimensions* for iterators is a tuple which maps the coordinates of an iterator back to the original coordinates of the domain it is iterating over. This is important for the operation of point promotion. The non-terminal *array_list* is an m-tuple of arrays which correspond to member variables of the aggregator objects.

We call two refined types equivalent if all their terminals are equivalent and terminal variables refer to the same object.

5.2.2 Identifying Configurations

Saaz sets layout and domain information in array constructors. Some languages, such as Fortran 95, allow array objects to change their bounds (size). They consider layout a language convention rather than an object attribute. As a consequence of a fixed layout, the compiler can optimize linearization calculations. Saaz allows layouts to be established at runtime. Array layout and domain are runtime static quantities (the equivalent of `const` data-members). Since domains are constant, and cannot change from one part of the program to another, Tettnang can avoid a lot of side-effect and dataflow analysis. Tettnang's refined types can reference immutable objects such as domains and immutable properties such as

$$
\begin{aligned}
\mathit{array\_type} &::= \{\texttt{Array} : \mathit{array\_refinement}\} \\
\mathit{array\_refinement} &::= \texttt{layout=}\mathit{layout} \wedge \texttt{domain=domain\_var} \\
\mathit{layout} &::= \texttt{Unknown}|\texttt{RowMaj}|\texttt{ColMaj} \\
\mathit{domain\_type} &::= \{\texttt{Domain} : \mathit{domain\_refinement}\} \\
\mathit{domain\_refinement} &::= \texttt{lwb=}\mathit{bounds} \wedge \texttt{upb=}\mathit{bounds} \\
\mathit{bounds} &::= (\mathit{bound}, ..., \mathit{bound}) \\
\mathit{bound} &::= \mathit{Int}|\texttt{new\_var} \\
\mathit{Int} &::= \texttt{int}|\texttt{URT} \\
\mathit{iterator\_type} &::= \{\texttt{Iterator} : \texttt{domain=domain\_var} \wedge \mathit{iterator\_refinement}\} \\
\mathit{iterator\_refinement} &::= \texttt{mapping=}\mathit{dimensions} \\
\mathit{dimensions} &::= (\mathit{Int}, ..., \mathit{Int}) \\
\mathit{aggregator\_type} &::= \{\texttt{Aggregator} : \mathit{array\_list}\} \\
\mathit{array\_list} &::= (\texttt{array\_var}, ..., \texttt{array\_var})
\end{aligned}
$$

Figure 5.3: Our refinement types keep track of configuration parameters.

array layout. In addition to the point being indexed, the linearization calculation requires two pieces of information, both of which are part of an array's refined type: the array's domain, and the array's layout.

Tettnang is able to use the arguments passed to an object's constructor to refine the object's type. To illustrate, we add constructors to our yz-Dissipation example from Figure 4.5. This modified code is shown in Figure 5.4. The integers from the constructor of the domain `dmn` are copied into the refinement-type's bounds `lwb` and `upb`. We say that two domains are *conforming* (and runtime equivalent) if they have equivalent refined types (that is, the upper and lower bounds are equivalent, respectively) where the bounds contain no `URT`s. The domain object itself, `dmn`, is referenced in the types of `v` and `w`. The refined type of `dmn` can then be retrieved, for example, to perform a conformity check on the domains. We say that two arrays are *conforming* if they have the same base type, the `domain` parts of their refined types are known and conforming, and the *layout* parts of their refined types match and are not `Unknown`. If Tettnang is unable to identify the domain from an array's constructor, the array's member variable holding it is used instead.

If `Y_AXIS` is a constant with a value of 1, then Tettnang is able to make

```
1   Domain3 dmn(x_lo,y_lo,z_lo, x_hi,y_hi,z_hi);
2   Array3 v(dmn, Layout::RowMaj);
3   Array3 w(dmn, Layout::RowMaj);
4   Array1 v_avg, w_avg;
5   const unsigned int Y_AXIS=1;
6   /* ... */
7   Array1 yzdiss(dmn.Collapse(Y_AXIS));
8   double cur;
9   for (Iterator1 j = dmn.begin(Y_AXIS); !j.end(); ++j)
10  {
11    Point1 q = j.Collapse();
12    yzdiss[q] = 0;
13    for (Iterator2 ik = j.Slice().begin(); !ik.end(); ++ik)
14    {
15      Point3 p = j.Promote(ik);
16      cur  = (((v[p + PZ] - v_avg[q]) -
17               (v[p - PZ] - v_avg[q])) / dz);
18      cur += (((w[p + PY] - w_avg[q+1]) -
19               (w[p - PY] - w_avg[q-1])) / dy);
20      cur *= cur;
21      cur /= 4.0;
22      yzdiss[q] += cur;
23    }
24    yzdiss[q] /= j.Slice().Size();
25  }
```

Figure 5.4: Our example of yz-Dissipation, with the addition of array declarations.

the following type refinements to Figure 5.4, where the variables at left have the refined types to the right of the colon:

$$\texttt{v} \quad : \{\texttt{Array} : \texttt{layout} = \texttt{RowMaj} \wedge \texttt{domain} = \texttt{dmn}\} \tag{5.1}$$

$$\texttt{w} \quad : \{\texttt{Array} : \texttt{layout} = \texttt{RowMaj} \wedge \texttt{domain} = \texttt{dmn}\} \tag{5.2}$$

$$\texttt{dmn} \quad : \left\{\texttt{Domain} : \begin{array}{l} \texttt{lwb} = \texttt{(x\_lo,y\_lo,z\_lo)} \wedge \\ \texttt{upb} = \texttt{(x\_hi,y\_hi,z\_hi)} \end{array}\right\} \tag{5.3}$$

$$\texttt{j} \quad : \{\texttt{Iterator} : \texttt{domain} = \texttt{dmn} \wedge \texttt{mapping} = \texttt{(1)}\} \tag{5.4}$$

$$\texttt{ik} \quad : \{\texttt{Iterator} : \texttt{domain} = \texttt{dmn} \wedge \texttt{mapping} = \texttt{(0,2)}\} \tag{5.5}$$

These refined types contain information about the program's data schema. Tettnang can now identify the dimensions and values traversed by Saaz iterators, how they form points, how the arrays are organized in memory, and thus how to index arrays. With this information, Tettnang can perform its transformations.

5.3 Implementation

Tettnang is organized into nine stages. The first and last are controlled by Rose: parsing the input file into an AST, and unparsing the modified AST to the output source file. The intermediate Tettnang modules preprocess code (*Cascade* and *Type-Refinement*), perform optimizing transformations (*For-Loops*, *Indexing*,

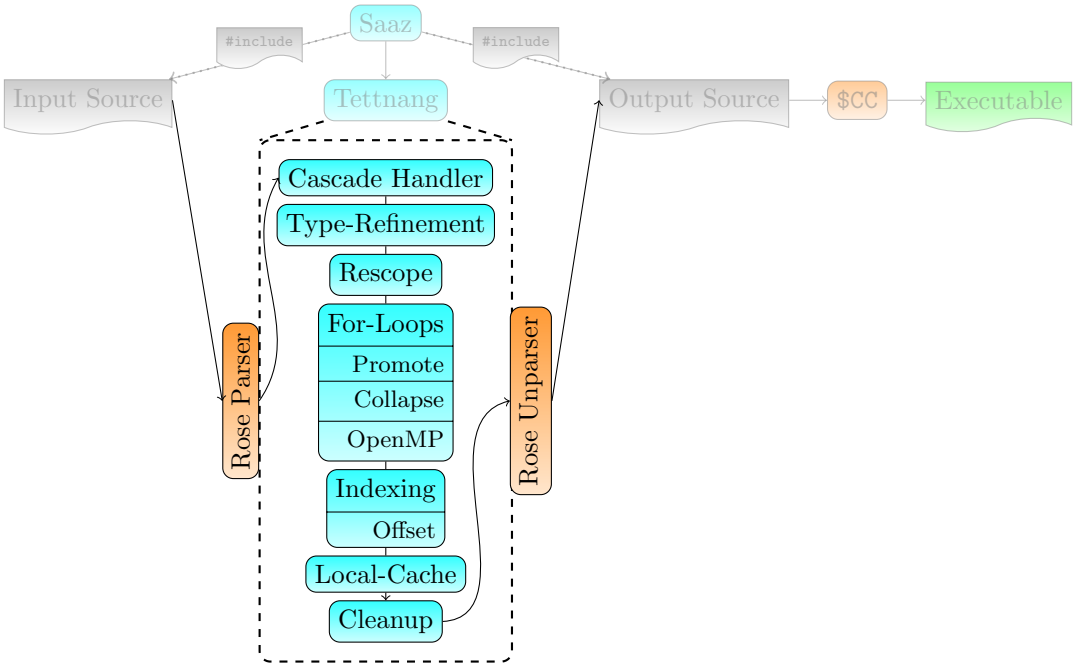


Figure 5.5: Tettnang's organization.

Local Cache), and normalize code (*Consolidate*, *Cleanup*). The staging of these parts is illustrated in Figure 5.5.

Except for the *Type-Refinement* module, which always runs, each module is enabled via command-line flags for optimization which are passed to the Tettnang translator. These flags are shown in Table 5.1. Except for the dependence upon the analysis provided by the Type-Refinement module, all modules are independent, and perform only local transformations.

Table 5.1: Flags passed to the Tettnang translator enable certain optimizations.

Flag	Module	Optimization
`--saaz:cascade`	Cascade	Handling the Cascade frontend to Saaz
`--saaz:0rescope`	Rescope	Use variables from outer scopes
`--saaz:0for`	For-Loop	Transform iterator for-loops into integer for-loops
`--saaz:0promote`	For-Loop	Optimize point promotion
`--saaz:0collapse`	For-Loop	Optimize collapsing iterators to lower dimensions
`--saaz:0openmp`	For-Loop	Insert pragmas for OpenMP parallelization
`--saaz:0index`	Indexing	Consolidate array linearization operations
`--saaz:0offset`	Indexing	Inline array linearization calculations
`--saaz:0cachelocal`	Local Cache	Introduce new variables to the stack
`--saaz:0reducevars`	Cleanup	Remove unused variables

5.3.1 The Cascade and Rescope Modules

Appendix B.2 introduces the Cascade frontend to Saaz. Cascade provides an interface to Saaz that renames some types and abstracts some operations that are common in CFD. Saaz is designed to support ad hoc queries, so presenting a restricted interface such as Cascade is somewhat anathema to Saaz's design goals. However, by compartmentalizing such an interface outside of Saaz, we do not change Saaz's core. We also allow users to ease their introduction to Saaz's more advanced features.

Tettnang starts with a module to handle Cascade's abstractions of Saaz. We did not want to cater Tettnang extensively to every frontend. Inside the *Cascade* module, no additional information is collected. Instead, the module overcomes Encapsulation overheads by inlining certain functions from the Cascade library, in particular, the correlation methods, and aggregators (including user-defined aggregators). Outside of the Cascade module, Cascade's has only a minimal effect on the Tracking module.

The inlining functionality is a slightly modified version of the inline operation from Rose. One complication of inlining is that it introduces additional copies of variables. The inline operation saves function arguments into new variables which it then uses in place of the function's parameters. The use of new variables for these parameters can introduce many new names for the same object. As a consequence, the *Rescope* module replaces array aliases from local scopes with the original aliases from more global scopes. The *Cleanup* module removes these (and other) unused variables.

5.3.2 The Type-Refinement Module

After eliminating the syntactic sugar from the frontends to Saaz (i.e. Cascade), Tettnang can perform type-refinement. Tettnang must refine the types of Saaz objects for use by later optimization passes. The *Type-Refinement* module traverses the user's source code, looking for declarations and assignments of Saaz objects. This module tracks the following entities:

Predicates so their use can be handled, and their use by iterators inlined;

Domains so their bounds can be identified in addition to their use in constructing arrays;

Arrays so their domain and layout can be identified in addition to their use in aggregators;

Aggregators so their use of arrays can be optimized;

Iterators so their `Collapse` and `Promote` operations can be rewritten;

Points so they can be eliminated as composite objects and have integers used instead;

Aside from refining the types of all the variables with Saaz types, we also keep track of variable assignments. When one variables is assigned to another, the receiver's type get updated, and a mapping is maintained between the variables. This also allows *Copy Propagation* [Muc97, p. 356], where some variables are replaced by their values (or other variables from an outer scope). Using variables from an outer scope makes it simple for the *Cleanup* module to eliminate local-variables from inner scopes when a more visible variable from an outer scope is available.

5.3.3 The For-Loop Module

The *For-Loop* module converts for-loops over Saaz iterators to for-loops over integers (See Section 3.2.4). The result is to enable a general-purpose compiler to understand what the loop is doing and to either parallelize or vectorize it, thus overcoming Isolation overheads. To do so, Tettnang matches for-loops that:

1. Declare a Saaz iterator to start at a domain's beginning: `Iterator i = dmn.begin();`
2. Are conditioned solely on the termination of that iteration: `i != dmn.end();` or `!i.end();`
3. Increment the iterator: `++i` or `i++`

If the iterator covers particular dimensions, then those are retrieved from the refined type of the iterator:

```
i = dmn.begin(X_AXIS, Z_AXIS)
```

Both the domain and the dimensions traversed had been inserted by the Type-Refinement module:

$$\texttt{i} : \{\texttt{Iterator} : \texttt{domain=dmn} \wedge \texttt{mapping=(X\_AXIS,Z\_AXIS)}\}$$

When multiple dimensions are traversed, the iterator's loop is replaced by nested for-loops over integers. Instead of using:

```
1 for (Iterator3 i = dmn.begin(1,0,2); !i.end(); ++i)
2 {
3   /*...*/
4 }
```

We get triply-nested loops:

```
1 for (Saaz::coord_t _i1_ = dmn.Min(1); _i1_ != dmn.Max(1) + 1; ++_i1_)
2 {
3   for (Saaz::coord_t _i2_ = dmn.Min(0); _i2_ != dmn.Max(0) + 1; ++_i2_)
4   {
5     for (Saaz::coord_t _i3_ = dmn.Min(2); _i3_ != dmn.Max(2) + 1; ++_i3_)
6     {
7       Iterator3 i = Saaz::Mk_Iterator(dmn,1,0,2,_i1_,_i2_,_i3_);
8       /*...*/
9     }
10  }
11 }
```

Note here that we still have the iterator, `i` (Line 7). It is important to preserve this object because it may be used inside the loop body for other tasks (such as indexing an array). Subsequent passes may be able to eliminate the iterator (the Cleanup module).

More complicated loops may include a predicate, or use nested Saaz iterators. The former is used for conditional queries (such as those restricted to inside or outside a flow structure such as a vortex core), the later for planar-average-based queries and aggregations (Section 3.3). Figure 5.6 shows an example. Because nested Saaz loops tend to make certain calls, the For-Loop module also handles optimizations for (different optimizations are colored as in Figure 5.6):

1. Collapsing iterators: extracting the correct m-dimensional point from an iterator which traverses m dimensions of an n-dimensional domain. Compare Figure 5.6, Line a.6 with Line b.6.

```
1
2  for (Iterator1 j = dmn.begin(1);
3       !j.end(); ++j)
4  {
5
6    Point1 pj = j.Collapse();
7
8
       for (Iterator2 ik = j.Slice().begin();
9           !ik.end(); ++ik)
10   {
11
12
13
14
15
16     Point3 p = j.Promote(ik);
17     /*...*/
18
19   }
20 }
```

(a): Loops over Saaz Iterators.

```
1  #omp parallel for
2  for (int _i1_ = dmn.Min(1);
3       _i1_ != dmn.Max(1) + 1; ++_i1_)
4  {
5    Iterator1 j_=Mk_Iterator(dmn,1,_i1_);
6    Point1 pj(_i1_);
7    Domain2 slice = j.Slice();
8    for(int _i2_ = slice.Min(0);
9        _i2_ != slice.Max(0) + 1; ++_i2_)
10   {
11     for(int _i3_ = slice.Min(1);
12         _i3_!=slice.Max(2) + 1; ++_i3_)
13     {
14       Iterator2 ik = Mk_Iterator(
15                        dmn,0,1,_i2_,_i3_);
16       Point3 p(_i2_, _i1_, _i3_);
17       /*...*/
18     }
19   }
20 }
```

(b): Loops over integers.

Figure 5.6: Eliminating Saaz iterators from nested loops. Corresponding parts are marked similarly.

2. Caching the `Slice` domain on the stack and replacing references with the new variable (Section 3.2.4, Figure 3.5c). The slice domain is stored in `slice` (Figure 5.6, Line b.7).

3. Combining points without using the `Promote` method of the iterator (`Promote` uses a virtual function call) (Section 3.2.4). This transformation rewrites Figure 5.6, Line a.16 as Line b.16.

4. Parallelize loops by inserting OpenMP directives at the top of the loops on Figure 5.6, Line b.1.

5.3.4 The Indexing Module

The *Indexing* module optimizes array indexing. Arrays in Saaz are not required to have a particular layout for their storage in memory. By supporting both row-major and column-major layouts, each Saaz array is compatible both in storage and in cache-benefits with arrays either from C or Fortran and Matlab.

The layout of an array is specified by a constructor parameter. Tettnang records this layout as a refinement to the type of the array variable. The Indexing module traverses the input program and finds all of the places in which a Saaz

array is indexed, and records the array and the indexing expression. Recall that two arrays are *conforming* if they have the same layout and their domains are conforming. Arrays conform at least to themselves, and to as many as all arrays in the program. When multiple conforming arrays are indexed in the same way, the linearization component of the array accesses can be *consolidated*. In this way, the overheads of index calculation will be incurred only once. For example, given refinement as in Equation 5.1 and Equation 5.2:

$$\texttt{v} \ : \{\texttt{Array} : \texttt{layout} = \texttt{RowMaj} \wedge \texttt{domain} = \texttt{dmn}\}$$

$$\texttt{w} \ : \{\texttt{Array} : \texttt{layout} = \texttt{RowMaj} \wedge \texttt{domain} = \texttt{dmn}\}$$

consider a loop which copies an array, thus indexing both the source and destination arrays at the same point:

```
1 for (Iterator3 i = dmn.begin(); !i.end(); ++i)
2 {
3
4   v[i] = w[i];
5 }
```

Without using the refined types we can only extract the indexing calculations to the data access (`Elements`) and the linearization operation (`LinearIdx`).

```
3
4   v.Elements(v.LinearIdx(i)) = w.Elements(w.LinearIdx(i));
```

However, since the refined types show that `v` and `w` are conforming, we know that the `LinearIdx` operation will behave the same for identical points. The points here are certainly identical, so the `LinearIdx` operation need only be performed once.

```
3   unsigned long long offset = v.LinearIdx(i);
4   v.Elements(offset) = w.Elements(offset);
```

The *Indexing* module will always consolidate linearization operations.

When we know the layout of an array, we know not just the semantics of the call to linearize a point (`LinearIdx`), but the implementation. Thus, we can inline calls to `LinearIdx` (option `--Ooffset`). A general-purpose compiler cannot perform this inlining because it does not have the greater context (i.e. the data schema, the storage layout of the arrays) at the call site.[3] General-purpose compilers do not have the same kind of semantic knowledge of a library that Tettnang does. In our example above, the `RowMaj` array `v` can have its `LinearIdx` call rewritten:

[3]As before, we tested GNU's `g++` compiler, version 4.7, and Intel's `icc`, version 12.0.

```
3   unsigned long long offset = (i.z-dmn.Min(2)) + ((i.y-dmn.Min(1)) +
        (i.x-dmn.Min(0)) * (dmn.Max(1)-dmn.Min(1)+1)) * (dmn.Max(2)-dmn.Min(2)+1);
```

Note that if the type of `v` was

$$\texttt{v} : \{\texttt{Array} : \texttt{layout=RowMaj} \wedge \texttt{domain=v.Domain()}\}$$

then `v.Domain()` would be used instead of `dmn`, but the accesses to `v[p]` and `w[p]` could not have been consolidated, as `v` and `w` would have different domains and thus non-conforming types.

This optimization replaces a function-pointer call with an expression semantically equivalent to the corresponding function, eliminating Generalization overheads. It also overcomes Isolation overheads, enabling the compiler to perform many other optimizations. For example, instead of querying the domain object with `dmn.Min(2)`, we can access the member variable directly as in `dmn.z_min`. Common sub-expressions can be eliminated. Some expressions (such as `dmn.Max(2) - dmn.Min(2) + 1`) can even be lifted out of the loop.

Finally, the Indexing module can optimize indexing at static offsets (also enabled by option `--Ooffset`). For example, if multiple arrays share a common index expression (even if they differ by a few grid points), their linearization operations can still be shared.

```
1 Point3 PY(0,1,0);
2 for (Iterator3 i = dmn.begin(); !i.end(); ++i)
3 {
4
5
6   v[i] = w[i+PY];
7 }
```

In this case, we can first linearize the offset, then tweak it slightly, like so:

```
1 Point3 PY(0,1,0);
2 for (Iterator3 i = dmn.begin(); !i.end(); ++i)
3 {
4   unsigned long long offset = (i.z-dmn.Min(2)) + ((i.y-dmn.Min(1)) +
        (i.x-dmn.Min(0)) * (dmn.Max(1)-dmn.Min(1)+1)) * (dmn.Max(2)-dmn.Min(2)+1);
5   unsigned long long offset_yp1 = offset + (dmn.Max(2) - dmn.Min(2) + 1);
6   v.Elements(offset) = w.Elements(offset_yp1);
7 }
```

We have now effectively consolidated the linearization calculations, even for non-identical points. This method of handling offsets is used no matter how large the offset. The resulting code may not be the most efficient way of calculating multiple offsets for every stencil (for example, a 9-point stencil in a plane), but it is superior to calculating an offset for each point individually.

offset_expression ::= *point_expression* [(+|-) *point_expression*]*
point_expression ::= `point_var` | *point_product* | `unit_point`
point_product ::= *constant*
| *constant* * *point_expression*
| *point_expression* * *constant*
constant ::= `int` | `double`

Figure 5.7: Offset-expression grammar.

The expressions supported as offsets are linear combinations of points using only: addition, subtraction, multiplication, and negation. Two types of expressions are supported: *offset_expression*, with the restriction that only one `point_var` variable is allowed (Figure 5.7); and `unit_point`, which is a terminal for a variable of type point with special properties. A `unit_point` has a unit value for one of the axes and has zeros for the others, thus $(0, 1, 0)$ is a unit point for the y axis. At the moment, `unit_points` are evaluated according to their name: an 'I' or an 'X' in the name indicates $(1, 0, 0)$ (respectively, 'J'/'Y' for $(0, 1, 0)$ or 'K'/'Z' for $(0, 0, 1)$). In the future we hope to track and identify these points.

5.3.5 The Local-Cache Module

We have not previously mentioned the refined-type for the Saaz domain, `dmn`. There are two important cases, with one enabling much better optimizations. The basic, non-fully-refined type for `dmn` could be

$$\{\texttt{Domain} : \texttt{lwb} = \texttt{(URT,URT,URT)} \wedge \texttt{upb} = \texttt{(URT,URT,URT)}\}$$

In this case, the best optimization which can be done for `dmn.Min(0)` would be to replace it with `dmn.x_min`. This is the sort of inlining a general-purpose compiler can perform. Still, accessing the desired value from the heap requires an extra level of indirection: accessing first `dmn`, and then the member `x_min`.

However, we can do better if `dmn` does have a fully refined type, say, that of Equation 5.3:

$$\texttt{dmn} : \{\texttt{Domain} : \texttt{lwb} = \texttt{(x\_lo,y\_lo,z\_lo)} \wedge \texttt{upb} = \texttt{(x\_hi,y\_hi,z\_hi)}\}$$

The member access `dmn.Min(0)` can be replaced with the actual constructor parameter `x_lo` (assuming `x_lo` is not written in the interim, which it seldom is). The rewritten access is faster as it is on the stack and does not require an extra level of indirection. Furthermore, it is likely to be on the same cache line as other frequently-used values.

When the Type-Refinement module encounters domain constructors, it fills in the type refinement to be similar to Equation 5.3. However, in real applications, domains are often not created, but instead come from arrays which are loaded from files on disk. In these cases, the type of `dmn` will not be fully refined. The purpose of the *Local-Cache* module is to make `dmn` fully-refined. This is done through the introduction of new, stack-local variables. New variables are introduced after the assignment to the domain, either from a call to load an array from disk, when an array is created (for its domain member-variables), or when a domain is created from a constructor.

```
1 Array3 arr;
2 OpenArray("myarray.sa3", arr);
3 Domain3 dmn = arr.Domain();
```

At this point our refined types are:

$$\texttt{arr} \quad : \{\texttt{Array} : \texttt{layout} = \texttt{Unknown} \wedge \texttt{domain} = \texttt{dmn}\} \tag{5.6}$$

$$\texttt{dmn} \quad : \{\texttt{Domain} : \texttt{lwb} = \texttt{(URT,URT,URT)} \wedge \texttt{upb} = \texttt{(URT,URT,URT)}\} \tag{5.7}$$

By introducing new variables for the values of `dmn`:

```
4 int x_lo = dmn.x_min; // or dmn.Min(0)
5 int y_lo = dmn.y_min;
6 int z_lo = dmn.z_min;
7 int x_hi = dmn.x_max;
8 int y_hi = dmn.y_max;
9 int z_hi = dmn.z_max;
```

We can now update our refined types to:

$$\texttt{arr} \quad : \{\texttt{Array} : \texttt{layout} = \texttt{Unknown} \wedge \texttt{domain} = \texttt{dmn}\} \tag{5.8}$$

$$\texttt{dmn} \quad : \{\texttt{Domain} : \texttt{lwb} = \texttt{(x\_lo,y\_lo,z\_lo)} \wedge \texttt{upb} = \texttt{(x\_hi,y\_hi,z\_hi)}\} \tag{5.9}$$

We can now proceed to replace function calls to *getters* (data-member-accessor functions) with not just member accesses, but values from the stack: `dmn.Min(0)` becomes `dmn.x_lo`.

5.3.6 The Cleanup Module

Finally, the *Cleanup* module removes declarations for variables which are never referenced, saving setup time and reducing the memory footprint. While only a small amount of memory is saved, it occupies space in the cache. The duplicated variable space introduced by the inlining of Cascade and aggregator operations is thereby recovered. In addition, we may cache variables optimistically in the Local-Cache module without paying for them if they don't get used.

The Saaz iterators which the For-Loop module moved out of the for-loops were still created using calls to `Mk_Iterator`. As the other modules have replaced calls to `Collapse` and `Promote`, and the domain is covered by integers instead of iterators, the iterators are no longer necessary. The Cleanup module therefore removes the iterators and the `Mk_Iterator` calls.

5.4 Related Work

Specialization of code has been explored in the past. Dynamic compilation [CN96] inserts checks and branches at runtime to choose appropriate specialization paths. Value profiling does so at compile-time based on statistical observations of past program values [CFE99]. Aigner and Hölzle [AH96] use value profiling to remove virtual function calls. Tettnang's code generation does not require a retrospective record of past executions.

Calder [CG94] and Bacon [BS96] look at the ability of three static analysis techniques to identify virtual function calls. The most effective technique presented in the two papers is Rapid Type Analysis (RTA). RTA identifies which types are actually instantiated and resolves virtual function calls where there is no ambiguity. Because RTA looks at the existence of objects, it is unable to provide any optimizations for array layout in the case of the existence of heterogeneous layouts, even when a query uses only arrays with homogeneous layouts.

Figure 5.8 gives an example of a case which could not be handled by RTA, but can be by Tettnang. Arrays A, B, and C are all defined over the same domain, but array C has a different layout than A and B. The first loop, starting at Line

```
 1 Domain3 dmn;
 2 Array3 A = Array3::Create(dmn, Layouts::RowMajor);
 3 Array3 B = Array3::Create(dmn, Layouts::RowMajor);
 4 Array3 C = Array3::Create(dmn, Layouts::ColMajor);
 5 for (Iterator i = dmn.begin(); !i.end(); ++i)
 6 {
 7   A[i] = B[i];
 8 }
 9 /* ... */
10 for (Iterator i = dmn.begin(); !i.end(); ++i)
11 {
12   C[i] = A[i] + k*B[i];
13 }
```

Figure 5.8: RTA is unable to optimize the indexing operations in these loops.

7, uses just arrays A and B. The indexing operations in the first loop use a row-major layout. RTA looks at which objects are instantiated. Because both row-major and column-major objects are created, RTA cannot disambiguate between any given use. If C had been row-major, then the indexing operations could be all identified as row-major. Tettnang is able to optimize indexing operations in both the loop which uses homogeneous layouts (Line 7) and that which uses heterogeneous layouts (Line 12).

RTA is only able to work with virtual function calls, but Saaz uses function pointers. Saaz could be recoded to use virtual function calls, but RTA would still be insufficient. Tettnang uses detailed refinement types that allow it to not just inline, but consolidate and optimize index linearization operations. RTA cannot optimize across objects as Tettnang can; at best, RTA would be able to inline linearization operations. RTA would be unable to consolidate the index calculations because each calculation would be using a different domain variable (even though the domain variables may reference the same domain object).

In contrast to many other systems that focus on optimizing transformations, Tettnang focuses largely on removing library overheads. A library that similarly addressed such overheads was the A++/P++ library by Parsons and Quinlan [PQ94]. In previous work [QSYS06] [QSVY06], Quinlan et al. use an earlier version of the Rose framework to perform transformations. Unlike Tettnang, which has access to built-in information, these translators use annotations to identify certain library constructs and transform code at the algorithmic level, targeting higher-level operations than Tettnang, primarily converting whole-array

operations into nested loops and fusing loops. A more modern approach to avoiding the temporaries caused by whole-array operations is that of Expression Templates [Vel96] [VJ02]. In any case, the conversion of whole-array operations is not relevant to Saaz, which uses looping constructs instead of whole-array operations. Tettnang does not address loop transformations such as blocking, skewing, fusion, and fission [WL91]. The information Tettnang gathers would make such transformations straightforward, but such transformations are orthogonal to those of Tettnang. Many Tettnang transformations would still be applicable after these loop transformations, so we leave them to future work.

Telescoping languages [KBC+05] analyze libraries and construct a compiler to optimize them. The optimizations see most of their gains from adding static-types to dynamically-typed languages such as Matlab. The closest to our work is the Broadway compiler [GL00], which seeks to address a wide variety of domains and levels of abstraction through library annotation. As with Tettnang, inlining and rewrite transformations are library-specific, since trade-offs can be significant. Such optimizations can be conditioned on object properties established and modified through dataflow analysis. Broadway does not address call-sequences or accesses to object members; it only optimizes function calls. This makes it useful for Encapsulation overheads, but inappropriate for addressing many Isolation overheads. In contrast to Tettnang, Broadway cannot consolidate offset computations across multiple arrays, build some offsets from others, or build a cache of local variables on the stack. Such contextual and inter-expression optimizations provide a significant portion of Tettnang's performance improvements.

The power of Tettnang lies is its ability to resolve, at compile time, certain control flow decisions that drastically impact performance. Expression Templates [Vel96] [VJ02] have been able to address composition through aggressive inlining of template methods, most notably in the Boost::Phoenix library [dGMH]. In particular, the physics library OpenFoam [JJvT07] uses heavily-templated C++ to write CFD simulators. uBLAS [WK+] provides the functionality of BLAS [DCHH88a] using Expression Templates. uBLAS uses the type system to encode storage schema such as layout and sparsity. Optimizations within uBLAS are lim-

ited to the elimination of temporaries and some virtual function calls related to layout.

Template metaprogramming, however, remains a poor tool for addressing call-sequences or annotating objects with state. Template-based libraries are typically harder to write, harder to use, and much harder to debug than conventional libraries. Template error messages which are legalistically accurate may not provide information in a way that a domain scientist can understand (see Appendix D for a more thorough analysis). There are some promising techniques to improve error messages [LFGC07], but these are not widely implemented.[4] We consider our technique more intuitive and more user-friendly for domain scientists.

The TaskGraph library [Rus06] builds an execution graph at runtime which describes the computation. The library then generates plain C++ source code to execute it, compiles that source code, links in the generated binary, and executes the computation. This method of delayed evaluation incurs the overhead of generating the code and running a compiler, although caching of the generated source code or binary can mitigate this cost. Tettnang uses a pre-processing pass of the source code and acts as an additional compiler in the tool chain, but incurs no runtime overhead. TaskGraph explores four optimization methods: code caching, loop fusion, array contraction, and runtime liveness evaluation. The first and second are not relevant to Saaz or its translator, Tettnang. Array contraction and runtime liveness evaluation are complementary optimizations not addressed by Tettnang. Because Saaz uses loops instead of whole-array operations, the users tend to perform array contraction on their own. We leave liveness evaluation up to users.

There has been some work in delayed evaluation libraries such as Dryad [IBY+07] and FlumeJava [CCPA+10] which can handle not just nested calls, but also sequences of adjacent (not nested) calls. Such libraries address algorithmic optimizations in the context of distributed computing. Tettnang operates on much lower-level abstractions, and so its optimizations are complementary.

Many of the approaches we have discussed in this chapter perform optimiza-

[4]The Clang compiler has slightly better error messages, but they are still obtuse.

tions which Tettnang does not. They may optimize code structure or arrangement, such as the appearance of temporary arrays, nested calls, extra loops, or the ordering of loops. Such optimizations could be performed by Tettnang as they are purely complimentary to those Tettnang does perform, and could be a target for future work. The closest in spirit to Tettnang is the Broadway compiler. Broadway, like Tettnang, addresses the overheads of libraries, but Broadway is only able to optimize function call-sites. Tettnang, on the other hand, specializes itself to one library, and is able to perform much more thorough optimizations. A custom translator is able to perform domain-specific translation and optimize libraries in ways these other approaches cannot. One day, we may find a general translator which can perform similar optimizations, but for now, the domain-specific approach looks promising.

Chapter 6

Queries and Results

In this chapter we explore the performance implications of overheads introduced by Saaz. Compiler- and library-writers would like to know which overhead categories are most expensive. Unfortunately, many implementation decisions actually cause overheads in more than one of our categories (particularly in Encapsulation); thus, identifying the performance implications of individual categories is difficult. Instead, we look at the performance implications of Saaz's individual design decisions.

We measure the performance impact of design decisions by looking at the effectiveness of various transformations performed by Tettnang. Each transformation corresponds roughly to a single abstraction employed by Saaz. Because Saaz's abstractions are used to varying degrees in different queries, we measure the cost of each abstraction across different queries.

At the heart of Saaz is its support for multiple array layouts. This support hides several configuration details, and is thus quite expensive. One transformation which targets one of its overheads provides our most significant performance gains by inlining the offset calculations. To identify exactly what parts of Saaz's support for multiple array layouts were the most expensive, we look at a microbenchmark. This microbenchmark identifies the performance of intermediate levels of abstraction between Saaz's implementation of array indexing and that of plain C++. Finally, we compare the performance of our queries when compiled with three different compilers: `g++` versions 4.2 and 4.7 and `icc` version 12.0. We chose these

compilers because Rose requires `g++` version 4.2, 4.7 is the latest version of `g++`, and version 12 is the latest version of `icc`, another popular compiler for scientific codes.

Experimental Setup Our experiments look at the performance of several queries. These queries are implemented several ways: in Saaz, in C++ using a C-style 3D array, and in C++ using a linear array. Scientific codes use both, and their performance characteristics can vary depending on the computation, memory stride, and compiler. In very specific cases, general-purpose compilers are able to optimize simple uses of C-style 3D arrays. Linear arrays, on the other hand, can be used for either row-major or column-major layouts, and can therefore be compatible with either C or Fortran arrays. The C++ codes using linear arrays index each array with the appropriate row-major or column-major linearization calculation (implemented via macros). The result is duplicate computations in the source code, but these are optimized away by the general-purpose compiler, which recognizes them as duplicate expressions. Internally, Saaz uses linear arrays to store data.

Tettnang uses flags to control which transformations it performs. Because each transformation targets a specific abstraction used by Saaz, we can account for the performance implications of individual Saaz abstractions. We incrementally apply Tettnang transformations to our Saaz implementation.

Test results are the fastest of a minimum of six runs on a dual-socket machine with 2 Xeon E5-2670 (Sandy-Bridge extended) 8-core CPUs (16 cores total) running at 2.6 GHz with 192GB of 1600MHz DDR3 RAM and a single 2TB 7200 RPM hard disk. All tests use a single core. Programs were compiled using `gcc v4.7` and command line options `-O2`.

6.1 Queries

We selected eight queries (Table 6.1) in three sets to illustrate different performance trade-offs. Our first set (Figure 6.1) consists of four *point-wise correlations* (Section 3.3.3) and three *derivative-correlations* (Section 3.3.4). The

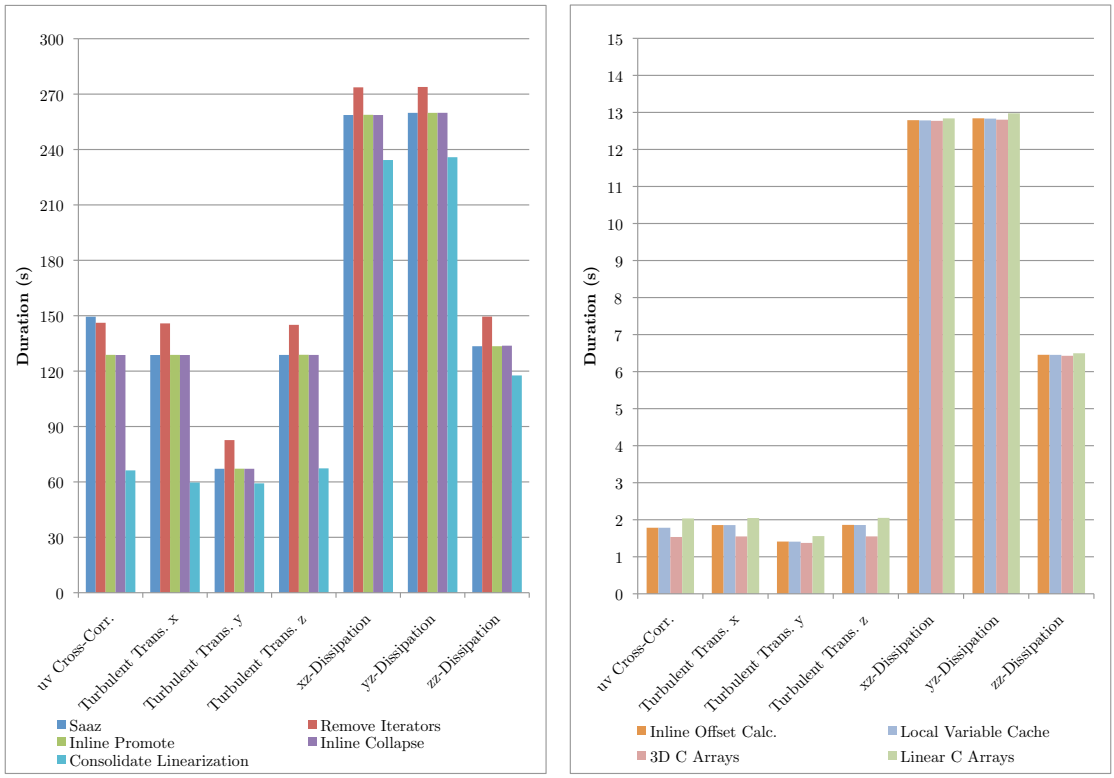
Table 6.1: Sample queries and their implementation. Note the difference in stride between the dissipation queries: some strides are along the x dimension (unit stride), others along y or z (stride > 1). u_avg, v_avg, and w_avg represent the planar average of the primitive variables u, v, and w, taken along the y axis. All queries include a planar average. The formulas show the result computed by the query for all points p in the domain of the arrays, before the final planar average is taken. $p.x$, $p.y$, and $p.z$ select the components of point p along the x, y, or z axes, respectively.

Description	Operation	Pseudo-code
UV Cross-Correlation	$\langle u'v' \rangle_y$	$uv[p] := (u[p] - u\_avg[p.y]) * (v[p] - v\_avg[p.y])$
Turbulent Transport x	$\langle u'u'v' \rangle_y$	$uuv[p] := (u[p] - u\_avg[p.y])^2 * (v[p] - v\_avg[p.y])$
Turbulent Transport y	$\langle v'v'v' \rangle_y$	$vvv[p] := (v[p] - v\_avg[p.y])^3$
Turbulent Transport z	$\langle w'w'v' \rangle_y$	$wwv[p] := (w[p] - w\_avg[p.y])^2 * (v[p] - v\_avg[p.y])$
xz-Dissipation	$\langle s_{13}s_{13} \rangle_y$	$xzdiss[p] := 0.25 * ($ $((w[p + \vec{X}] - w\_avg[p.y]) - (w[p - \vec{X}] - w\_avg[p.y]))^2$ $+((u[p + \vec{Z}] - u\_avg[p.y]) - (u[p - \vec{Z}] - u\_avg[p.y]))^2)$
yz-Dissipation	$\langle s_{23}s_{23} \rangle_y$	$yzdiss[p] := 0.25 * (((w[p + \vec{Y}] -$ $w\_avg[p.y + 1]) - (w[p - \vec{Y}] - w\_avg[p.y - 1]))^2 +$ $((v[p + \vec{Z}] - v\_avg[p.y]) - (v[p - \vec{Z}] - v\_avg[p.y]))^2)$
zz-Dissipation	$\langle s_{33}s_{33} \rangle_y$	$zzdiss[p] := 0.25 * (2 * ((w[p + \vec{Z}] - w\_avg[p.y]) -$ $(w[p - \vec{Z}] - w\_avg[p.y]))^2)$

second is our *eduction criteria*, λ_2 (Section 3.3.5). Our final set (Figure 6.3) is the *predicated* execution of the first set. By thresholding λ_2, we identify about 2.6% of the domain as part of coherent structures.

Input arrays u, v, and w contain $1536 \times 768 \times 768$ double-precision floating-point numbers and are laid out in column-major order (note that our previous discussions in Sections 3.3, 4.2, and 5.2 used row-major ordering as an example, but this distinction should not be relevant here).

We apply the transformations discussed in Section 5.3 incrementally, moving performance of the Saaz implementation closer and closer to a plain C++ implementation. As the code is transformed to use fewer and fewer of Saaz's abstractions, the penalties of those abstractions are eliminated, significantly improving performance. Without Tettnang, the Saaz implementation runs an average of 65 times slower than a plain C++ implementation. With Tettnang's transformations, the transformed Saaz implementation performs an average of 64 times better than the original Saaz code. Some transformed queries even perform 16% better than the best plain C++ implementation, but each does no worse than 83% of the plain C++ implementation.



(a): Baseline and first four transformations.

(b): Final transformations and target C++ implementations.

Figure 6.1: Performance of the various unconditional queries with incrementally more of Tettnang's transformations applied. The first bar in (a) is unmodified Saaz. The final two bars in (b) are our C++ implementations. Note the difference in scales between the two graphs. Inlining offset calculations was the most effective transformation, resulting in $\frac{1}{20}$ the duration, or a 20 times increase in performance.

6.1.1 Transformations

Our first transformation converts for-loops over iterators to nested for-loops over integers. By itself, it does not substantially improve performance. In fact, as Figures 6.1a and 6.3a show, it can significantly decrease performance. This is because, by itself, this transformation does change the loops, but it also must reconstruct the iterator object itself, since other operations require it (Figure 6.2 Lines 3 and 12). Note that this reconstruction is generated by Tettnang, and need never be written by a user. Reconstructing the iterator adds overhead. This transformation does remove function-call overhead from Encapsulation, but most importantly, Isolation and Generalization overheads. This provides more opportunities for later transformations which do improve performance. In fact, as later

```
1 for (int y = dom.y_min; y != dom.y_max + 1; ++y)
2 {
3   Iterator1 i = Mk_Iterator(dom,1U,y);
4   Point1 q = i.Collapse();
5   Point1 q(y);
6   /* ... */
7   Domain2 dom2 = i.Slice();
8   for (int x = dom2.y_min; x != dom2.y_max + 1; ++x)
9   {
10    for (int z = dom2.x_min; z != dom2.x_max + 1; ++z)
11    {
12      Iterator j = Mk_Iterator(dom2, 1,0, x,z);
13      Point3 p = i.Promote(j);
14      Point3 p(x,y,z);
15      /* ... */
16    }
17  }
18  /* ... */
19 }
```

Figure 6.2: Converting for-loops over iterators to for-loops over integers requires that the iterators be reconstructed (Lines 3 and 12) so they can be used in subsequent computations (Lines 4, 7, and 13). When Tettnang knows which axes are being traversed by the iterators, it can rewrite the `Collapse` and `Promote` operations on them. Thus Tettnang rewrites Line 4 to Line 5 and Line 13 to Line 14. Because the inner-loop's iterator, `j` is not used in subsequent operations (`q` and `p` are used for indexing), it can be removed (Line 12).

transformations avoid the need to use the iterator object, it can be removed.

Our second and third transformations inline the point promotion and point collapsing operations. These operations do not have the most expensive overheads, but we do want to address all overheads. For all but our λ_2 query, the innermost (2D) iterator is used only in the promotion of the outer (1D) iterator (Figure 6.2 Line 13). When we inline the point-promotion operation, we no longer need the innermost iterator, and can avoid creating it. This eliminates most of the overhead which was introduced as a side-effect of transforming the for-loops over iterators. See Figure 6.2.

Our fourth transformation is similar to partial redundancy elimination [MR79]. Consolidation of linearization calculations reduces the number of calls to `LinearIdx`. Three out of four of our point-wise correlations use two arrays indexed at the same point. For the unconditioned queries, consolidation cuts running time nearly in half (Figure 6.1a). The remaining correlation, Turbulent Transport y, only accesses a single array, so there is no redundancy to eliminate: the only improvement it gets is from saving a function call. After the linearization calculations have been consolidated, the running times between the four point-wise correlation queries are nearly identical. Furthermore, the conditioned queries for

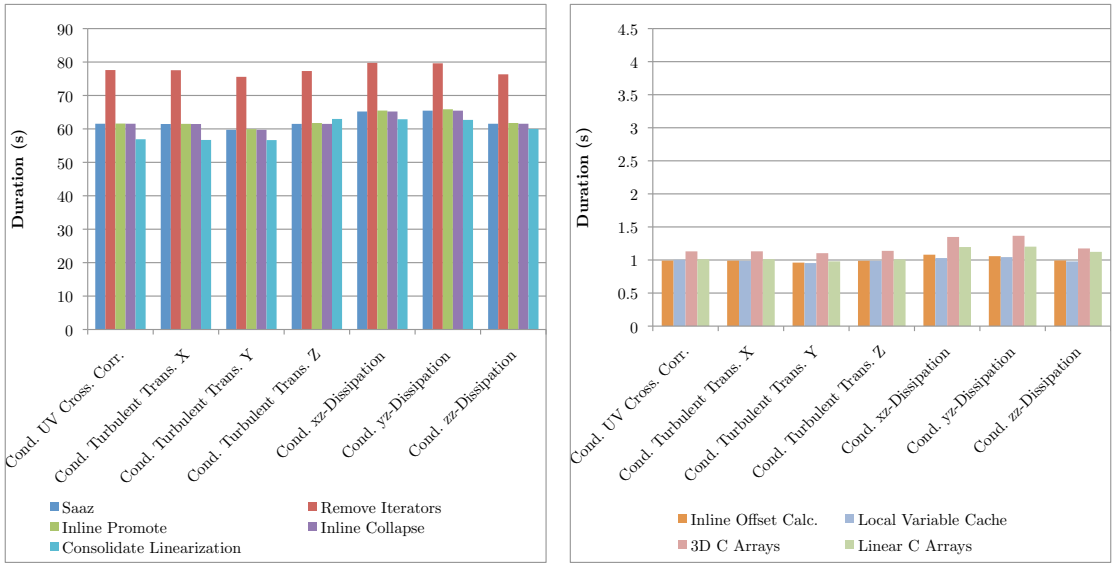
each query implementation have relatively uniform running times (Figure 6.3). From this we can infer that the overwhelming cost of these operations stems from the linearization calls.

Our derivative correlation queries do not benefit from the consolidation of linearization operations because they are stencil operations: they do not reuse a single point, but instead access adjacent points in three dimensions. These points share a common index with offsets of the form $p \pm (a, b, c)$. Stencil operations are addressed by a second indexing optimization: inlining the linearization calculations (recall Section 5.3.4). As the difference in axes between the two graphs in each of Figures 6.1 and 6.3 shows, inlining the linearization calculation significantly increases performance. Performance for our queries (except for λ_2) increases by twenty times. λ_2 does not improve as much because its mathematical operations consume a much higher proportion of its runtime than the overheads introduced by Saaz. Because the inlining of linearization calculations provides such a dramatic shift in performance, Figure 6.5 in Section 6.2 examines this improvement at a finer granularity.

Tettnang's final transformation is the local-variable cache. This transformation uses stack-local variables to hold copies of data from the heap. By accessing variables on the stack instead of variables on the heap, the compiler is more likely to place the variables into registers. As we can see, this transformation provides little improvement for `g++` version 4.7. The greater effect of this transformation for other compilers is explored in Section 6.3.

6.1.2 Query Comparisons

Comparing our three sets of queries offers some insight as to the cost of Saaz overheads. The non-conditioned, point-wise correlation queries spend much of their time accessing memory. They perform few floating-point operations (3 FLOPS in the innermost loop), and so the cost of indexing is significant. When these queries are masked via our eduction criteria (λ_2), they spend significantly more time on indexing and accessing memory. The consolidation of indices and inlining of point-promotion and point-collapsing is insignificant when the vast majority of



(a): Baseline and first four transformations.

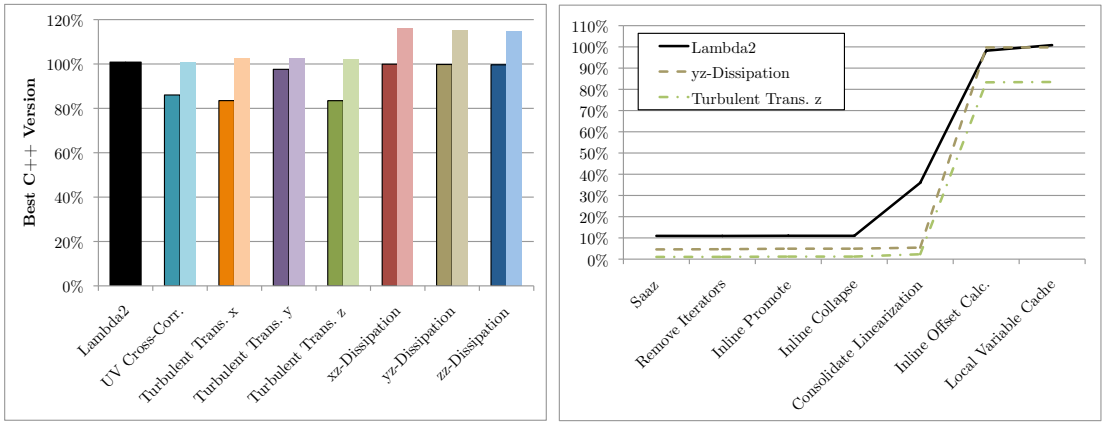
(b): Final transformations and target C++ implementations.

Figure 6.3: Performance of the various conditioned queries with incrementally more of Tettnang's transformations applied. The first bar in (a) is unmodified Saaz. The final two bars in (b) are our C++ implementations. Note the difference in scales between the two graphs. Again, inlining offset calculations was the most effective transformation, resulting in $\frac{1}{20}$ the duration, or a 20 times increase in performance.

points do not perform these operations. The local-variable cache is more helpful in the conditioned operations because the conditioned queries spend relatively more time on the linearization calculation than retrieval or computation using the input arrays.

When executed conditionally, the derivative correlations behave very similarly to the point-wise correlations. This is because the vast majority of time is spent indexing and thresholding the λ_2 array. This cost overwhelms the extra computation which is performed by the derivative correlations. In both unconditional and conditional derivative correlation sets, there are slight performance variations between the xz-, yz-, and zz- Dissipation queries. This is because the different derivative directions cause a difference in stride. Furthermore, zz-Dissipation only accesses one array.

Internally, each Saaz array uses a linear C-style array to store its data. There are two differences between the query's implementations, as fully-transformed by Tettnang, and the query's plain C++ implementation with a linear



(a): All queries.

(b): Progressive transformations.

Figure 6.4: The overhead penalties after all of Tettnang's transformations for all, and three representative queries. 100% corresponds to no overhead, compared to the faster of our two C++ implementations for that particular query. Secondary, lighter bars are conditional queries.

array. The first is relatively minor: array data is accessed through a data-member pointer of the Saaz array object (`A.data` vs. `A_data`). The second is more significant: linearization calculations are consolidated. The plain C++ linear array implementation performs indexing via macros which expand to the linearization calculation. For the point-wise correlation queries, the compiler is able to recognize common subexpressions between these linearization operations, which are evaluated at the same point. For the derivative correlation queries, however, the case is different. While the compiler may recognize some common subexpressions, Tettnang does more than eliminate these subexpressions. Tettnang optimizes stencil calculations by calculating offsets from the center point as deviations from its linearized value. This gives Tettnang an opportunity to improve performance over the plain C++ implementation which uses linear arrays. Indeed, the conditioned queries achieve higher performance than their plain C++ counterparts. See Figure 6.4.

For unconditioned point-wise queries, the plain C++ version using C-style 3D arrays is 12%-25% faster than the one using linear arrays (Figure 6.1b). C-style 3D arrays include more accessible semantic information which the general-purpose compiler is able to use to optimize simple patterns. When the loops have a conditional check, however, these built-in patterns cannot be matched, and so

the general-purpose compiler cannot optimize them. These patterns are built-in and typically vary across compiler versions. For the conditional case, the C-style 3D arrays perform 11% more poorly than the linear arrays.

6.1.3 Summary

Tettnang is able to significantly reduce the overheads of Saaz. Saaz queries are almost completely rewritten, eliminating the vast majority of overheads, and even offering better performance than plain C++ implementations. In fact, most of the queries translated by Tettnang perform better than their plain C++ counterparts. In all cases, the Tettnang-translated queries perform better than the plain C++ version which uses linear arrays (the same format used by Saaz). Because the general-purpose compiler is able to optimize the use of C-style 3D arrays under special circumstances, our unconditional point-wise correlations perform slightly worse by comparison. Still, these are our worse-performing queries, and they still achieve at least 83% of this optimized performance. This represents an 80-fold improvement over plain Saaz.

The point-wise correlation queries use either one or two arrays (of `u`, `v`, and `w`), all indexed at corresponding (identical) points (e.g. `u[p] * v[p]`). Consolidating indices is thus a very important optimization for these queries. The derivative correlation queries are dissipation components and access adjacent points in a stencil operation (e.g. `v[p+PZ] - v[p-PZ]`) to compute a finite-difference derivative. Different memory behavior can be seen here, as different dimensions will each exhibit a different stride. As discussed previously (Section 5.2), the type refinements that Tettnang uses to track data schema allow Tettnang to inline the offset calculations. This information is also used to combine the linearization computations of stencils, and construct each as an offset from another. The transformation which inlines the linearization computations improves performance between 3 and 64 times across the queries.

Tettnang is able to achieve this dramatic improvement by using semantic knowledge of Saaz. By understanding the semantics of Saaz objects, Tettnang is able to track key properties and use that information at call-sites. Tettnang's

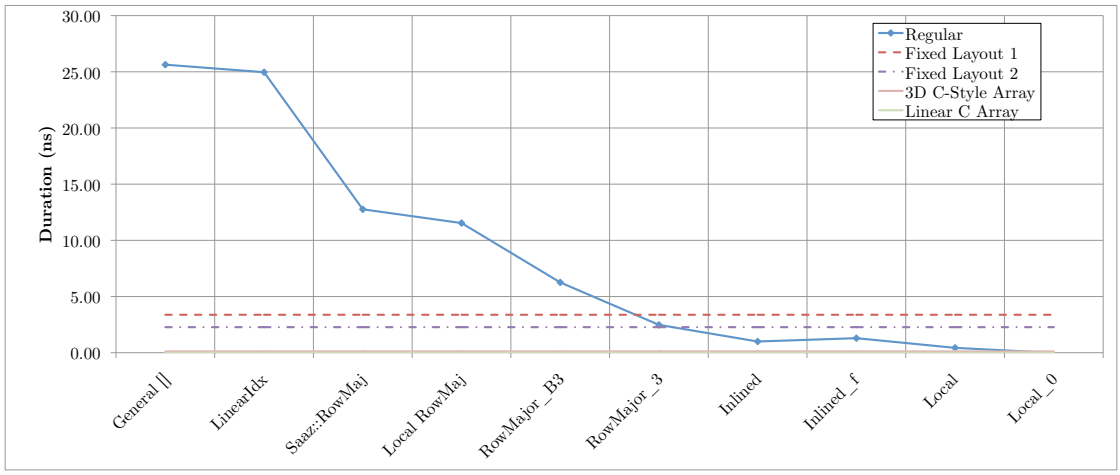


Figure 6.5: Performance of intermediate stages in the inlining of an indexing operation.

refined types track data schemas, allowing Tettnang to, in essence, *inline through a function pointer*. Tettnang does not have to prove that the function pointer does not change. Because array data schema are immutable, Tettnang *knows* that the function pointer *cannot* change.

Tettnang's translation renders performance competitive with hand coding, demonstrating the power of treating the Saaz abstractions as primitives in an Embedded Domain-Specific Language. Tettnang translates Saaz code to improve performance to as much as 84 times faster than the original Saaz implementation and 16% faster than the fastest plain C++ implementation.

6.2 Microbenchmarks

We showed in Section 6.1 that the most expensive operation in our queries was the linearization of multidimensional points. When Tettnang can determine an array's layout, it rewrites indexing operations, replacing the function pointer that linearizes a point with an equivalent, but less costly expression. To examine exactly where all the overhead from this operation originates, we measured the overhead of several alternate implementations of the indexing operation. These implementations are progressively more specialized, incorporating increasingly more of the problem's data schema.

The most expensive abstraction in Saaz is that of hiding array layouts. It is possible that Saaz would have much higher performance and not need Tettnang were Saaz to fix the layout of arrays and not support both row-major and column-major arrays. To investigate, we also compare the performance of two alternative Saaz implementations where array layout is fixed. In both of these alternative Saaz implementations, all arrays are row-major, so their indexing operations do not require a function pointer to perform linearization. In the first version, the indexing operator (`operator[]`) is the same as normal Saaz, only it calls a statically-defined function to perform the linearization calculation (`Elements(RowMajor_3(p))` for point `p`). In the second version, the indexing operator inlines the data access and inlines the row-major computation. The body of this function is the same as that generated by Tettnang before the local-variable cache transformation and is the same as *Inlined_f*. If the compiler were to inline both the `Elements` and the `RowMajor_3` calls in the first version's indexing operator, it would be identical to the indexing operator of the second version.

To put these timings in perspective, we also compare performance to our two plain C++ implementations, one using a C-style 3D array, the other using a linear C-style array. The performance of the code after all transformations have been applied is greater than the alternative Saaz implementations (and comparable to the plain C++ implementations). Our experiments have enabled us to isolate the effect of supporting multiple array layouts, enabling us to show that this important interoperability feature does not give rise to all the overheads of the indexing operations of Saaz.

Our experiment measures the overhead of a loop over all the points in a $512 \times 512 \times 512$[1] array of double-precision floating-point numbers. The duration of this loop is subtracted from the durations of each rewritten loop. This resulting time is divided by the number of elements, to give the cost of the different implementations. The loop body indexes the array at the current point, and assigns a value to it:

[1]Note that this is a different size than our previous experiments in Section 6.1. This should not be concerning because our timings here are the times for a single loop iteration, and all accesses have unit-stride.

```
1   Domain3 dmn(0, 0, 0, 512, 512, 512);
2   Array3 arr(dmn, Layouts::RowMajor);
3   double count = 0;
4   for(Iterator3 p = dmn.begin(); p != dmn.end(); ++p)
5   {
6
7     arr[p] = count++;
8   }
```

Each intermediate transformation will rewrite the indexing operation (`arr[p]`) to further specialize it.

LinearIdx inlines the bracket-operator (`operator[]`) function call, breaking out the calls to `Elements` and `LinearIdx`. Our loop body is now:

```
6   unsigned long long idx = a.LinearIdx(i);
7   a.Elements(idx) = count++;
```

Saaz::RowMajor calls the row-major linearizing function directly, instead of using the `LinearIdx` function pointer. From here on, `idx` remains an `unsigned long long` and each fragment includes `a.Elements(idx) = count++;`, but we omit it from our code examples for the sake of clarity.

```
6   idx = Saaz::RowMajor(i, dom.Min(), dom.Max());
```

Local RowMajor moves the row-major linearization function into the global namespace. These functions are identical, but have different names. Since the remaining functions exist in the global namespace, this is an intermediate step.

```
6   idx = ::RowMajor(i, dom.Min(), dom.Max());
```

RowMajor_B3 specializes the `RowMajor` function to three dimensions. The generic `RowMajor` function is a function template which can handle an arbitrary number of dimensions. `RowMajor_B3` keeps bounds-checks present in the previous implementations and returns the result of the linearization expression.

```
6   idx = RowMajor_B3(dom, i);
```

RowMajor_3 is the same as `RowMajor_B3`, but does not perform bounds-checking.

```
6   idx = RowMajor_3(dom, i);
```

Inlined inlines `RowMajor_3`, replacing the function call with its return expression. This removes the cost of a function call. As we can see, at this point, performance is better than that of both the fixed-layout versions. The general-purpose compiler does not inline the function call here, and Tettnang does.

```
6   idx = (i.z-dom.Min(2)) + ((i.y-dom.Min(1)) + (i.x-dom.Min(0)) *
        (dom.Max(2)-dom.Min(2)+1)) * (dom.Max(1)-dom.Min(1)+1);
```

Inlined_f Inlines the function calls to access the data-members of the domain directly, replacing `dom.Min(0)` with `dom.x_min`. Because performance is more or less identical to the previous version, we suspect the domain member access functions were already being inlined by the compiler.

```
6   idx = (i.z-dom.z_min) + ((i.y-dom.y_min) + (i.x-dom.x_min) *
        (dom.y_max - dom.y_min + 1))  * (dom.z_max - dom.z_min + 1);
```

Local uses a local variable instead of accessing the bounds from the domain. This is as far as Tettnang can usually optimize.

```
6   idx = (i.z-x_min) + ((i.y-y_min) + (i.x-x_min) *
        (y_max - y_min + 1)) * (z_max - z_min + 1);
```

Local_0 removes the subtractions where the domain's minimum bounds are zero. Note that Tettnang is able to determine this only when it sees calls to the domain constructor, and not when the domain is loaded from disk as part of an array.

```
6   idx = i.z + (i.y + i.x * (y_max + 1)) * (z_max + 1);
```

The most expensive part of indexing is the function pointer. Because Tettnang can identify what this function pointer may resolve to, it is able to recover from this expense. This information also lets Tettnang inline the function. Extra information lets it optimize even more, although the fact that a domain's lower-bounds are zero is typically very hard to establish. While the function pointer is the most expensive component of linearizing a point, even after eliminating the function pointer, we can still improve performance further by replacing it with the equivalent expression and using stack-local variables. How much this will improve will depend on the ability of the compiler to allocate registers. In this test of `g++` version 4.7, performance improved by 0.63 ns per access.

6.3 Compiler Comparison

Our previous results were obtained with code generated by the GNU C++ compiler, `g++`. `g++`, however, is not the only compiler used by the scientific computing community. Intel's `icc` plays a significant role as well. Figure 6.6 shows all our previous benchmarks as compiled with three different compilers: `g++` versions 4.2.4 (`g++-4.2`) and 4.7.1 (`g++-4.7`), and `icc` version 12.0.2 (`icc-12.0`).

While the Intel compiler performs similarly to GNU for the plain C++ tests (C-style 3D Arrays and Linear C Arrays), it does much worse for the Tettnang-translated Saaz code (an average of 36% worse). The only exception is the λ_2 ("Lambda2") query. The Intel compiler is strongly tuned for Intel processors, which we use in our tests. For strongly arithmetic codes, such as λ_2, `icc` is able to generate highly-tuned assembly, making heavy use of instruction pipelining.

For most of the queries, `icc` seems to perform the same or worse with C-style 3D arrays than it does with linear arrays. There are two explanations for this: either `icc` does not have the types of pattern-recognition for multidimensional C-style arrays which are present in `g++`, or `icc` is able to better optimize linear arrays. Because queries implemented with linear arrays perform similarly whether compiled with `g++` or `icc`, we suspect that `icc` lacks the pattern-recognition necessary to optimize the uses of 3D arrays which appear in our queries.

We include `g++-4.2` because its performance differs greatly from `g++-4.7` for the local-variable cache transformation. `g++-4.7` has almost identical performance before and after the local-variable cache transformation, while this transformation will, on average, almost double the performance of `g++-4.2`. We believe this is because `g++-4.7`, which is a more current compiler, is able to more effectively allocate registers. The local-variable cache transformation is meant to encourage the compiler to put data from the heap into registers. While the improvements to `g++` have made this transformation unnecessary for it, the transformation is still helpful for `g++-4.2` as well as the latest version of the Intel compiler, which sees similar improvements to `g++-4.2`.

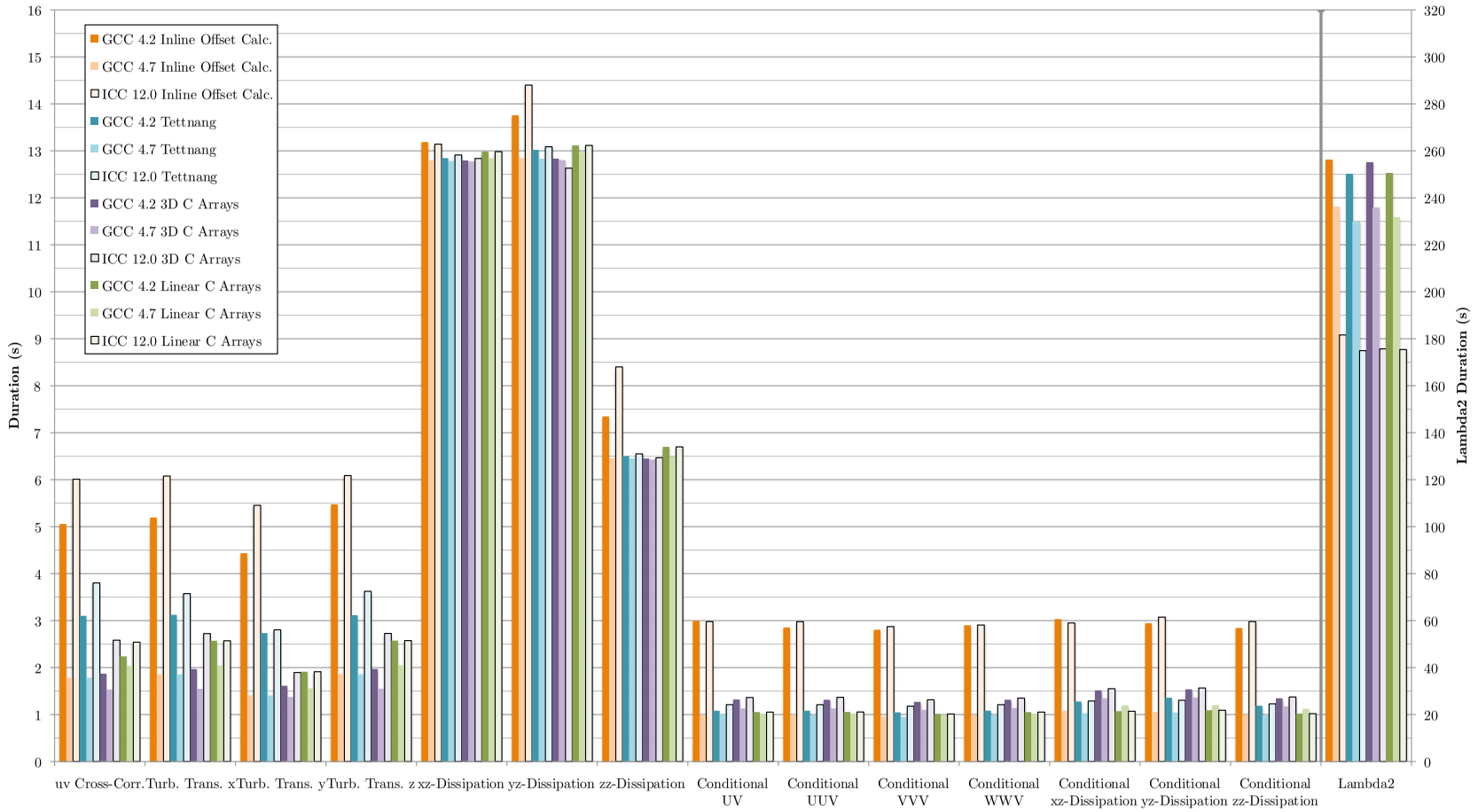


Figure 6.6: A comparison of three compilers: `g++-4.2` (dark), `g++-4.7` (medium), and `icc-12.0` (light, outlined). We look at performance for the last two Tettnang transformations: inlining of offset calculations (orange), and the local-variable cache (teal); as well as our two plain C++ implementations: 3D C-style arrays (purple), and Linear C arrays (green). λ_2 is the longest-running query, so it uses a separate axis, on the right.

Chapter 7

Conclusion

7.1 Limitations and Future Work

In this section, we present limitations of the ideas and implementations of Tettnang and Saaz. We propose extensions to address these limitations, as well as future directions.

There are certain characteristics of CFD codes that simplify the design of Tettnang. Operational masks for conditional execution are made more clear through the use of Saaz predicates. By moving the masking operations into the library, their semantics are made more clear to Tettnang.

Currently Tettnang does not give diagnostic warnings or domain-specific error messages, but gathers enough information that it could provide some useful feedback to the user. For example, Tettnang gathers information about both array sizes and loop behavior. Often this is enough information to perform compile-time bounds checking of loops and array accesses. We leave as future work the incorporation of translator-time error messages for loop bounds which will lead to accesses outside array domains.

Saaz allows users to specify the data schema for arrays. This capability is needed to support interoperability of analysis tools with different simulators. This flexibility, however, has inherent costs. When data schema cannot be identified, or when arrays are loaded from disk, more extreme measures must be taken to determine object properties. Figure 7.1 shows some possibilities. We could add

```
1 /* run with option
2  *   --tettnang:assume-arrays-rowmajor */
3 int main()
4 {
5   Array3 arr;
6   OpenArray("array.sa3", arr);
7   /* Tasks */
8 }
```

(i): Tettnang could be passed command-line options containing layout information.

```
1 int main()
2 {
3   Array3 arr;
4   OpenArray("array.sa3", arr);
5   #pragma array arr has layout row-major
6   /* Tasks */
7 }
```

(ii): Tettnang could read pragmas to get information about layout.

```
1 int main()
2 {
3   Array3 arr;
4   OpenArray("array.sa3", arr);
5   assert(arr.Layout()==Layouts::RowMajor);
6   /* Tasks, optimized by Tettnang */
7 }
```

(a): Tettnang could insert an assertion of its assumptions.

```
1 int main()
2 {
3   Array3 arr;
4   OpenArray("array.sa3", arr);
5   if (arr.Layout()==Layouts::RowMajor)
6   {
7     /* Tasks, optimized by Tettnang */
8   }
9   else
10  {
11    /* Tasks, as before */
12  }
13 }
```

(b): Tettnang could branch if its assumption is correct, or keep the unmodified, original Saaz code.

```
1 int main()
2 {
3   Array3 arr;
4   OpenArray("array.sa3", arr);
5   if (arr.Layout() != Layouts::RowMajor)
6   {
7     Transpose(arr, Layouts::RowMajor);
8   }
9   /* Tasks, optimized by Tettnang */
10 }
```

(c): Tettnang could guarantee its assumption is correct by performing the transpose if it is not.

Figure 7.1: Possible ways of telling Tettnang that an array is row-major (i, ii), and methods of handling not knowing (a, b, c).

flags (Figure 7.1i) or pragmas (Figure 7.1ii) to Tettnang to enable the programmer to make assertions about array properties, for example. Another possibility would be to have Tettnang insert verification checks into the code it specializes. These checks could: abort the code upon failure (Figure 7.1a), run a different (perhaps unmodified) version of the code (Figure 7.1b), or transform the data into the form for which Tettnang has specialized the code (Figure 7.1c). We leave these extensions as future work.

In cases where arrays are passed as arguments, well-established techniques such as interprocedural analysis and value tracking can be used. The properties

relevant to data schema (an array's layout and domain) are immutable properties. This simplifies inter-procedural analysis: the properties need not be tracked, but identified only once. Tettnang currently does not perform any interprocedural analysis. Instead, it inlines certain functions and makes use of built-in knowledge of Saaz functions. Aggregator functions are inlined to simplify analysis, but values passed into other functions are not used in the analysis of function bodies. Such interprocedural analysis exists in other systems, and including it in Tettnang would make Tettnang more effective in practice. Such analysis does not detract from the effectiveness of Tettnang's transformations in its absence. Tettnang and Saaz together form an Embedded Domain-Specific Language that extends the primitive data-types of its host language. They extend arrays to support new operations (multidimensional arrays with row-major and column-major indexing) with little to no performance costs.

We would also like to extend Saaz to support other kinds of storage organization. While the interface presented to Saaz queries is unlikely to change, support for a technique such as Adaptive Mesh Refinement (AMR) would require changes to both Saaz and Tettnang. Support for sparse domains, and potentially sparse arrays would also be useful. Where user-codes iterate over domains and access domain size (such as in planar averages), such a change could be transparent. We would also like to support different kind of linearization operations beyond different layouts. For example, support for periodic boundary conditions would require that some points outside of an array's domain can be mapped into it. This would require that Tettnang incorporate more layout types than just row-major and column-major.

One particular area that could be improved is the handling of unit points (Section 5.3.4). Unit points are used in point-valued expressions to access nearby points. For example, A`[p + PI]` accesses the value at A`[p.x+1,p.y,p.z]`. At the moment, Tettnang assumes that the point `PI` is $(1, 0, 0)$, based on its name. Tettnang could incorporate some analysis to verify that this is the case, or at least a runtime check to ensure that it is.

Perhaps the biggest limitation of Tettnang is that it is specialized to Saaz

alone. The techniques employed by Tettnang address overheads from all three of the categories that we identified (*Encapsulation*, *Isolation*, and *Generalization*). We have demonstrated that these overheads appear in Saaz in various forms, but we have not given examples from other libraries. Future work would examine how overheads from our categories appear in other libraries and how they might be overcome using the techniques that we have introduced in Tettnang. A good place to start may be numerics libraries such as TooN [Ros] or analytics libraries such as PAT from the Large Hadron Collider [FJHL08], each of which appears to also exhibit these overheads. Perhaps a limited set of annotations (along the lines of Broadway [GL00]) could be added to a library to communicate what information to track (or how to do so). A schema might be able to describe transformations based on this information. In this way, Tettnang could be extended to handle more libraries in a more general fashion.

The development of a general-purpose description of overheads (or possible optimizations) may make it possible to perform the sorts of optimizations Tettnang does, but at a lower-level than that of source code. If we can use lower-level information, then perhaps a system such as LLVM [LA04], which includes significant analytical capabilities, could be used to optimize code even further.

7.2 Conclusion

We have introduced the Saaz library to address difficulties faced by CFD scientists. Saaz uses an array-based data-model to abstract data organization. This allows analysis codes to be more interoperable, thus reducing the time and effort necessary to construct new models and physical understanding.

Like most libraries, however, Saaz introduces overheads. We have classified these overheads into three categories: *Encapsulation*, *Isolation*, and *Generalization*. Object-oriented languages are especially prone to *Isolation* overheads, requiring sophisticated static reasoning to identify runtime behavior. *Isolation* overheads may stem from *Generalized* operations and, in turn increase *Encapsulation* overheads. Saaz supports ad hoc queries, and as a consequence, is only able to provide abstrac-

tions of data. We have shown how the overheads Saaz introduces are significant, and presented a technique to remove them.

We address the overheads in Saaz with a customized compiler. The incorporation of library semantics into the compiler's reasoning allows it to gain more knowledge about a program and thereby address the overheads that the library had introduced. Our compiler analyzes codes written with the Saaz library. By understanding the data schema Saaz is abstracting, Tettnang is able to transform general Saaz calls into specialized plain C++ implementations.

A series of type-refinements are made to track these schemas of Saaz array objects. Various modules in Tettnang are then able to apply individual localized transformations that could not have been performed by a general-purpose compiler. Saaz's use of immutable data-members to manage data schemas greatly simplifies Tettnang's analysis. Future libraries and languages could benefit from using immutability more frequently, perhaps as a default. As we have shown with Saaz and Tettnang, a few immutable properties can convey enough information to allow a domain-specific compiler to reason about object behavior. This reasoning can lead to the execution of significant optimizations.

Previous efforts to tackle library overheads have been unable to overcome all the types of overheads that a library such as Saaz introduces. These prior efforts have been general-purpose and not domain-specific. In the future it may be possible to provide a generalized framework that would allow, for any sufficiently-annotated library, the sort of optimizations that Tettnang performs. Before committing to a generalized form, however, it would be prudent to investigate the effectiveness of our specialized approach in other domains. For now, we have shown how a domain-specific translator (Tettnang) can optimize the Saaz scientific library to not just overcome library overheads, but in some cases actually improve performance relative to plain C++ implementations.

Arrays are an important datatype in the scientific computing community. There is great potential in the ability to extend array types to support new operations and behavior. We have shown how to efficiently incorporate support for disparate existing behaviors (row-major and column-major layouts). This paves

the way for incorporating new behaviors, such as boundary conditions, into array types.

Appendix A

Saaz Implementation

We have previously presented a summary of the Saaz classes and interface (Chapter 3). Up until now, we have used simplified type names to make examples more concise (e.g. `Domain3` instead of `Domain<3>`). This chapter delves into a more detailed explanation of Saaz classes without this simplification. Appendix C uses the exact Saaz syntax introduced here.

All datatypes live in the `Saaz` namespace. We break the types in Saaz into two groups. The first basic datatypes are simple types and objects that hold data, but don't have meaningful members besides accessors. Second, are the Saaz classes. These hold more complex data and influence control flow.

A.1 Basic Datatypes

coord_t

```
typedef int coord_t;
```

Whenever Saaz needs to use a datatype which represents a member of an index space, it uses `coord_t`. That is, domains are cross-products of ranges which are of type `coord_t`. This is currently a typedef to an `int`, but could just as easily be a `float` if it becomes necessary to change the coordinate system in the future. Index linearization operations would of course need to make sure they arrived at appropriate integer locations in memory.

dims_t

```
typedef unsigned int dims_t;
```

Whenever Saaz needs to use a datatype which represents a dimension it uses a `dims_t`. Axes in Saaz are numbered starting with 0.

Point

```
template <dims_t num_dims>
class Point;
```

Points in Saaz are simple structures which are passed by value. For dimensions 1, 2, and 3, the components are of type `coord_t` and are named `x`, `y`, and `z`. These components are union'd with an array, so they can be accessed using a numerical offset, as well as lower-dimensional points. For example `Point<2>` is declared as follows:

```
template < >
class Point<2>
{
public:
  union
  {
    struct
    {
      union
      {
        coord_t x;
        Point<1> X;
      };
      coord_t y;
    };
    coord_t lst[2];
  };
};
```

The use of `lst` makes it easy to access a component of a point given a particular axis. For example, given a point `p`, `p.lst[1]` will select the `y` component of `p` and be equivalent to `p.y`. This makes it easier to do coordinate transformations: `p.lst[Y_AXIS]` can logically represent `p.y`, even if `Y_AXIS` doesn't happen to map to 1 in this particular case.

As a consequence of the union-structure of points, they cannot have direct constructors. Instead, `Point` objects are constructed with a call to the overloaded `P` functions, which creates a point with as many dimensions as it has arguments.

Pointers

```
template <typename Destination>
struct Ptr;
```

`Ptr` is a pointer wrapper for the Saaz classes: `Domain` and `Array`. Because domains and arrays live on the heap, they are handled primarily through pointers. Regular C-style pointers already have a defined definition of the `operator[]` method which cannot be overridden. To support indexing of arrays using the `operator[]` method, we use a specialized `Ptr` class to forward all functions to the underlying `Array` object. As a consequence, instead of using `Array` or `Domain` objects directly, users must use

```
Ptr< Array<Domain<num_dims>, Element_t> >
```

and

```
Ptr< Domain<num_dims> >
```

These forwarding function increase Encapsulation overhead, and makes alias analysis much harder for the compiler, increasing Isolation overheads. Our analyses regarding compiler abilities used simplified examples which did not have this additional complexity, but used plain C pointers with Saaz arrays or other objects (i.e. `Array<Domain_t, Element_t>*`, `Person*`).

The `Ptr` class is intended to be specialized. We use manual partial-instantiation within the Saaz library for all the necessary classes.

A.2 Classes

Domains

```
template <dims_t num_dims>
class Domain;
```

Domains use a factory method called `Create` to obtain a new `Domain` object (stored in a `Ptr` to a `Domain` object). The rank of a domain is available via its class's static value `NumDims`.

Domain objects are immutable. Unlike other systems such as Matlab and Fortran, domains in Saaz do not support the `reshape` operation. A domain's extents are immutable and as specified by parameters to the `Create` call which constructs it.

Arrays

```
template <typename Domain_t, typename Element_t>
class Array;
```

Unlike many languages and libraries, which treat array-layout as a convention, Saaz allows arrays to have either a row-major or a column-major layout. When languages like Matlab and Fortran choose the column-major order layout, they eliminate a check and simplify the computations which must be done to index an array. In Saaz, array layout is immutable; once an array is constructed, its layout never changes. An array's domain is also immutable, preventing its index set from changing.

The template arguments to an array specify what type of domain the array is defined over (`Domain_t`), as well as what type of data it will store (`Element_t`). The only requirement for the element type is that it be default- and copy-constructable. Because arrays take a domain as an argument, they could, in the future, be defined over sparse domains without changing their syntax.

Arrays use a factory method called `Create` to obtain a new `Array` object (stored in a `Ptr` to an `Array` object). An array's domain type, storage element type, and number of dimensions are available in the `Domain_t`, `Element_t`, and `NumDims` static members of the `Array` type.

Iterators

```
template <dims_t total_dims>
template <dims_t out_dims>
class Domain<total_dims>::Iterator<out_dims>;
```

Iterators are member types of the domains they iterate over, but they all implement the same concept, which is similar to that of `std::iterator`. See Figure A.1. The iterator must define its type as `Self_t` (Line 3), and the type of the domain over which it is iterating as `Domain_t` (Line 4). Since iterators may iterate over fewer dimensions than the domain which called `begin` (which we will call the *embedding domain*), they keep track of what points they can cover. `Base` is the point in the embedding domain that the point is covering (Line 5). The iterator may be collapsed (Line 10) to obtain a point for the axes being covered (a point in

```
1  concept Iterator<dims_t out_dims, typename EmbeddingDomain>
2  {
3    typename Self_t; // the type of the iterator
4    typename Domain_t; // domain being traversed, an EmbeddingDomain::Slice<out_dims>
5    typename Base; // the base point, a Point<EmbeddingDomain::NumDims>
6    typename Promoted_t; // the result of a call to Promote, a Point<EmbeddingDomain::
         NumDims>
7    typename Slice_t; // the domain not covered: EmbeddingDomain::Slice<EmbeddingDomain::
         NumDims - out_dims>;
8
9    Domain_t::Ptr_t Dom(); // the domain object being traversed
10   Point<out_dims> Collapse(); // convert to a point which can index a Domain_t
11
12   Slice_t::Ptr_t Slice(); // returns the domain perpendicular to this iterator's traversal
          at the point covered by the iterator
13   Promoted_t Promote(Point<EmbeddingDomain::NumDims - out_dims> p); // combine the point p
          with this one to obtain a point which can index an array allocated over a domain of
          type EmbeddingDomain.  this handles all the concerns with iterating along different
          axes.
14
15   Self_t& operator++(); // prefix increment
16   Self_t operator++(int); // postfix increment
17   bool end(); // true if the iterator is at the end of the domain.  equivalent to operator
         ==(embedding_domain.end())
18 };
```

Figure A.1: The `Iterator` concept.

`Domain_t`). `Promoted_t` (Line 6) is the point type that is the result of the `Promote` operation (Line 13), and it is a point in the embedding domain. The domain being traversed is of type `Domain_t` (Line 4) and may be retrieved using the `Dom` function (Line 9). The remainder of the embedding domain is of type `Slice_t` (Line 7) and is accessible using the `Slice` function (Line 12). The increment operators (Lines 15 and 16) advance the iterator. The `end` function (Line 17) returns true if the iterator is at the end of the domain it is covering. `!i.end()` should return the same value as `i != dmn.end(`$\langle axes \rangle$`)`, where `dmn` is the embedding domain which created the iterator via `i = dmn.begin(`$\langle axes \rangle$`)`.

Predicates

```
template <dims_t num_dims>
struct Predicate;
```

Predicates in Saaz live in the `Predicate` sub-namespace of the `Saaz` namespace. All predicates must derive from the `Thunk` class (Figure A.2). The basic criteria is that it be able to take a `Point` object and return true if computation should be evaluated at that point, and false otherwise. All domains are predicates, passing all points within their bounds.

There is a special predicate, `OptimizedEverywhere` (Figure A.3). All functions and classes in Saaz which take a predicate specialize an implementation for

```
1 namespace Predicate
2 {
3   template < dims_t num_dims >
4   struct Thunk
5   {
6     static const dims_t NumDims = num_dims;
7     virtual ~Thunk() { }
8     virtual bool operator()(Point<num_dims> p) const = 0;
9   };
10 }
```

Figure A.2: The `Thunk` base class.

```
1 namespace Predicate
2 {
3   template < dims_t num_dims >
4   struct OptimizedEverywhere : public Thunk<num_dims>
5   {
6     bool operator()(Point<num_dims> p) const
7     { assert(false); return true; } // this should never actually be called
8   };
9 }
```

Figure A.3: The `OptimizedEverywhere` predicate.

this type. This specialized implementation will not have the overhead of checking the predicate. In this way, code can be written to handle predicates without worrying about extra overhead in the case where the predicate is always true.

Saaz also provides several built-in predicates for simple operators:

1. `Everywhere` : `true`
2. `Nowhere` : `false`
3. `Mask` : `arr[p] != 0`
4. `LessThan` : `arr[p] < threshold`
5. `GreaterThan` : `arr[p] > threshold`
6. `LessThanOrEqualTo` : `arr[p] <= threshold`
7. `GreaterThanOrEqualTo` : `arr[p] >= threshold`
8. `Equals` : `arr[p] == threshold`
9. `NotEquals` : `arr[p] != threshold`

The conditionals compare a `threshold` value (passed to the predicate's constructor) to the value at the specified point (`p`) of an array (`arr`) which is also passed to the predicate's constructor.

Reductions

```
template <dims_t total_dims, typename element_t,
          typename OutDomain_t, typename out_element_t>
struct Aggregator;
```

Reductions in Saaz are implemented by subclasses of the `Aggregator` class. Reductions can run in parallel if they subclass the `ParallelAggregator` class. Both the `Aggregator` and `ParallelAggregator` class are class templates, but we omit their rather long template declarations here for clarity[1]. An aggregator's `OutputDimensions` specify which axes show up in the result array, the data in the remaining dimensions is reduced to a single `out_element_t`.

`Aggregator` objects implement (override) six member functions, corresponding to different parts of the loop nest. These functions are: `Initialize`, `PreProcess`, `Process`, `PostProcess`, `Finalize`, and `Return`. Figure A.4 illustrates how an aggregator object (`agg`) implements a reduction. Line 2 constructs the domain to be output by using the collapse method of the input domain, `dom`. Effectively, the extents matching the specified axes of the input domain are copied to a new domain. Line 3 provides the aggregator's initialization routine with the output domain. This lets the aggregator construct an array to hold the result, as well as any intermediate arrays. The aggregator is responsible for destroying this domain object, and can do so here. Once the aggregator has initialized, the outer loop (Line 4) covers the dimensions to which the reduction collapses the original domain (`dom.begin(agg->GetOutputDimensions())` will cover the same points as `collapsed.begin()`). Line 6 takes the subdomain in the dimensions perpendicular to those covered by the outer iterator, `i`.

Because iterators in Saaz are embeded in the domain over which they iterate, they must be reduced in rank if they are to index the domain they traverse. In

[1]See files `saaz/aggregator.hxx` and `saaz/aggregator.inl` in the Saaz distribution.

```
1 typedef Domain<total_dims>::Slice<total_dims-OutDomain_t::NumDims>::Type Slice_t;
2 Ptr< OutDomain_t > collapsed = dom.Collapse(agg->GetOutputDimensions());
3 agg->Initialize(collapsed);
4 for (Domain_t::Iterator<OutDomain_t::NumDim> i = dom.begin(agg->GetOutputDimensions());
      !i.end(); ++i)
5 {
6   Ptr<Slice_t> slice = i.Slice();
7   Point<out_dims> pi = i.Collapse();
8
9   agg->PreProcess( pi ); // convert i to a Point<out_dims> instead of Point<total_dims>
        and initialize this aggregation's counter to it
10
11  for (Slice_t::Iterator<OutDomain_t::NumDims> j = slice.begin(); !j.end(); ++j)
12  {
13    agg->Process( pi, i.Promote(j) );
14  }
15
16  agg->PostProcess( pi );
17 }
18 agg->Finalize();
19 return agg->Return();
```

Figure A.4: The steps involved in performing a reduction, using the aggregation object `agg` over the input domain `dom`.

our case, `i` is a point in `total_dims`-dimensional space. To convert it to the output domain (`OutDomain_t::NumDims`-dimensional space), we must `Collapse` it (Line 7, discussion in Section 3.2.4).

Line 9 lets the aggregator handle setup for the inner loop. For example, an averaging reduction might use this opportunity to initialize the sum counter to zero. The loop at Line 11 traverses the original domain's remaining dimensions. The `Process` call at Line 13 performs the majority of the work. It is here that the most expensive operations of a reduction can be performed. The `Process` function is passed a point in the output domain (`pi`), and a point in the input domain (`i.Promote(j)`). This point-promotion is necessary to combine the points which are traversing the disjoint subdomains into a point in the original domain (see discussion from Section 3.2.4).

The exit of the inner loop is handled at Line 16 in the `PostProcess` method (this allows the average aggregator to divide by the number of points covered). `Finalize` (Line 18) performs any cleanup the aggregator needs, such as freeing temporary arrays. `Return` (Line 19) provides the result of the aggregator.

Appendix B

Saaz Extensions

The basic Saaz library provides some useful primitives for a wide range of computational tasks. There are still, however, some cases in which it is useful to extend the basic Saaz library.

B.1 Fortran Support

Saaz uses `mmap` to ensure fast disk-load times and accesses for arrays. `mmap` allows a program to treat a region of disk as a region of memory. The operating system maps the appropriate memory region from the file cache to provide direct access to this data. Writes to memory are reflected in changes on disk, and reads from memory access data from disk. Saaz uses its own file format to make this process easier.

Because many CFD simulators are implemented in Fortran, we have ensured that there is a method of saving and loading data from these files into programs which are written in Fortran. Saaz provides a Fortran interface to saving and loading files. The `fortransave` family of functions lets Fortran-allocated arrays be saved in the Saaz file format.

Loading Saaz-formatted files is more complicated. The simplest way is for the file to be read directly into a Fortran array. Unfortunately, this can take a long time and additional memory to buffer the input file.

Ideally we could just point a fortran array to the data which has been

```
1        interface
2             subroutine saazopen3d(dest, mx, my, mz, filename)
3                 implicit none
4                 real*8, pointer,dimension(:,:,:) :: dest
5                 integer mx, my, mz
6                 character filename*(*)
7             end subroutine saazopen3d
8             subroutine saazreassign(dest, src)
9                 implicit none
10                real*8, pointer,dimension(:,:,:) :: dest, src
11            end subroutine saazreassign
12            subroutine saazforcefree(arr)
13                implicit none
14                real*8, dimension(:,:,:) :: arr
15            end subroutine saazforcefree
16       end interface
```

Figure B.1: The interface declaration. Note that some arrays are passed as a pointer. This ensures that they are passed as a `void**`, thus allowing the C++ routine to change the destination of the Fortran variable.

mmaped by C++. Unfortunately, while Fortran arrays are opaque pointers, their indexing is handled by another data-structure created by the compiler to hold the array's extents. The user has no access to this data-structure, but it is necessary to map the multidimensional indices of Fortran arrays to a linear offset in memory (this is the process of linearization). Because this data-structure is generated by the Fortran compiler, and used only by code that the Fortran compiler generates, it is not available externally. This means a C++ function cannot access or create this data structure on its own.

Instead, by using Fortran pointers, we are able to trick the Fortran compiler into creating the extent table data-structure for us. Because we need pointers, this technique requires at least Fortran 90. The basic technique is to use C++ to perform tasks which the Fortran compiler is unaware of. By violating the assumptions made by the Fortran compiler, we can manipulate its data structures as we see fit. Figure B.1 shows the Fortran interface for these C++ functions. It is necessary to provide the interface explicitly because the implicit interface will not use the necessary `pointer` variables.

Figure B.2 shows the steps necessary for a Fortran program to `mmap` a Saaz array file. This requires three variables: two pointers (`read_array`, and `use_array`), and an allocatable Fortran target array (`indexing_dummy`). Figure B.3 shows the manipulation of these pointers. The first step is on Line 5. This is a call to a C++ function which will `mmap` the file "`double.sa3`" into the `read_array` variable. Note

```
1       real*8,target,allocatable,dimension(:,:,:) :: indexing_dummy
2       real*8,pointer :: read_array(:,:,:)
3       real*8,pointer :: use_array(:,:,:)
4       integer mx,my,mz
5       call saazopen3d(read_array, mx, my, mz, 'double.sa3')
6       allocate (indexing_dummy(1:mx, 1:my, 1:mz))
7       use_array => indexing_dummy
8       call saazreassign(use_array, read_array)
9       call saazforcefree(indexing_dummy)
```

Figure B.2: The steps necessary to `mmap` Saaz arrays into Fortran programs.

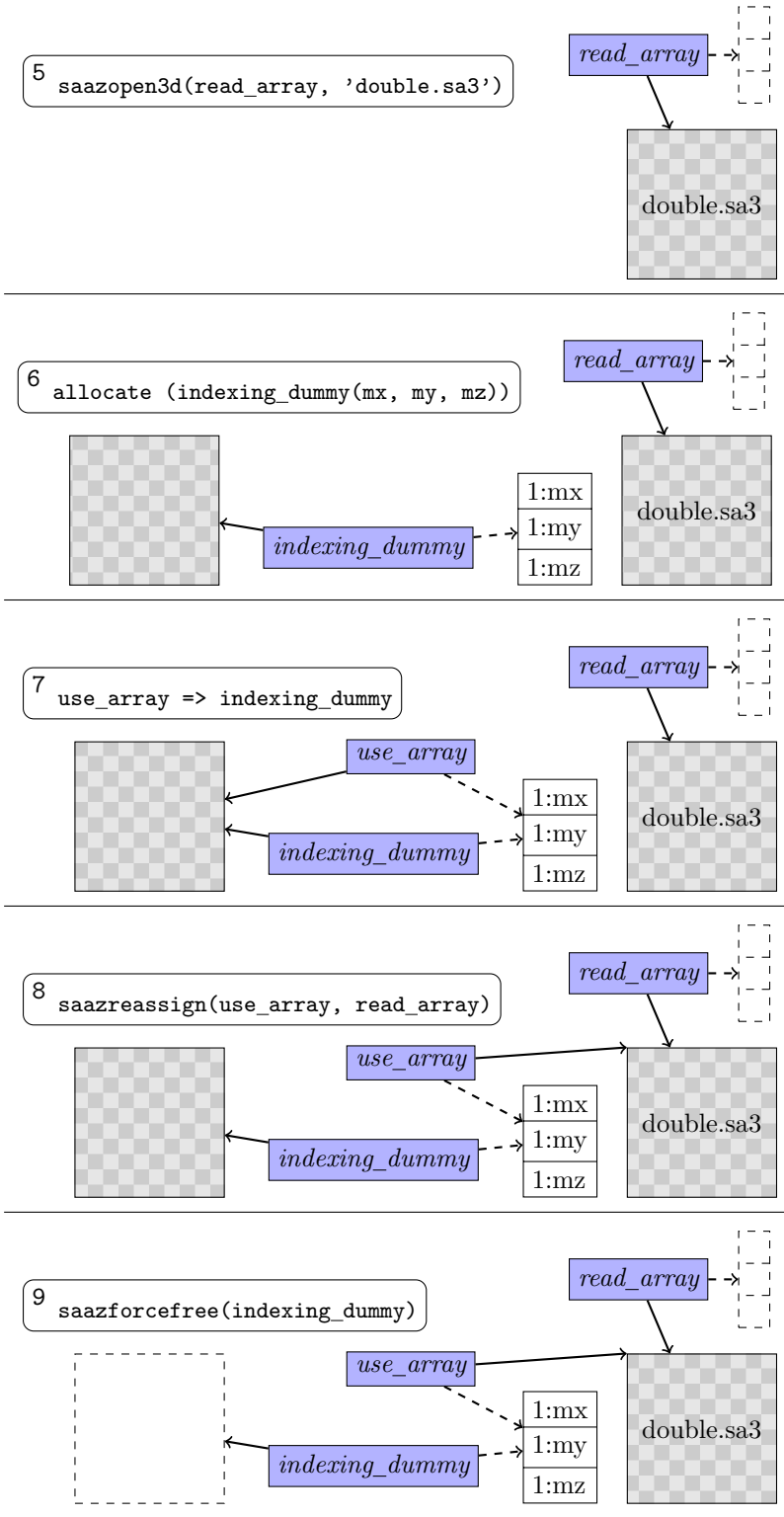
that the Fortran variable has not been initialized. Because the variable is passed by address, the C++ function is able to reassign what that variable points to. In a similar fashion, it also returns the bounds of the array: `mx`, `my`, and `mz`. The regular fortran array, `indexing_dummy`, is allocated on Line 6 to have the same dimensions as the array which has been loaded.

At this point, there are now two full arrays which have been allocated in memory. `indexing_dummy` has been allocated by Fortran, and, depending on the compiler, may or may not have been initialized. If it has not been initialized, then it might not be occupying physical memory. Because the `read_array` array has been mapped, but has not been read (outside of the first page), the pages for the array have not been faulted into memory yet, and it takes up minimal physical space (although it still has a virtual address space allocated to it).

Line 7 tells Fortran that the `use_array` variable will point to the data in the `indexing_dummy` array. At this point, Fortran associates the extent map and linearization parameters that were set for `indexing_dummy` to be the ones to use for `use_array`. Now `use_array` can be indexed properly, but it still doesn't point to the right data, so it should not be actually indexed yet. To assign the data, we use `saazreassign` to assign the `use_array` pointer the value of `read_array`. At this point, `read_array` can be reused to load another array.

Finally, because `indexing_dummy` is taking up memory, we use `saazforcefree` (Line 9) to release that memory (using the C++ function, `free`). This is somewhat dangerous, because it depends on the Fortran allocate using the corresponding `malloc`. At the time of this writing, our process works with both the Intel and GNU Fortran compilers (ifort 12.0 and gfortran 4.7).

Subsequent arrays can be loaded using the same `read_array`. If these new



(Line 5) C++ is able to assign the value-pointer of `read_array`, but does not have access to Fortran's extent table.

(Line 6) We create Fortran's extent table by allocating `indexing_dummy`.

(Line 7) The array we want to use is then assigned to `indexing_dummy`, telling Fortran that they have the same extent table.

(Line 8) C++ is then able to redirect the value pointer of `use_array` to point to the mmaped file.

(Line 9) Finally, we use C++ to free the memory allocated for `indexing_dummy`. We can now reuse `read_array`. `indexing_dummy` may never be used again, or deallocated in Fortran.

Figure B.3: The internal pointers manipulated in Figure B.2, to which the line numbers refer. Solid arrows are pointers, while dashed arrows are maps the Fortran compiler maintains to the extent tables for each array.

arrays have the same extents, then `indexing_dummy` can be used, but it must not be reallocated. Because the Fortran compiler thinks `use_array` and `indexing_dummy` refer to the same memory locations, they share an extent table. Deallocating `indexing_dummy` will corrupt `use_array`.

B.2 Cascade

Not all domain scientists are completely comfortable with C++. In an effort to make Saaz more user-friendly for non-expert programmers, we have built the Cascade library on top of Saaz. Cascade renames some data-types and wraps certain operations which are common in CFD. Because it supports only a few canned operations, Cascade is less flexible than Saaz. Nonetheless, it is useful under certain circumstances.

Saaz uses a schema for arrays and domains that makes use of template parameters for properties like rank (number of dimensions). Cascade introduces separate types for common array types using a full type name instead of a template. For example, a 3-dimensional array of floats is `Array_f3`, a 1-dimensional array of doubles is `Array_d1`. This is accomplished by making subclasses that forward all their functions on to the normal Saaz datatypes.[1] This makes types easier to work with for those not familiar with templates.

We also use macro wrappers to make it easier to define computations over arrays. The `MakeFunction` macro takes a function name, a name for a point-valued variable, and the body which is to execute at every point in a domain. Combined with a `Foreach` function template (similar to `std::foreach`), this makes it easy to write new queries and hide boilerplate. An example is given in Figure B.4. Recall that `u` is the velocity in the x direction, `um` is the planar average of `u`. The `cfg` object wraps a number of common assumptions, such as which axis is the inhomogeneous one, and what the coordinate mapping may be. Note that this example uses the `LessThan` predicate to operate on points `p` in the domain of `lambda2` only where `lambda2[p] < threshold`.

[1] These subclasses are actually subclasses of the `Ptr` pointers to the Saaz types. They do not introduce new forwarding, but use that of the `Ptr` type from which they inherit.

```
1 MakeFunction(
2   ComputeUU, p,
3   {
4     double un = (u[p] - um[p.lst[cfg.INHOMOGENEOUS_DIM]]);
5     uu[p] = un*un;
6   }
7 );
8 ...
9 CFD_Foreach(cfg, lambda2.Domain(), ComputeUU(), LessThan_d3(lambda2, threshold));
```

Figure B.4: An example of using `MakeFunction` to compute the self-correlation of the velocity in the x direction. The operation is restricted to locations inside vortices: places where `lambda2` is less than `threshold`.

```
1 MakePlanarAverage_d(
2   ComputeYZDiss, j, p,
3   Square( ((v[p + cfg.PK] - vm[j  ]) - (v[p - cfg.PK] - vm[i  ])) / (2*cfg.dz) ) +
4   Square( ((w[p + cfg.PJ] - wm[j+1]) - (w[p - cfg.PJ] - wm[i-1])) / (2*cfg.dy) )
5 );
6 ...
7 CFD_Aggregate(cfg, inset, ComputeYZDiss());
```

Figure B.5: An example of using `MakePlanarAverage` to compute the yz-Dissipation. `Square` is a simple function which squares it's argument.

The `MakePlanarAverage` macro family makes it easy to write a particular common type of reduction: the *planar average*. Many queries which operate on the full 3-dimensional domain are reduced to a single dimension. The macros allow the user to specify which parameters (if any) the aggregator takes as well as the expression to compute for the `Process` function, which gets averaged over each plane. An example is given in Figure B.5

B.3 Saaz Utilities

Over the course of working with Saaz it has been useful to construct several utility programs.

1. `convert` is perhaps the most useful. This program gives information about any Saaz array files, such as their rank, dimensions, and layout. It can also convert Saaz array files into an ascii format suitable for importation into Matlab or gnuplot. Finally, it can copy a subregion of an array file into another array file. The subregion can be specified by coordiantes or by another array file and a threshold value.

2. `diff` compares two arrays and reports at what points their values differ, using a range for floating-point values.

3. `expand` and `reduce` change the size of an array, either linearly interpolating values or dropping values. This is useful to run quick sanity-checks against real data-sets that are smaller or larger than an existing one.

Appendix C

Extended Examples

In this appendix we present extended examples. We present multiple implementations of the planar average query (Section 3.3.2), as well as our eduction criteria, λ_2(Section 3.3.5). Figure C.1 shows how the planar average would be implemented in Fortran. Note that the inner loop covers the z dimension and then the x dimension because Fortran arrays are column-major. The C examples iterate over the x dimension and then the z dimension because they store arrays in a row-major order. Figures C.2 and C.3 implement the planar average in plain C++, using no additional user-defined abstractions. The first uses C-style 3D arrays, the second uses linear arrays. Both use a row-major layout, although, if the linearization operation were altered in Figure C.2 Lines 15-17, it could use a column-major layout.

Our next set of planar average implementations are performed in Saaz. Figure C.4 shows the code structure which is used in our results from Chapter 6. This code matches the exact syntax in Saaz as presented in Appendix A. The heavy use of template arguments motivated the development of the Cascade frontend. Figure C.5 demonstrates how the types used in Cascade simplify the type names.

Figure C.6 demonstrates how Saaz's built-in averaging-aggregator can be used to perform the planar average. Figure C.7 shows the use of the Cascade's built-in planar-average-calculating function. The `CFD_Config` type holds configuration parameters relating to the data schema, in particular the mapping from

```
1       function PlanarAverage(arr, x_min, x_max,
2      & y_min, y_max, z_min, z_max)
3        integer x_min, x_max
4        integer y_min, y_max
5        integer z_min, z_max
6        real*8 dimension arr(x_min:x_max,
7      & y_min:y_max, z_min:z_max)
8        integer i,j,k
9        real*8 avg(y_min:y_max)
10       do j=y_min,y_max
11         avg(j) = 0
12         do k=z_min,z_max; do i=x_min,x_max
13           avg(j) = avg(j) + arr(i,k,j)
14         end do; end do
15          avg(j) = avg(j) / dble((x_max-x_min+1)
16     & * (z_max-z_min+1))
17       end do
18       return avg
19      end function
```

Figure C.1: Planar Average, implemented in Fortran.

```
1  double* PlanarAverage(double* arr,
2      int x_min, int x_max, int y_min,
3      int y_max, int z_min, int z_max)
4  {
5    int i,j,k;
6    unsigned long long idx;
7    double* avg = new double[y_max-y_min+1];
8    for (j = y_min; j < y_max + 1; ++j)
9    {
10     avg[j] = 0;
11     for (i = x_min; i < x_max + 1; ++i)
12     {
13       for (k = z_min; k < z_max + 1; ++k)
14       {
15         idx = (k-z_min) + ((j-y_min) +
16               (i-x_min) * (y_max-y_min+1))
17                         * (z_max-z_min+1);
18         avg[j] += arr[idx];
19       }
20     }
21     avg[j] /= (z_max-z_min+1)*(x_max-x_min+1);
22   }
23   return avg;
24 }
```

Figure C.2: Planar Average, implemented in C, using linear arrays.

```
1  double*** Make3DArray(int x_len, int y_len, int z_len)
2  {
3    uint8_t* raw = new uint8_t[sizeof(double**) * x_len + sizeof(double*) * (y_len * x_len)
         + sizeof(double) * (z_len * y_len * x_len)];
4    double*** arr1 = (double***)raw;
5    raw += x_len * sizeof(double**);
6    for (int i = 0; i < x_len; ++i)
7    {
8      arr1[i] = (double**)(raw + (i * y_len) * sizeof(double*));
9    }
10   raw += x_len * y_len * sizeof(double*);
11   elem_t** arr2 = (elem_t**)raw;
12   for (int i = 0; i < x_len; ++i)
13   {
14     for (int j = 0; j < y_len; ++j)
15     {
16       arr1[i][j] = (double*)(raw + ((i*y_len*z_len + j*z_len) * sizeof(double*)));
17     }
18   }
19 }
20
21 double* PlanarAverage(double*** arr)
22 {
23   int i,j,k;
24   unsigned long long idx;
25   double* avg = new double[y_max-y_min+1];
26   for (j = 0; j < y_max - y_min + 1; ++j)
27   {
28     avg[j] = 0;
29     for (i = 0; i < x_max - x_min + 1; ++i)
30     {
31       for (k = 0; k < z_max - z_min + 1; ++k)
32       {
33         avg[j] += arr[i][j][k];
34       }
35     }
36     avg[j] /= (z_max-z_min+1)*(x_max-x_min+1);
37   }
38   return avg;
39 }
```

Figure C.3: Planar Average, implemented in C, using C-style 3D arrays. We also include the helper function, `Make3DArray`, which allocates a contiguous memory region and fills in the C-pointers so it can be accessed with three indexing operations.

```
1  using namsepace Saaz;
2
3  Array<Domain<1>, double>::Ptr_t
4  PlanarAverage(Array<Domain<3>, double>::Ptr_t arr, dims_t Y_AXIS)
5  {
6    Domain<3>::Ptr_t dom = arr.Domain();
7    Domain<1>::Ptr_t outdom = dom.Collapse(Y_AXIS);
8    Array<Domain<1>, double>::Ptr_t avg = Array<Domain<1>, double>::Create(outdom, true);
9    for (Domain<3>::Iterator<1> j = dom.begin(Y_AXIS); !j.end(); ++j)
10   {
11     Point<1> pj = j.Collapse();
12     avg[pj] = 0;
13     for (Domain<2>::Iterator<2> ik = j.Slice().begin(); !ik.end(); ++ik)
14     {
15       Point<3> ijk = j.Promote(ik);
16       avg[pj] += arr[ijk];
17     }
18     avg[pj] /= j.Slice().Size();
19   }
20   return avg;
21 }
```

Figure C.4: Planar Average, implemented in Saaz, using a loop nest.

```
1  using namsepace Saaz;
2
3  Array_d1 PlanarAverage(Array_d3 arr, dims_t Y_AXIS)
4  {
5    Domain_3 dom = arr.Domain();
6    Domain_1 outdom = dom.Collapse(Y_AXIS);
7    Array_d1 avg(outdom);
8    for (Domain_3::Iterator_1 j = dom.begin(Y_AXIS); !j.end(); ++j)
9    {
10     Point_1 pj = j.Collapse();
11     avg[pj] = 0;
12     for (Domain_2::Iterator ik = j.Slice().begin(); !ik.end(); ++ik)
13     {
14       Point<3> ijk = j.Promote(ik);
15       avg[pj] += arr[ijp];
16     }
17     avg[pj] /= j.Slice().Size();
18   }
19   return avg;
20 }
```

Figure C.5: Planar Average, implemented in Saaz, using Cascade types.

```
1  using namsepace Saaz;
2
3  Array<Domain<1>, double>::Ptr_t
4  PlanarAverage(Array<Domain<3>, double>::Ptr_t arr, dims_t Y_AXIS)
5  {
6    Aggregator_Avg< 3, double, Domain<1> > averager;
7    averager.SetOutputDims(Y_AXIS);
8    averager.Apply(arr);
9    return averager.Return();
10 }
```

Figure C.6: Planar Average, implemented in Saaz, using an aggregator object.

coordinates to dimensions.. Normally this would be passed to a function, but we construct it here to show how it relates to the `Y_AXIS` variable which a Saaz implementation uses to keep track of the inhomogeneous y dimension. `cfg.SetDims` sets up the making from coordinates to dimensions.

Figures C.8 and C.9 contain the code for our eduction criteria, λ_2(see Section 3.3.5). This example uses 3D C-style arrays. In our results from Chapter 6, λ_2 is thresholded to include approximately 2.6% of the domain.

```
using namsepace Saaz;

Array<Domain<1>, double>::Ptr_t
PlanarAverage(Array<Domain<3>, double>::Ptr_t arr, dims_t Y_AXIS)
{
  CFD_Config cfg;
  switch (Y_AXIS)
  {
    case 0: cfg.SetDims(CFD_Config::J, CFD_Config::I, CFD_Config::K); break;
    case 1: cfg.SetDims(CFD_Config::I, CFD_Config::J, CFD_Config::K); break;
    case 2: cfg.SetDims(CFD_Config::I, CFD_Config::K, CFD_Config::J); break;
  }
  cfg.SetInhomogeneous(CFD_Config::J);
  return CFD_PlanarAverage_d3(cfg, arr);
}
```

Figure C.7: Planar Average, implemented in Saaz, using Cascade's planar average support.

```
double*** Lambda2(double*** u, double*** v, double*** w,
    int x_min, int x_max, int y_min, int y_max,
  int z_min, int z_max, double dx, double dy, double dz)
{
  static const double pi = 3.141592653;
  double*** lambda2 = Make3DArray<double>(x_min,x_max,y_min, y_max,z_min,z_max);
  for (int i = x_min+1; i < x_max; ++i)
  for (int j = y_min+1; j < y_max; ++j)
  for (int k = z_min+1; k < z_max; ++k)
  {
    double s11,  s12,  s13,  s22,  s23,  s33; // diagonal
    double o12,  o13,  o23;               // off-diagonal
    s11 = (u[i+1][j  ][k  ] - u[i-1][j  ][k  ]) / (2*dx);
    s22 = (v[i  ][j+1][k  ] - v[i  ][j-1][k  ]) / (2*dy);
    s33 = (w[i  ][j  ][k+1] - w[i  ][j  ][k-1]) / (2*dz);
    s12 = .5 * ( (u[i  ][j+1][k  ] - u[i  ][j-1][k  ]) / (2.0 * dy) +
                 (v[i+1][j  ][k  ] - v[i-1][j  ][k  ]) / (2.0 * dx) );
    s13 = .5 * ( (u[i  ][j  ][k+1] - u[i  ][j  ][k-1]) / (2.0 * dz) +
                 (w[i+1][j  ][k  ] - w[i-1][j  ][k  ]) / (2.0 * dx) );
    s23 = .5 * ( (v[i  ][j  ][k+1] - v[i  ][j  ][k-1]) / (2.0 * dz) +
                 (w[i  ][j+1][k  ] - w[i  ][j-1][k  ]) / (2.0 * dy) );
    o12 = .5 * ( (u[i  ][j+1][k  ] - u[i  ][j-1][k  ]) / (2.0 * dy) -
                 (v[i+1][j  ][k  ] - v[i-1][j  ][k  ]) / (2.0 * dx) );
    o13 = .5 * ( (u[i  ][j  ][k+1] - u[i  ][j  ][k-1]) / (2.0 * dz) -
                 (w[i+1][j  ][k  ] - w[i-1][j  ][k  ]) / (2.0 * dx) );
    o23 = .5 * ( (v[i  ][j  ][k+1] - v[i  ][j  ][k-1]) / (2.0 * dz) -
                 (w[i  ][j+1][k  ] - w[i  ][j-1][k  ]) / (2.0 * dy) );

    double a11,  a12,  a13,  a22,  a23,  a33;
    a11 = s11*s11 + s12*s12 + s13*s13 -
        o12*o12 - o13*o13;
    a12 = s11 * s12  +  s12 * s22  +  s13 * s23  -
        o13 * o23;
    a13 = s11 * s13  +  s12 * s23  +  s13 * s33  + //yes, this is a +
        o12 * o23;
    a22 = s12*s12 + s22*s22 + s23*s23  -
        o12*o12 - o23*o23;
    a23 = s12 * s13  +  s22 * s23  +  s23 * s33  -
        o12 * o13;
    a33 = s13*s13 + s23*s23 + s33*s33 -
        o13*o13 - o23*o23;

    // Continued on next page
```

Figure C.8: The first part of λ_2, implemented using C-style 3D arrays.

```
44      // Continued from previous page
45
46      double B,  C,  D;
47      B = -(a11 + a22 + a33);
48      C = -(a12*a12 + a13*a13 + a23*a23  -  a11 * a22 - a11 * a33 - a22 * a33);
49      D = -(2.0 * a12 * a13 * a23  -  a11 * a23*a23  -  a22 * a13*a13  -  a33 * a12*a12  +
            a11 * a22 * a33);
50
51      double q,  r;
52      q = (3.0 * C - B*B) / 9.0;
53      r = (9.0 * C * B - 27 * D - 2.0 * B*B*B) / 54.0;
54      double theta;
55      theta = acos( r / sqrt( -q*q*q ) ); // q is always < 0
56
57      double eigen_1,  eigen_2,  eigen_3;
58      eigen_1 = 2 * sqrt( -q ) * cos(theta / 3.0) - B / 3.0;
59      eigen_2 = 2 * sqrt( -q ) * cos((theta + 2.0 * pi) / 3.0) - B / 3.0;
60      eigen_3 = 2 * sqrt( -q ) * cos((theta + 4.0 * pi) / 3.0) - B / 3.0;
61
62      if      (eigen_1 <= eigen_2 && eigen_2 <= eigen_3)
63        lambda2[i][j][k] = eigen_2;
64      else if (eigen_3 <= eigen_2 && eigen_2 <= eigen_1)
65        lambda2[i][j][k] = eigen_2;
66      else if (eigen_1 <= eigen_3 && eigen_3 <= eigen_2)
67        lambda2[i][j][k] = eigen_3;
68      else if (eigen_2 <= eigen_3 && eigen_3 <= eigen_1)
69        lambda2[i][j][k] = eigen_3;
70      else if (eigen_2 <= eigen_1 && eigen_1 <= eigen_3)
71        lambda2[i][j][k] = eigen_1;
72      else if (eigen_3 <= eigen_1 && eigen_1 <= eigen_2)
73        lambda2[i][j][k] = eigen_1;
74      else
75        assert(false);
76    }
77    return lambda2;
78  }
```

Figure C.9: The second part of λ_2, implemented using C-style 3D arrays.

Appendix D

Template Errors

A simple example of two expression template errors. (Figure D.1) shows the code for a simple system of expression templates. (Figure D.2) shows the error-messages that result when the user makes the mistake of adding `arr` instead of `one` to `x` on Line 39. Because templates are expanded on-demand, the compiler reports an error on Line 39 but not on Line 38, where the templates are not expanded. This type of context-dependent error can be confusing. The error in (Figure D.3) occurs when removing the parentheses around the `Variable()` constructor on Line 35. This is an especially obscure error, as parentheses are usually just for precedence and it is counter-intuitive to think that the parentheses would cause the `x` variable to not be an `Expression` containing a `Variable`, but rather a function pointer which took a function pointer which returned a `Variable` object. Neither of these errors are nearly helpful enough for inexperienced users who would rather be told "you used `arr` instead of `one`", or "trust me, you need parentheses around `Variable()`".

```
1 #include <vector>
2 #include <algorithm>
3 struct Variable {
4   double operator() (double v) { return v; }
5 };
6 struct Constant {
7   double c;
8   Constant (double d) : c (d) { }
9   double operator() (double) { return c; }
10 };
11 template <typename E> struct Expression {
12   E expr;
13   Expression (E e) : expr(e) { }
14   double operator() (double d) { return expr(d);  }
15 };
16 template <typename L, typename R, typename OP>
17 struct BinaryExpression {
18   L l; R r;
19   BinaryExpression (L l_, R r_) : l (l_), r (r_) { }
20   double operator() (double d)
21   { return OP::apply (l(d), r(d)); }
22 };
23 struct Add {
24   static double apply(double l, double r)
25   { return l + r; }
26 };
27 template <typename L, typename R>
28 Expression< BinaryExpression<L, R, Add> >
29 operator+(L l, R r) {
30   typedef BinaryExpression<L, R, Add> Expr;
31   return Expression<Expr>(Expr(l, r));
32 }
33 int main() {
34   Expression<Constant> one(Constant(1));
35   Expression<Variable> x((Variable()));
36   std::vector<double> arr;
37   arr.push_back(10); arr.push_back(20);
38   arr + x;
39   std::for_each(arr.begin(), arr.end(), arr + x);
40 }
```

Figure D.1: add-error.cxx

```
1 add-error.cxx: In member function ‘double BinaryExpression<L, R, OP>::operator()(double) [
      with L = std::vector<double, std::allocator<double> >, R = Expression<Variable>, OP =
      Add]’:
2 add-error.cxx:14:   instantiated from ‘double Expression<E>::operator()(double) [with E =
      BinaryExpression<std::vector<double, std::allocator<double> >, Expression<Variable>,
      Add>]’
3 /usr/include/c++/4.2.1/bits/stl_algo.h:159:   instantiated from ‘_Function std::for_each(
      _InputIterator, _InputIterator, _Function) [with _InputIterator = __gnu_cxx::
      __normal_iterator<double*, std::vector<double, std::allocator<double> > >, _Function =
       Expression<BinaryExpression<std::vector<double, std::allocator<double> >, Expression<
      Variable>, Add> >]’
4 add-error.cxx:39:   instantiated from here
5 add-error.cxx:21: error: no match for call to ‘(std::vector<double, std::allocator<double>
       >) (double&)’
```

Figure D.2: The g++ error message caused by Figure D.1, Line 39 of main.

```
1 add-error.cxx: In member function ‘double BinaryExpression<L, R, OP>::operator()(double) [
      with L = std::vector<double, std::allocator<double> >, R = Expression<Variable> (*)(
      Variable (*)()), OP = Add]’:
2 add-error.cxx:14:   instantiated from ‘double Expression<E>::operator()(double) [with E =
      BinaryExpression<std::vector<double, std::allocator<double> >, Expression<Variable>
      (*)(Variable (*)()), Add>]’
3 /usr/include/c++/4.2.1/bits/stl_algo.h:159:   instantiated from ‘_Function std::for_each(
      _InputIterator, _InputIterator, _Function) [with _InputIterator = __gnu_cxx::
      __normal_iterator<double*, std::vector<double, std::allocator<double> > >, _Function =
       Expression<BinaryExpression<std::vector<double, std::allocator<double> >, Expression<
      Variable> (*)(Variable (*)()), Add> >]’
4 add-error.cxx:39:   instantiated from here
5 add-error.cxx:21: error: no match for call to ‘(std::vector<double, std::allocator<double>
       >) (double&)’
6 add-error.cxx:21: error: cannot convert ‘double’ to ‘Variable (*)()’ in argument passing
```

Figure D.3: The g++ error message caused by removing the parentheses around `Variable()` in Figure D.1, Line 35.

Bibliography

[ABD+90] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing 1990, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In Pierre Cointe, editor, *ECOOP 96 Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer Berlin / Heidelberg, 1996.

[AWZ88] Bowen Alpern, Mark N. Wegman, and Frank Kenneth Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pages 1–11. ACM, 1988.

[BC89] M.J. Berger and Phil Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[BC00] Paul Billant and Jean-Marc Chomaz. Experimental Evidence for a New Instability of a Vertical Columnar Vortex Pair in a Strongly Stratified Fluid. *Journal of Fluid Mechanics*, 418(1):167–188, 2000.

[Bew10] Thomas Bewley. *Numerical Renaissance: Simulation, Optimization & Control*. Renaissance Press, San Diego, 2010.

[BS96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[CCPA+10] Ashish Raniwala Craig Chambers, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 363–375, New York, NY, USA, 2010. ACM.

[CFE99] Brad Calder, Peter Feller, and Alan Eustace. Value Profiling and Optimization. *Journal of Instruction-Level Parallelism*, 1(6), March 1999.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 397–408, New York, NY, USA, 1994. ACM.

[CN96] Charles Consel and François Noël. A General Approach for Run-Time Specialization and its Application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 145–156, New York, NY, USA, 1996. ACM.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.

[CPC90] Min S. Chong, A. E. Perry, and B. J. Cantwell. A General Classification of Three-Dimensional Flow Fields. *Physics of Fluids*, 2:408–420, 1990.

[dAJH12] Rodrigo Vilela de Abreu, Niclas Jansson, and Johan Hoffman. Computation of Aeroacoustic Sources for a Complex Nose Landing Gear Geometry Using Adaptivity. In *In Proceedings of the Second Workshop on Benchmark problems for Airframe Noise Computations*, BANC-II, 2012.

[DCHH88a] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: an Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988.

[DCHH88b] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.

[Dew03] Stephen C. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley Professional, 2003.

[dGMH] Joel de Guzman, Dan Marsden, and Thomas Heller. Boost::Phoenix.

[DMN68] Ole-Johan Dahl, BJørn Myhrhaug, and Kristen Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the second conference on Applications of simulations*, pages 29–31. Winter Simulation Conference, 1968.

[FBK98] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient Runtime Support for Irregular Block-Structured Applications. *Journel of Parallel and Distributed Computing*, 50:61–82, April 1998.

[Fin98] Stephen J. Fink. *Hierarchical Programming for Block-Structured Scientific Calculations*. PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 1998.

[FJ05] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):232–275, February 2005.

[FJHL08] Francesco Fabozzi, Christopher D. Jones, Benedikt Hegner, and Luca Lista. Physics Analysis Tools for the CMS experiment at LHC. *Nuclear Science, IEEE Transactions on*, 55(6):3539–3543, 2008.

[GL00] Samuel Z. Guyer and Calvin Lin. An Annotation Language for Optimizing Software Libraries. *ACM SIGPLAN Notices*, 35(1):39–52, January 2000.

[GST+02] Jim Gray, Alex S. Szalay, Ani R. Thakar, Peter Z. Kunszt, Christopher Stoughton, Don Slutz, and Jan vandenBerg. Data Mining the SDSS SkyServer Database. Technical Report MSR-TR-2002-01, Microsoft Research, February 2002.

[HC88] Paul N. Hilfinger and Phillip Colella. FIDIL: A Language for Scientific Programming. Technical report, Lawrence Livermore National Laboratory, January 1988.

[HJ11] Johan Hoffman and Niclas Jansson. A Computational Study of Turbulent Flow Separation for a Circular Cylinder Using Skin Friction Boundary Conditions. *Quality and Reliability of Large-Eddy Simulations II*, 16:57–68, 2011.

[Hud96] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[HWM88] Julian C. R. Hunt, A. A. Wray, and Parviz Moin. Eddies, Streams, and Convergence Zones in Turbulent Flows. In *Studying Turbulence Using Numerical Simulation Databases, 2*, volume 1, pages 193–208, 1988.

[HY00] George Haller and George Xian Zhi Yuan. Lagrangian Coherent Structures and Mixing in Two-Dimensional Turbulence. *Physica D: Nonlinear Phenomena*, 147(3–4):352–370, 2000.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Eurosys, pages 59–72. ACM, 2007.

[JH95] Jinhee Jeong and Fazle Hussain. On the Identification of a Vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995.

[JJvT07] Hrvoje Jasak, Aleksandar Jemcov, and Željko Tuković. OpenFOAM: A C++ Library for Complex Physics Simulations. In *Coupled Methods in Numerical Dynamics*, pages 1–20, 2007.

[KASB11] Alden King, Eric Arobone, Sutanu Sarkar, and Scott B. Baden. The Saaz Framework for Turbulent Flow Queries. In *Proceedings of the 2011 IEEE conference on e-Science*. IEEE, December 2011.

[KB12] Alden King and Scott B. Baden. Reducing Library Overheads through Source-to-Source Translation. In *Proceedings of the 2012 International Conference on Computational Science*, June 2012.

[KBC+05] Ken Kennedy, Bradley Broom, A. Chauhan, Rob Fowler, J. Garvin, C. Koelbel, C. McCosh, and John Mellor-Crummey. Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proc. IEEE*, 93:387–408, 2005.

[KRJ10] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Dsolve: Safety Verification via Liquid Types. In *Computer Aided Verification*, pages 123–126. Springer, 2010.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LFGC07] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for Type-Error Messages. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 425–434, New York, NY, USA, 2007. ACM.

[LPW+08] Yi Li, Eric Perlman, Minping Wan, Yunke Yang, Charles Meneveau, Randal Burns, Shiyi Chen, Alexander Szalay, and Gregory Eyink. A Public Turbulence Database Cluster and Applications to Study Lagrangian Evolution of Velocity Increments in Turbulence. *arXiv.org*, pages 1–31, 2008.

[MEW08] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *Linq in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[ML85] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184. Prentice-Hall, Inc., 1985.

[Moi10] Parviz Moin. Needs and Opportunities in Fluid Dynamics Modeling and Flow Field Data Analysis. 2010.

[MR79] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Commun. ACM*, 22(2):96–103, February 1979.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[OSMC10] Catherine Olschanowsky, Allan Snavely, Mitesh R. Meswani, and Laura Carrington. PIR: PMaC's Idiom Recognizer. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, pages 189–196. IEEE Computer Society, 2010.

[OTCS09] Catherine Olschanowsky, Mustafa Tikir, Laura Carrington, and Allan Snavely. PSnAP: Accurate Synthetic Address Streams Through Memory Profiles. In *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing*, LCPC '09, October 2009.

[PMJ+05] Markus Püschel, José Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic,

Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, February 2005.

[PQ94] R. Parsons and Dan Quinlan. A++/P++ Array Classes for Architecture Independent Finite Difference Computations. In *Proceedings of the Conference on Object-Oriented Numerics*, OONSKI, pages 408–418, March 1994.

[QMPS02] Dan Quinlan, D.J. Miller, B. Philip, and Markus Schordan. Treating a User-Defined Parallel Library as a Domain-Specific Language. In *Proceedings of the 16th international Parallel and Distributed Processing Symposium*, volume 2 of *IPDPS*, pages 324–, Washington, DC, USA, April 2002. IEEE Computer Society.

[QSVY06] Dan Quinlan, Markus Schordan, Richard Vuduc, and Qing Yi. Annotating User-defined Abstractions for Optimization. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8, April 2006.

[QSYS06] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. *Classification and Utilization of Abstractions for Optimization*, volume 4313 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[RBKJ12] Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. CSolve: Verifying C with Liquid Types. In *Computer Aided Verification*, pages 744–750. Springer, 2012.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[RL08] James J. Riley and Erik Lindborg. Stratified Turbulence: A Possible Interpretation of Some Geophysical Turbulence Measurements. *Journal of the Atmospheric Sciences*, 65(7):2416–2424, 2008.

[Ros] Edward Rosten. TooN: Tom's Object-Oriented Numerics Library.

[Rus06] Francis Russell. Delayed Evaluation and Runtime Code Generation as a means to Producing High Performance Numerical Software. Technical report, University of London, 2006.

[TL72] Hendrik Tennekes and John L. Lumley. *A First Course in Turbulence*. The MIT Press, 1972.

[TTS+08] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep Typechecking and Refactoring. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 37–52, New York, NY, USA, 2008. ACM.

[Vel96] Todd Veldhuizen. *Expression Templates*, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.

[VJ02] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

[WK+] Joerg Walter, Mathias Koch, et al. Boost::uBLAS.

[WL91] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, pages 452–471, 1991.

[WSO+09] Nicholas J. Wright, Shava Smallen, Catherine Olschanowsky, Jim Hayes, and Allan Snavely. Measuring and Understanding Variation in Benchmark Performance. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, pages 438–443. IEEE, 2009.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.