Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2012) 1–10

Procedia Computer Science

Reducing Library Overheads through Source-to-Source Translation

Alden King, Scott Baden

University of California, San Diego

Abstract

Object oriented application libraries targeted to a specific application domain are an attractive means of reducing the software development time for sophisticated high performance applications. However, libraries can have the drawback of high abstraction penalties. We describe a domain specific, source-to-source translator that eliminates abstraction penalties in an array class library used to analyze turbulent flow simulation data. Our translator effectively flattens the abstractions, yielding performance within 75% of C code that uses primitive C arrays and no user-defined abstractions.

1. Introduction

Over the past few decades, advances in computational infrastructure and numerical methods have delivered simulations of unprecedented fidelity and accuracy, leading to new scientific discovery. However, a side effect of this "computational revolution" is the prodigious output of simulation data that challenges our ability to infer the relevant science. For example, there are important applications where the dynamics may be dominated by coherent structures and patterns, but these may be difficult to define, let alone understand. While visualization is an important tool for understanding pattern evolution, the process is subjective, not scalable to the study of *populations* of structures and patterns, and does not directly quantify the contribution of coherent structures to flow properties.

By and large, the CFD community relies on ad-hoc analysis tools that are customized to a specific application and must be painstakingly recoded for each new usage pattern. While there do exist some publicly available tools [1], they present the user with predefined or "canned" primitives. When a new need arises, there may be no way to optimize user-authored extensions since the existing framework is designed to optimize only built-in functionality.

We previously developed **Saaz**, a C++ library for authoring *extensible* analysis tools for study of turbulent flow simulations [2]. We attached Saaz to running simulations and were effectively able to increase the time resolution of flow analysis by filtering data based on user-specified criteria, reducing the rate at which simulated data was committed to disk. Saaz's performance was not an issue because the domain scientists did not need to analyze every time step of the simulation: the cost of Saaz's authored analyses was just 10% of the execution time of the simulator.

When we look at offline datasets, however, the picture changes. With offline analysis, there is no simulator to overwhelm the cost of analysis in Saaz. We determined that Saaz incurs overheads from its use of class abstractions that

[1]This work was supported by NSF contract OCE-0835839.

Email addresses: `apking@cs.ucsd.edu` (Alden King), `baden@cs.ucsd.edu` (Scott Baden)

are ubiquitous to C++ class libraries. In this paper, we discuss these sources of overhead and our custom source-to-source translator, called **Tettnang**, that rewrites Saaz applications to eliminate most of the library overheads. Tettnang enables users to benefit from Saaz's abstractions without incurring unacceptable performance penalties.

Tettnang treats the Saaz library as an Embedded Domain Specific Language. This approach separates correctness and efficiency: the library takes responsibility for correctness, the translator handles efficiency. This separation lets users continue to author applications using robust high level APIs while still benefiting from the performance of an implementation tuned to their specific problem(s). With Tettnang, users of the Saaz library can utilize its high-level abstractions essentially for free. While the Tettnang translator is specialized to the Saaz library, other libraries introduce overheads similar to those in Saaz. Tettnang demonstrates a technique which we believe can be applied to other libraries. Some optimizations (identified in Section 4.2) are quite generic and can be applied to other libraries.

In the next section we introduce the Saaz library and the requirements it imposes on implementations. Section 3 illustrates how the requirements of Saaz influence its organization and how the class abstractions used impose penalties on performance. Section 4 introduces Tettnang as the solution to library overheads, followed by performance metrics in Section 5. Finally, we present related work in Section 6 and conclude with future work in Section 7.

2. The Saaz Library

Saaz was designed to study turbulent flow simulations. Such simulations are difficult to characterize because there is no universal definition of turbulent structures, known as vortex cores. Accordingly, there are several popular vortex core eduction methods [2]. Existing flow analysis tools are largely ad-hoc. As requirements change, the tools must be recoded, sometimes at great cost. Relational databases have failed to gain general acceptance by the CFD community because they disrupt locality in tightly wound loop nests [3], and there have been few inroads [1].

Saaz enables the domain scientist to utilize mathematical rules to identify features that are posited to be dynamically important. This approach fundamentally changes the discovery process; instead of asking "do these data fit the model?" we ask: "what kinds of models explain the data?" This change of viewpoint is important: an outstanding difficulty in studying turbulent flow is that algorithms used to understand data are themselves subject to revision. This imposes on Saaz a requirement to support not a fixed interface, but ad hoc queries.

By providing reusable primitives rather than a fixed set of operations, Saaz enables domain scientists to revise queries and even issue ad hoc queries without becoming entangled in tedious implementation details. With Saaz they can formulate a new vortex eduction formula without having to entirely rewrite their analysis infrastructure. Computational science requires experimentation in analysis as much as in simulation. Despite being the status quo in most existing tools, a fixed set of predetermined primitives is not appropriate for CFD research. To support the search for mathematical formulas, Saaz must support customized queries. Customized queries are supported in Saaz through the use of extensive composition of basic array operations. This composition must be efficient and must not incur high runtime overheads.

A common approach to efficient composition which avoids runtime overheads is demonstrated by Expression Templates [4]. Expression templates make use of the fact that C++ templates are Turing-complete, using a C++ compiler to build trees of operations within the template parameters of user-defined types. Aggressive inlining can then yield very efficient composition of these operations, even moving them within loops, flattening many nested function calls into a sequence of basic operations. This sort of template programming, however, remains a poor tool for addressing sequences of operations which are not composed. Optimizing call-sequences requires deferring evaluation until sequences can be refactored (typically by a runtime [5]). Furthermore, code using expression templates can exhibit context-dependent compile-time errors deep in the call stack and far removed from the original user's code which caused it. If the wrong object is passed into a function, an error may not arise until it is passed through many other functions, and finally to a function which actually uses a non-existent attribute (method, data-type, or data-member) of that object. The users of Saaz are domain scientists who do not have extensive programming experience with templates. Their ability and willingness to track down these kinds of errors, let alone understand them, is limited at best. An error reporting that an object does not have a particular attribute does not help the domain scientist realize he accidentally used the wrong variable.

Saaz supports a simple data model: the multidimensional array. Unlike conventional languages, we index a Saaz array with a multidimensional point-valued object, i.e. `A[p]`. A point corresponds to an ordered tuple of indices traditionally used to reference arrays. This notation simplifies subscript expressions [6] [7] [8] and enables the user to write dimension-independent loop bodies, enhancing code reuse.

A Saaz *array* is a set of elements, all of the same type, defined over a bounding box called the *domain* of the array. Each point in the domain of an array is mapped to a stored value (element). Notably, Saaz supports arrays with different in-memory layouts, such as row-major and column-major orders. By comparison, traditional approaches impose a fixed storage layout, which can be heavily optimized. For example, Matlab and FORTRAN use column-major order, while C and C++ use row-major order. Relaxing the assumption of a fixed data layout introduces high overheads both in translating a point-valued subscript into a memory offset, and in iterating over the elements of an array. Still, we found this capability to be useful in flow analysis as it facilitates interoperability, that is, sharing of data products and analysis tools. We were able to take existing analysis code and without modification, run it over another group's dataset which used different conventions.

Despite the apparent overheads of Saaz, we were able to use knowledge about how users formulate queries to eliminate most of the overheads, improving user productivity significantly. We next discuss the overheads in detail, and following that, our translation solution, Tettnang.

3. Overheads

To better understand the performance differences between code written with Saaz and code written as *Vanilla C* (that is, C with C-style 3D arrays and without user-defined abstractions), we categorize different kinds of overhead. Our Saaz example exhibits these overheads, but these categories are not unique to Saaz. Overheads are consequences of building abstractions, particularly with classes, and exist in all other libraries, although their relative costs may vary with implementations and usage patterns. Overheads are determined not solely by what a library provides, but also how it provides it: the library's interface. Our categories are encapsulation, generalization, and isolation:

C1. *Encapsulation* refers to the movement of data into objects (or structures) and the outlining of code into functions or methods. This can introduce function call overhead, pointer arithmetic, inlining issues, duplication of metadata, loss of cache benefits, and indirection.

C2. *Generalization* refers to features which may not be needed in all cases, but are used at different times and in different combinations. This typically introduces extra checks for assumptions, branches for different capabilities, or extra arithmetic operations.

C3. *Isolation* occurs when encapsulation prevents both the compiler and user-code from making global decisions. Code behind a class interface cannot take advantage of global patterns to simplify its calculations. Such code often has to recompute values known to other objects or other parts of the program.

Each of Saaz's abstractions carries a cost. Saaz focuses on usability for domain scientists. As mentioned before, Saaz avoids techniques such as compile-time evaluation with templates that are unreasonable for domain-science focused programmers. A well-designed library must balance usability and performance. If a library is too difficult for use by its intended audience, then it will not be adopted. Yet, if that library is too inefficient, it may still not be adopted. Our goal is to balance these concerns, and we believe that our solution of employing a custom source-to-source translator to mediate the penalties of abstraction in an application library has done so.

Our approach is based on the following observation: high-level abstractions do not necessarily have inefficient implementations; by carrying information about the operation being performed, they enable domain specific optimizations. In these cases, much of the application overhead can be between these specialized calls, when data formats must be changed or user-defined operations are implemented manually. One of the goals of Tettnang is to show how to optimize call-sequences and manually-implemented operations. Saaz tries to be general enough that it can express more complex, problem-specific functions. As a consequence, individual operations in Saaz must be composed or chained together in sequence to express more sophisticated operations. Because each operator in Saaz has a restricted view (C3), it is not able to specialize its operations in the same way that other libraries can (C2). We have therefore designed Tettnang to obtain a global view of operations and target not just the overhead composed calls, but the overheads of call-sequences and their side-effects.

3.1. Array Layout

When we index a multidimensional array, we need to map the multidimensional index onto a scalar offset, a process called *linearization*. The offset corresponds to the position of the indexed data item within the array's storage area. In conventional languages such as FORTRAN, arrays have an assumed layout, and the functional form of the mapping is known by the compiler. However, since Saaz allows the user to specify the layout at runtime and at the granularity of individual arrays, the compiler does not know what function performs the linearization. To support this generalization (C2), the runtime ultimately sets a function pointer to perform the linearization. Invoking a function pointer is costly, however, and increases the cost of an array reference by about a factor of 10.

Fortunately, the overhead of this abstraction is frequently unnecessary. Linearization is a function of an array's layout, and the domain on which it is defined. Saaz does not permit layouts to change, or domains to change shape. In many analysis codes, queries written with Saaz involve *conforming* arrays, that is, arrays which have the same layout and are defined over the same domain. Special-casing the linearization operation (C2) for these arrays makes it cheaper and can eliminate expensive common subexpressions (C3). Tettnang performs this specialization automatically.

3.2. Domain Objects

Domain objects are a form of metadata in Saaz. They encapsulate (C1) the index set for an array, and thus the bounds for looping iterators. While domains are a convenient means of organizing data, storing data in objects still introduces overhead (C1 and C3). Compilers could, in theory, get around this overhead, but in practice the requisite analysis is too expensive, and so compilers can optimize accesses only under certain circumstances. While the individual cost of each operation can be small, the cumulative effect is large when many accesses are made within loops. Using stack-local variables instead of object data-members can increase performance. Hand-implemented code typically contains bounds on the stack, either as local variables, or as function parameters.

3.3. Iterators

```
Domain dmn; int sum = 0;
for(Iterator p = dmn.begin(); p != dmn.end(); ++p)
{ sum += array[p];  }
```

Figure 1: A simple example illustrating a Saaz iterator.

Saaz uses iterators to make loops rank independent and order agnostic. Figure 1 illustrates a simple summation; note that the rank of the array, domain, and iterator is hidden. Saaz uses function calls instead of integer comparisons to evaluate iteration bounds and to increment the iterator, adding encapsulation overhead (C1). Compared to Vanilla C, the iterator does not add comparisons or operations other than a function call.

Perhaps the most significant cost of iterators is that they inhibit compiler optimizations (C1). C++ compilers such as those from GNU and Intel will not vectorize loops over iterator objects. Loops over integers that have well-defined bounds (even if not constant, but an integral variable), can be vectorized by the compiler or parallelized with OpenMP. Tettnang's loop transformations convert loops over iterators to loops over integers, thus enabling these optimizations.

4. The Tettnang Translator

Our translator, Tettnang, was built using the Rose source-to-source translator [9] from Lawrence Livermore National Laboratory. The Rose compiler framework is a convenient means of tuning libraries to specific program requirements. Rose is a member of the family of language processors that support semantic-level optimizations including Telescoping languages [10] [11] and Broadway [12]. It has previously been used to produce promising results in realizing semantic-level optimizations of C++ class libraries [13] [14]. Tettnang uses Rose to parse C++ source code and manipulate it before unparsing it and passing the unparsed C++ source code to a general-purpose C++ compiler.

Which code transformations can be performed by a compiler will largely be a function of the information the compiler has about the program. Keywords such as `restrict` give the compiler information about how pointers behave. We incorporate information about the semantics of Saaz into Tettnang. This means Tettnang does not have to perform interprocedural analysis of Saaz, because it already knows what side effects may or may not occur. Furthermore, Tettnang knows how objects are aliased, and so does not have to view the heap as an "anything goes" zone, as other compilers such as Intel and GNU do. In the future it may be possible to avoid incorporating Saaz-specific semantics by using an annotation language such as that in Broadway.

Several Saaz design choices greatly simplify the need for analysis on the part of Tettnang. Of particular importance is the fact that domains, the domains arrays are defined over, and the layouts of each array are all constant. This greatly reduces the amount of side-effect analysis which must be performed.

This section discusses how we designed Tettnang to utilize information about common use cases of the Saaz library to reduce library overheads. While performance is a big motivation for the transformations, those we discuss here specifically address the overhead introduced by abstractions and not general performance improvements [15] [16]. Tettnang does not, for example, optimize for cache or perform other classic optimizations. These are out of the scope of Tettnang and are assumed to be performed independently.

4.1. Index Linearization

Recall that because Saaz does not have a fixed convention for array layout, that the linearization function for each array object can be different. Because the choice of the linearization function is hidden behind a class interface, general-purpose compilers cannot determine what that choice will be (C2). Tettnang is able to examine the array constructors to identify layout, and thus inline the function call, overcoming Encapsulation (C1) to address the cost of Generalization (C2). For example, if Tettnang can determine the layout of an array, `A`, to be row-major, and its domain to be $[x\_lo : x\_hi] \times [y\_lo : y\_hi] \times [z\_lo : z\_hi]$, then it can save the cost of the expensive function pointer call by rewriting `A.LinearIdx(i,j,k)` as follows:

$$(k - z\_lo) + ((j - y\_lo) + (i - x\_lo) * (y\_hi - y\_lo + 1)) * (z\_hi - z\_lo + 1) \tag{1}$$

In addition to the cost of the function pointer, these expressions contain two expensive integer multiplies and seven additions and subtractions. The costs of these calculations can easily overwhelm the cost of accessing the desired array element, especially for simpler kernels. Multiple array references involving conforming arrays will linearize identically (Figure 2a). Tettnang can therefore consolidate the common linearization expressions and perform them only once (Figure 2b). This will cut running time of Figure 2a in half. With this technique, Tettnang addresses the effects of Isolation (C3), eliminating the common expressions from the linearization operations.

4.2. Domain Objects & Iterators

Encapsulation also increases the cost of obtaining the bounds of a domain, in this case, the cost of the accessor function. While modern compilers can inline accessor function calls, there still remains the extra cost of accessing member variables. Tettnang can determine a domain's bounds by analyzing the domain's construction. Consider, for example, the case of `Domain3 dmn(x_lo,y_lo,z_lo, x_hi,y_hi,z_hi)` (Figure 2a and b). Tettnang can replace calls to `dmn.Min(0)` or accesses to `dmn.x_min` with accesses to the stack-local variable `x_lo` that was passed to the domain constructor. If Tettnang is unable to evaluate the expression as a variable, it performs the next optimization.

Accessing member variables in an object is still more expensive than accessing them from the stack. In our experiments with the GNU and Intel (v12) compilers, we found that the encapsulation of values into member variables seemed to discourage the compiler from placing the values in registers, despite frequent use. Tests indicate it might be because such values live on the heap, although the exact reason for this differentiation is under investigation. Copying member variables onto the stack also increases cache locality with other frequently used variables which may not be adjacent on the heap. In the case where a domain's constructor is not available (for example, if it was created in another function or read from disk) or a constructor parameter is an expression instead of a variable or constant, we can still create new local variables to store the value of the member variable. Henceforth, function calls to read that bound can be rewritten as references to the new local variable (Figure 2c). Other transformations generate member accesses for the data which this optimization caches, so it is important to save this transformation until the end. This method of copying member variables onto the stack can be applied to other libraries that frequently access member variables through accessor functions.

Identification of domain bounds is especially effective when translating iterators. A single loop over an iterator can be replaced with the corresponding triple-nested loops over integers (Figure 2c to Figure 2d).

```
Domain3 dmn(x_lo, y_lo, z_lo,  x_hi, y_hi, z_hi);
Array3 A(dmn, Layout::RowMaj), B(dmn, Layout::RowMaj);
for(Iterator3 p = dmn.begin(); p != dmn.end(); ++p)
  A[p] = B[p];
```

(a) A simple example illustrating Saaz's abstractions.

```
Domain3 dmn(x_lo, y_lo, z_lo,  x_hi, y_hi, z_hi);
Array3 A(dmn,  Layout::RowMaj), B(dmn,  Layout::RowMaj);
for(Iterator3 p = dmn.begin(); p != dmn.end(); ++p) {
  unsigned long offset = A.LinearIdx(p);
  A.Elements(offset) = B.Elements(offset);
}
```

(b) When `A` and `B` are conforming, Tettnang can consolidate their linearization operations, making only one call to `LinearIdx`.

```
Domain3 dmn(x_lo, expr1, expr2,  x_hi, expr3, z_hi);
int y_min = dmn.Min(1), y_max = dmn.Max(1), z_min = dmn.Min(2);
Array3 A(dmn,  Layout::RowMaj), B(dmn,  Layout::RowMaj);
for(Iterator3 p = dmn.begin(); p != dmn.end(); ++p) {
  unsigned long offset = (p.z-z_min)+((p.y-y_min)+((p.x-x_lo)*(y_max-y_min+1)))*(z_hi-z_min+1);
  A.Elements(offset) = B.Elements(offset);
}
```

(c) Tettnang creates local variables to cache the values of member variables which are unknown to it. Expressions which Tettnang cannot analyze are shown as `expr1`, `expr2`, and `expr3`. `LinearIdx` is expanded as in Equation 1.

```
Domain3 dmn(x_lo, expr1, expr2,  x_hi, _expr3, z_hi);
int y_min = dmn.Min(1), y_max = dmn.Max(1), z_min = dmn.Min(2);
Array3 A(dmn,  Layout::RowMaj), B(dmn,  Layout::RowMaj);
for (int i = x_lo; i <= x_hi; ++i) {
  for (int j = y_min; j <= y_max; ++j) {
    for (int k = z_min; k <= z_hi; ++k) {
      unsigned long offset = (k-z_min)+((j-y_min)+((i-x_lo)*(y_max-y_min+1)))*(z_hi-z_min+1);
      A.Elements(offset) = B.Elements(offset);
} } }
```

(d) Tettnang is able to convert the iterator loop to nested loops with integral bounds. Note that accesses to the member variables of the composite index `p` have been replaced with the new loop variables (`i`, `j`, and `k`).

Figure 2: Our code example with progressively more Tettnang transformations applied.

5. Results

5.1. Queries

To validate Tettnang, we applied it to seven queries authored in Saaz (Table 1). These queries were obtained from the CFD group at UCSD that helped develop Saaz initially [2]. They perform correlations and derivatives along different axes, thus demonstrating the three categories addressed in Section 3.

We divide the queries into two types: those that take derivatives, and those that do not. The first query, a *cross-correlation* of velocity, does not take derivatives: we denote it by $\texttt{PlanarAverage}_y\,(u'v')$, where the $\cdot'$ notation signifies normalization, and $\texttt{PlanarAverage}_y\,(\cdot)$ is the planar average taken along the y-axis (Figure 4 shows an example result and the pseudo-code for it). Normalization entails subtracting, from each point, the average value of the field variable in the containing plane normal to the y-axis. Since stratification causes different energy levels for each plane, we use normalization to get the same baseline for each plane. This query measures the prevalence of coherent flow structures: it computes a planar average of the point-wise product of the two normalized field variables, u, and v (in 3D). Thus, the cross-correlation uses a two-level loop, the outer loop iterating along the y-axis, and the inner loop computing the sums of the products within each plane. This is an extension of the planar average operation itself. The turbulent transport queries are mathematically similar, and measure the transfer of energy in certain directions.

Dissipation is the second type of query, and it takes derivatives. Dissipation measures the loss of energy due to turbulence, and is thus important for understanding turbulent systems. Regions with more dissipation tend not to form coherent structures, as the dissipative energies move the fluid apart more than similar velocities move it together.

5.2. Experimental Setup

To evaluate Tettnang, we wrote several different variants of each query. The first variant is written in *Vanilla C*, C code using C-style 3D arrays and no user-defined abstractions. The second variant was written in Saaz; the rest are the result of passing the Saaz code through Tettnang with various levels of optimization. All tests ran on a two-socket

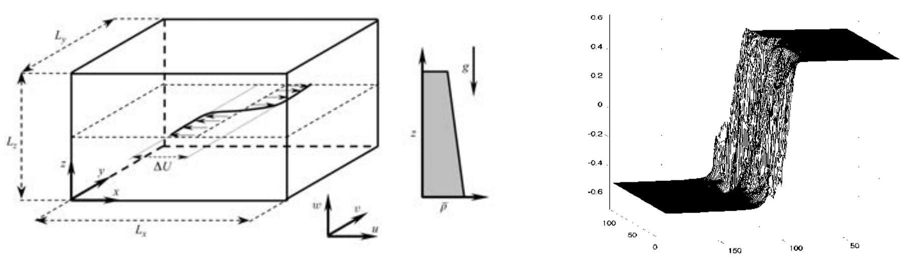
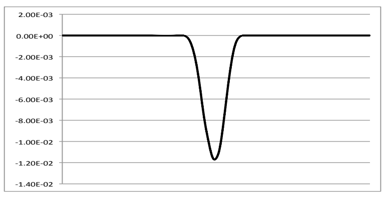


Figure 3: (Left) Configuration of the temporally evolving shear layer. Two streams of fluid are shown, one moving in the positive x direction and the other in the negative x direction. The streams are separated by the $y = 384$ mid-plane. Stratification (mean density variation) is in the z direction, which is shown by the constant density gradient, $d\overline{\rho}/dz$. (Right) Simulation data for the u velocity in the yz-plane.



```
$u_a$ := PlanarAverage_y($u$, domain of $u$))
$v_a$ := PlanarAverage_y($v$, domain of $v$))
$\forall$ Planes $p$ **in** domain of $u$
    $\forall$ Points $q$ **in** $p$
        $uv[p] += ((u[p \oplus q] - u_a[p]) * (v[p \oplus q] - v_a[p]))$
    $uv[p] := uv[p]/size(p)$
return $uv$
```

Figure 4: An example query: the velocity cross-correlation with normalization inlined. $\oplus$ represents the gather from a 1D and 2D point into a 3D point.

3.0 GHz quad-core Intel Xeon processor (X5450, or "Harpertown") with 2x6MB of L2 cache and 32GB of main memory. All code was compiled using the GNU compiler suite v4.2 and command line options `-O2`, although both later versions and the Intel compiler exhibit similar characteristics.

5.3. Optimizations

Tettnang currently implements six optimizations, described previously. These optimizations deal solely with eliminating overheads of abstraction and are not optimizations in the traditional sense. Tettnang does not implement traditional loop-based optimizations such as blocking for cache, loop fusion, loop fission, or vectorization [16]. Such optimizations are orthogonal to Tettnang's and are out of the scope of this paper.

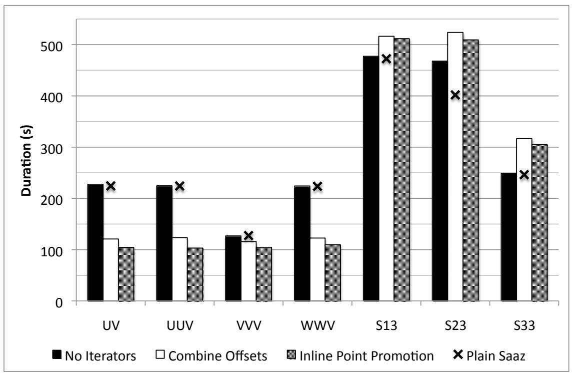
We applied Tettnang's optimizations incrementally to all seven queries (Table 1). Because all arrays are conforming, their linearization functions are identical. The first four queries apply pointwise multiplication to combine array arguments. Since each array is indexed at the same point, their respective linearization computations are all identical, and can be combined. The remaining queries access adjacent points in a stencil operation to compute finite-differences. The patterns of memory accesses (strides) are different, and depend on the direction the derivatives are taken. For example the differentiation operation $\partial_x(u')$ incurs a unit stride, while $\partial_z(u')$ incurs a much larger stride: the size of a plane.

Our *No Iterators* optimization transforms loops using Saaz iterators into ordinary loops over integers. While the Saaz iterators do not themselves have significant overhead, traditional compilers cannot optimize code using Saaz iterators as they would a traditional loop over integers (C1). Vectorization and parallelization (via OpenMP), for example, require knowing how loops are bounded and advanced - facts that an iterator object obscures. As Figure 5a shows, our first optimization on its own yields only a minor performance improvement, but it is required for later compiler optimizations.

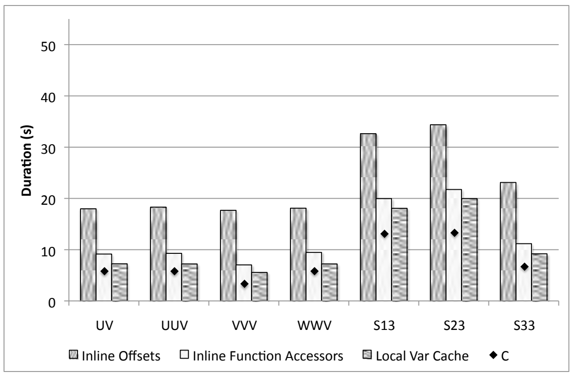
Because different Saaz arrays can have different layouts, each array reference incurs the cost of linearization (C3). By comparison, code written in Vanilla C (or FORTRAN) performs this calculation once per layout, instead of once per array reference. Our second optimization, *Combine Offsets*, performs common subexpression elimination on linearization calls when referencing multiple conforming arrays. Thus, the cost of an array access is more a function

Table 1: Sample query kernels. The first four do not take derivatives, the last three do. The full query is the planar average of the result of the calculation in the column labeled "Kernel" [2]. $\partial_z(A)$ represents the finite-difference derivative of A taken along the z dimension: $\frac{u[p+\vec{Z}]-u[p-\vec{Z}]}{\Delta z}$.

Description (Name)	Kernel
Velocity Cross-Correlation (UV)	$u'v'$
Turbulent Transport X (UUV)	$u'u'v'$
Turbulent Transport Y (VVV)	$v'v'v'$
Turbulent Transport Z (WWV)	$w'w'v'$
xz-Dissipation (S13)	$(0.25 \times (\partial_z(u') + \partial_x(w')))^2$
yz-Dissipation (S23)	$\left(0.25 \times (\partial_z(v') + \partial_y(w'))\right)^2$
zz-Dissipation (S33)	$(0.25 \times (2 * \partial_z(w')))^2$



(a) First 3 Transformations



(b) Second 3 Transformations

Figure 5: Performance of the various queries with incrementally more of Tettnang's transformations applied. Our fourth optimization, *Inline Offsets*, significantly improved performance, requiring a change in scale between the left and right graphs.

of the number of layouts than the number of array references - users pay only for the compatibility functionality if they need it (C2).

Three of the first four queries benefit significantly from combining offsets (Figure 5). Because they access two arrays, their running time is cut approximately in half. (For k conforming arrays, the cost drops to $1/k$.) The `VVV` query (Turbulent Transport Y) doesn't benefit from combining offsets because there is only one array access. The small performance gain is due to an inline performed to set up for combining (apparently non-existent) offsets. The last three queries don't benefit either, but for a different reason. These queries take derivatives: although they index conforming arrays, they do so at different points. The computed offsets will differ, and so the index calculations cannot be shared. The *Inline Offsets* optimization addresses this issue. For reasons of space, we omit a discussion of the *Inline Point Promotion* optimization.

The information Tettnang gathered to combine offsets also identifies the linearization function (pointed to by `LinearIdx`). Tettnang will inline this function and eliminate common subexpressions which had previously been obscured (*Inline Offsets*). The result of inlining the index calculation and combining non-identical offsets is the most significant speedup of any of our optimizations: a speedup of around a factor of ten, hence the difference in scales between Figure 5a and b. Performance has now approached that of the Vanilla C version.

Our two remaining optimizations, *Inlining Function Accessors* and the *Local Variable Cache*, are not specific to Saaz and address overheads of data encapsulation (C1). Domain objects hold integer values for their upper and lower bounds in each dimension. Loop iterations and linearization operations both need to check domain bounds. Many accessor function calls are exposed through the inlining performed by previous optimizations. Tettnang translates accessor calls to the corresponding object member references. We can go further by extracting these values into stack-local variables as a sort of cache. This encourages the compiler to place them in registers, or, when that is not possible (such as in the case of complex kernels), to access them with a single stack offset. This is cheaper than accessing the member through its object.

After applying all of our optimizations, performance is now 75% of Vanilla C, and 27 times faster than the original

Saaz code. If we run all of these queries as a typical query suite, our total time is 74 seconds on a warm file cache. A cold cache adds 122 seconds. While there is still room for improvement, it is now more important to batch queries than to further improve Tettnang's optimizations.

5.4. Multicore Parallelization

Tettnang also generates OpenMP code to enable Saaz queries to run on multicore processors. Since Saaz is intended for data-intensive applications, large shared memory machines designed for such applications are the target of this capability. Our queries currently fit into traditional server nodes, but we will discuss larger scale parallelism in the future [17].

OpenMP has a very restricted set of forms for the loops that it will parallelize [18]. Not until version 3.0 did OpenMP support `stl` iterators. Tettnang effectively extends OpenMP's capability by adding support for Saaz's new iterator datatype: it translates loops with the Saaz iterator to loops over integers, and then inserts the appropriate pragmas. Without any extra effort on the part of the Saaz programmer, Tettnang is able to parallelize serial code.

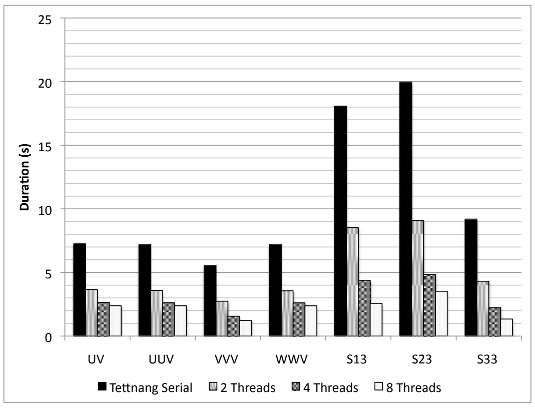


Figure 6: Performance of the various queries with different numbers of OpenMP threads. “Tettnang Serial” refers to the Tettnang-processed code with all the optimizations applied (“Local Var Cache” from Figure 5).

Figure 6 shows the timings for various thread counts using OpenMP directives that have been inserted by Tettnang. These tests are all run on our previously described SMP node. There are some small super-linear speedups which we are still investigating. The last three queries involve derivatives and thus more memory accesses. They also involve more FLOPS and more integer adds and multiplies from the extra offset computations. Because they are more compute-intensive, they see better speedup on more cores.

6. Related Work

In contrast to many other systems that focus on optimizing transformations, the transformations in Tettnang focus largely on removing library overheads. One of the early libraries similarly addressed was the A++/P++ library [19]. In [14], an earlier version of the Rose compiler was used to perform transformations. Unlike Tettnang, which has built-in information, these translators use annotations to identify certain library constructs. Their transformations consist mainly of converting whole-array operations to nested loops. Loop fusion is also performed. Saaz's use of looping constructs instead of whole-array operations makes this transformation less important. Tettnang does not address loop transformations, although the data it gathers would make such transformations straightforward. Many Tettnang transformations would still be applicable after these loop transformations.

Telescoping languages [10] analyze libraries and construct a compiler to optimize them. Their 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 [12] which seeks to address a wide variety of domains through library annotation. 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 accessing object members, but focuses on functions, making it inappropriate for addressing isolation overheads (C3). 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 improved performance.

The power of Tettnang lies is its ability to resolve, at compile time, certain control flow decisions that drastically impact performance. Expression Templates [4] [20] have been able to address composition through aggressive inlining of template methods. In particular, the physics library OpenFoam [21] uses heavily-templatized C++ to write simulators. Template metaprogramming, however, remains a poor tool for addressing call-sequences or annotating objects with state. There has been some work in delayed evaluation libraries [22] [5] which can handle call-sequences.

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. We consider our technique more intuitive and more user-friendly for domain scientists.

7. Conclusion and Future Work

We have identified a bottleneck in scientific data analysis and eliminated it. Saaz provides abstractions to increase programmer productivity, while Tettnang optimizes use of those abstractions to achieve performance comparable to a low-level Vanilla C implementation. The Tettnang translator uses facts about Saaz's semantics to gather knowledge about library use and specialize the otherwise generic library code. Ultimately, Saaz's support for interoperability and focus on higher-level abstractions paid off: Saaz queries which have been optimized by Tettnang achieve performance within 75% of Vanilla C. By customizing a translator, we have enabled the Saaz library to utilize high levels of abstraction with little performance cost.

We leave as future work optimizations that deal with the memory hierarchy, e.g. blocking for cache and loop fusion. Tettnang's infrastructure gathers sufficient information to enable us to perform the analysis required to implement such optimizations, potentially enabling us to overtake Vanilla C. Tettnang's analyses gather enough information to detect a number of array indexing errors, but currently issue no warnings. Tettnang can also be extended to support additional Saaz features such as other array layouts and sparse iteration spaces.

References

[1] E. Perlman, R. Burns, Y. Li, C. Meneveau, Data exploration of turbulence simulations using a database cluster, in: Proceedings of the ACM/IEEE SC2007 Conference, SC, ACM, New York, NY, USA, 2007, pp. 1–23.

[2] A. King, E. Arobone, S. Sarkar, S. B. Baden, The saaz framework for turbulent flow queries, in: Proceedings of the 2011 IEEE conference on e-Science, IEEE, 2011.

[3] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, G. Eyink, A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence, arXiv.org (2008) 1–31.

[4] T. Veldhuizen, Expression Templates, SIGS Publications, Inc., New York, NY, USA, 1996, pp. 475–487.

[5] A. R. Craig Chambers, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, N. Weizenbaum, Flumejava: Easy, efficient data-parallel pipelines, in: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI, ACM, New York, NY, USA, 2010, pp. 363–375.

[6] P. N. Hilfinger, P. Colella, FIDIL: A Language for Scientific Programming, SIAM, 1989.

[7] S. J. Fink, S. B. Baden, S. R. Kohn, Efficient run-time support for irregular block-structured applications, Journel of Parallel and Distributed Computing 50 (1998) 61–82.

[8] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken, Titanium: a high-performance java dialect, Concurrency: Practice and Experience 10 (11-13) (1998) 825–836.

[9] D. Quinlan, D. Miller, B. Philip, M. Schordan, Treating a user-defined parallel library as a domain-specific language, in: Proceedings of the 16th international Parallel and Distributed Processing Symposium, Vol. 2 of IPDPS, IEEE Computer Society, Washington, DC, USA, 2002.

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

[11] B. Broom, R. Fowler, K. Kennedy, Kelpio: A telescope-ready domain-specific i/o library for irregular block-structured applications, in: Proc. First IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE, 2001, pp. 148–155.

[12] S. Z. Guyer, C. Lin, An annotation language for optimizing software libraries, ACM SIGPLAN Notices.

[13] D. J. Quinlan, M. Schordan, Q. Yi, B. R. de Supinski, A c++ infrastructure for automatic introduction and translation of openmp directives, in: Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming, WOMPAT, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 13–25.

[14] D. Quinlan, M. Schordan, R. Vuduc, Q. Yi, Annotating user-defined abstractions for optimization, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 2006, p. 8.

[15] M. E. Wolf, M. S. Lam, A loop transformation theory and an algorithm to maximize parallelism, IEEE Transactions on Parallel and Distributed Systems (1991) 452–471.

[16] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann for Academic Press, London, UK, 2002.

[17] A. King, The saaz library and compiler, Ph.D. thesis, University of California, San Diego, La Jolla, CA, USA (2012).

[18] L. Dagum, R. Menon, Openmp: An industry-standard api for shared-memory programming, IEEE Computational Science and Engineering 5 (1998) 46–55.

[19] R. Parsons, D. Quinlan, A++/p++ array classes for architecture independent finite difference computations, in: Proceedings of the Conference on Object-Oriented Numerics, OONSKI, 1994, pp. 408–418.

[20] J. de Guzman, D. Marsden, T. Heller, Boost::phoenix.

[21] H. Jasak, A. Jemcov, Željko Tuković, Openfoam: A c++ library for complex physics simulations, in: Coupled Methods in Numerical Dynamics, 2007, pp. 1–20.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, in: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Eurosys, ACM, 2007, pp. 59–72.