Improving the Computational Intensity of Unstructured Mesh Applications

Brian S. White, Sally A. McKee
Computer Systems Lab
Cornell University

Bronis R. de Supinski, Brian Miller,
Daniel Quinlan, Martin Schulz
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

ABSTRACT

Although unstructured mesh algorithms are a popular means of solving problems across a broad range of disciplines—from texture mapping to computational fluid dynamics—they are often dominated not by computation, but by mesh overhead. Our study of an object-oriented mesh-based benchmark reveals that 72% of its execution time is spent on mesh-related operations, such as iterating over faces or chasing pointers. We report a series of optimizations—some traditional, some novel—that dramatically improve the benchmark's computational intensity—the ratio of floating point operations to memory accesses. This improvement is attributable to an eight-fold reduction in memory operations and results in a 4.7× speedup in execution time.

Our work demonstrates that common subexpression elimination and code motion are important optimizations for mesh-based codes. However, conservative analysis prevents their application. We discuss these barriers to analysis and argue that an understanding of mesh semantics complements more traditional analyses, such as pointer alias analysis, and certifies the correctness of these optimizations. Our identification of overheads in mesh-based codes, optimizations that address them, and limitations of current compiler analyses are required for our eventual goal of automating these optimizations in a semantics-aware compiler.

1. INTRODUCTION

The flexibility of unstructured meshes, or unstructured grids, is reflected in their application across a wide cross-section of important scientific challenges: meshes facilitate the study of gravitational collapse to black holes and are used to simulate blood flowing through the heart. Even within a narrow domain, mesh solvers can be parameterized according to equation (e.g., Navier-Stokes or Euler); assumptions (e.g., the ideal gas law or van der Waals equation of state); and fluid (e.g., water or a monatomic gas).

Leveraging this flexibility to provide a problem-independent framework that can be reused across physical simulations requires a code base that is generic, modular, and extensible to the specifics of the domain in terms of mesh structure, boundary conditions, operators, and physical quantities represented as fields. Any general framework intended to encompass such an extensive application space will be large: even the simplified `KOLAH` mesh framework, studied in this work as a representative of libraries used in Lawrence Livermore National Laboratory's more extensive production codes, has 282 files and 68,000 lines of code. These considerations are best addressed by an object-oriented design and implementation.

With their heavy use of indirect addressing and pointer chasing, unstructured grid codes are highly sensitive to memory performance [10]. Unlike structured grids, which use a regular spatial decomposition that is easily traversed by a stencil, unstructured grids employ an irregular mesh to cover a volume using geometric mesh entities, including faces, edges, and nodes. Object-oriented implementations exacerbate poor memory performance through the additional indirection induced by object-based indexing of fields, such as momentum and pressure: where an imperative approach uses a `for` loop with an integer induction variable to access an array, these codes dereference mesh entity iterators and then use the entity to index into field abstractions.

The resulting mesh overhead is significant. A hydrodynamic simulation within the `KOLAH` framework executes nearly one and a half branches per floating point instructions and nine times as many loads. In contrast, `mesh`, an unstructured grid benchmark written in C and popular in the literature [7], has a relatively high computational intensity: nearly one floating point instruction per memory access and 100 floating point instructions per branch.

Table 1 shows the execution-time distribution of a benchmark built within the `KOLAH` framework. We group statements into three categories—mesh, memory, and computation—decomposing compound statements as required. Computation statements are arithmetic operations on local variables. Memory statements access operands for computation statements or write their results to memory. Mesh statements represent mesh-related overhead: iteration, iterator dereferencing, and pointer chasing. The table shows that mesh overhead dominates `KOLAH`'s execution time: 72% of time is dedicated to mesh-related operations, while computation consumes only 13%.

This significant overhead may seem surprising given the efficiency of STL and similar libraries that are the foundations of `KOLAH`. Though the implementation of underlying abstractions is highly tuned, optimizing across library invocations

ICS'05 June 20-22, Boston, MA, USA.

Statement Type	Percent of Execution Time
Mesh	72 %
Memory	15 %
Computation	13 %

Table 1: Distribution of execution time.

requires contextual information unavailable during library development. Even after inlining abstractions to view this context, compiler analysis is often too conservative to certify the safety of many relevant optimizations, such as loop fusion [21]. Our remedy is not to pursue more sophisticated pointer alias analysis, but to exploit higher-level, expressive semantic information. High-level knowledge of an iterator, for example, can transform it from an unintelligible collection of pointers to a well-defined abstraction with specific semantics that complement traditional, fine-grained analysis [28].

We leverage the following semantics of unstructured grids in general, and `KOLAH` in particular, to manually introduce safe optimizations that ameliorate mesh-induced overhead:

- Object-based field indexing is side-effect free and context-insensitive;
- Mesh iterators do not revisit elements;
- The mesh is static.

The third semantic assumption can be relaxed, as discussed in Section 6.

We exploit these features to improve the computational intensity of a `KOLAH`-based benchmark. When applied on a POWER3, our optimizations provided a 6.5× reduction in the ratio of memory accesses to floating point operations, yielding a ratio of 2.1, and a nearly 5.4× reduction in the ratio of branches to floating point operations. We achieve a 3.6× improvement in Mflips or millions of floating point instructions plus fused multiply-adds per second. These factors contribute to an overall 4.7× speedup in total execution time with respect to the `KCC` front-end compiler used in tandem with IBM's `xlc` compiler. These results are not particular to this system or compiler: the same optimizations achieve a 4.9× speedup over the baseline when both are compiled with `gcc` 3.3.3 `-O3` on a Pentium 4.

A compiler armed with semantic knowledge of meshes could attain similar performance results. With respect to our ultimate goal of automatically introducing semantics-aware optimizations within the ROSE compiler framework [21], this paper makes several significant contributions.

Quantifying mesh overhead in a representative unstructured grid library. Given our experience with unstructured grid technology, we believe that `KOLAH` is not exceptional in its high ratio of mesh overhead to computation, but is an exemplar of object-oriented mesh-based libraries.

Identifying barriers to compiler optimization. The pointer-rich abstractions used in meshes present barriers to compiler analysis. Section 3 discusses how semantic knowledge overcomes these barriers and enables automation of optimizations targeting mesh overheads.

Identifying traditional compiler optimizations important to mesh-based codes. Section 5 shows that two well-known optimizations—code motion of conditionals and common subexpression elimination—provide a 46% performance improvement. Identifying those traditional optimizations most likely to impact unstructured grids allows us to determine how best to complement existing analysis within a compiler.

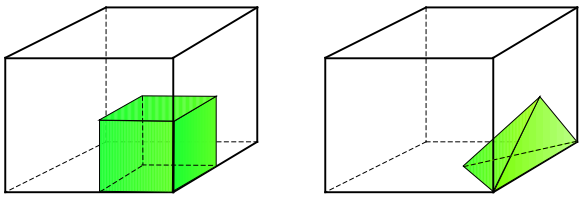

Figure 1: Mesh entities: corner (left) and side (right) within a zone.

Introducing novel optimizations. Our observed mesh overheads motivate a series of novel optimizations that improve computational intensity and overall performance:

- *Mesh precomputation:* Physical quantities, such as volumes, change every time step, but mesh connectivity information, such as the list of mesh edges, does not. Computing these static quantities once during application setup and storing them for subsequent access relieves the code from recomputing them at each time step, and results in an additional 19% reduction in execution time, as shown in Section 6.1.
- *Iteration-space narrowing:* Section 6.2 proposes a technique to extract side-effect free function calls from a loop if they are executed repeatedly with the same inputs. Embedding them in a new iteration space that avoids this repetition results in a 2× performance improvement.
- *Iteration-space partitioning and loop specialization:* Many static mesh properties, such as an edge's nearest neighbors, are frequently evaluated in conditionals. Section 7.2 shows that by partitioning an iteration space according to each element's response to such a conditional, we can create partially evaluated versions of a loop without the conditional that iterate over a subset of the original iteration space. The outcome of the conditional is implied by the partitioned iteration space without being explicitly evaluated.
- *Packing order confliction resolution:* The ordering induced by data packing prevents iteration-space partitioning and loop specialization. We believe we are the first to address conflicting order preferences. In so doing, we obtain the benefits of both optimizations—a reduction in dynamic instructions and a lower average load latency—for an additional 14% improvement.

2. A MESH FRAMEWORK

The versatility and scientific importance of meshes have spurred development of libraries that encapsulate their abstractions [4, 20]. These libraries manage connectivity information between the mesh entities that discretize a volume. They provide interfaces to traverse mesh entities and to access data stored in fields, which sample continuous physical quantities at discrete points corresponding to mesh entities.

`KOLAH` is a framework that provides such functionality to facilitate benchmarking numerical methods on arbitrary polygonal and polyhedral meshes. Its design was motivated by the classes and patterns used in production codes at Lawrence Livermore National Laboratory. It relies upon a generic mesh interface and provides reference mesh implementations along with a variety of mesh utility functions. `testhydro` is a

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

Figure 2: Iteration over zones.

benchmark using these facilities to solve the Euler equations using the Lagrangian method and an ideal gas law equation of state.

Numerical algorithms are written to KOLAH's generic mesh interface, enabling underlying mesh implementations to be evaluated without rewriting the algorithm for each mesh instance. KOLAH interprets a variety of input mesh specifications, representing each as a class. After reading the input mesh, a compatibility layer converts and copies data from the underlying mesh implementation to a common mesh form.

The generic polyhedral mesh interface provides geometric mesh entities including zones, sides, faces, corners, edges, and nodes. To provide an intuitive understanding of mesh entities, Figure 1 shows a zone as it would appear in a rectilinear mesh, along with a representative corner and side. In general, a zone is a three-dimensional subvolume used to partition the volume discretized by the mesh; it needn't be a cube. A zone volume is itself subdivided into three-dimensional corners. A corner corresponds to each zone node and also has as vertices the zone center, the face centers of all faces containing the node, and the edge centers of all edges containing the node. A zone volume may also be subdivided into three-dimensional sides, whose vertices are the zone center, a face center, and two nodes of an edge lying in that face. Finally, faces are two-dimensional entities that cover a zone's surface.

The mesh interface maintains connectivity information, allowing accesses via STL-like iterators, as shown by the code in Figure 2 which iterates over zones. Mesh entity abstractions provide similar iterator access to neighboring entities. For example, given a node, one can iterate over all zones containing it using data contained in the node object. This fairly complete connectivity information provides great flexibility in writing numerical algorithms. The figure also shows the data access paradigm popular in KOLAH. It dereferences a `ZoneIterator` to obtain a zone object and uses that object to index a field, such as the pressure field `P`.

3. BARRIERS TO AUTOMATION

Because user-defined abstractions, such as STL-based iterators and mesh entities, are not part of the base language, compilers do not recognize them and so are not aware of their high-level properties. Instead compilers rely on a myopic approach that cobbles together alias and side-effect information on those parts of the abstraction that are defined in the base language. In some cases, this task is futile because source code is not distributed with libraries implementing the abstractions. When source code is available, a potentially costly global analysis may be needed to examine both the use of the abstractions and their implementation. Finally, without any *a priori* understanding of abstractions, compiler analysis is often too conservative to apply transformations to them.

The simple loop shown in Figure 2 displays both the need for optimization in mesh-based applications and the inherent difficulties a compiler faces in performing those optimizations. This section shows that traditional compiler analyses are inadequate to determine the correctness of applying common subexpression elimination and iteration-space reordering to this loop. We discuss simple semantics of mesh abstractions in general, rather than of KOLAH in particular, that would enable these optimizations.

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

A number of remedies would enable common subexpression elimination. Moving the implementation of `getID` to its header file would allow inlining and would overcome the immediate barrier to applying the optimization. Annotating `getID`'s prototype with a declaration that it is side-effect free would have the same effect. However, these approaches are intimately tied to the implementation of field addressing in KOLAH. Other mesh libraries are likely to offer the same, side-effect free style of access, but are unlikely to employ `getID` in doing so; the tedious, iterative round of discovering and annotating any functions which cannot be inlined will need to be done anew for each application.

Our solution recognizes fields as an abstraction common across mesh libraries. Doing so allows us to imbue them with semantics that carry over from one implementation to another: object-based indexing, the overloaded indexing of a field with a mesh entity, is a side-effect free operation dependent only on its mesh entity argument. This expressive statement enables common subexpression elimination since the compiler is confident nothing within the loop modifies the `ZoneIterator`.[1]

Nested loops in `testhydro` access zones in a non-strided manner. Packing the zones can mitigate the effects of such accesses by rearranging their memory layout. Unfortunately, reordering data to benefit nested loops leads to computational reordering of the loop in Figure 2. In order to reorder this loop safely, a compiler must disambiguate the loop's pointers to guarantee that there are no loop-carried dependences. Since alias analysis is unable to determine the uniqueness of each element in the iteration space, the compiler must assume the iterator is non-trivial and may repeatedly access a zone to create a flow dependence on `e`. This dependence prevents loop reordering. Fortunately, the simple assertion that mesh iterators do not revisit elements ensures there is no such dependence. These semantics complement side-effect analysis, allowing it to determine that the loop may be reordered.

4. METHODOLOGY AND BASELINE

To quantify the importance of the semantics outlined in the introduction, we apply the optimizations they enable to the computational core of the KOLAH-based hydrodynamics benchmark `testhydro`. The results are reported as averages over three runs of ten time steps and were collected using the

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

Processor	375 MHz POWER3-II
Integer Units	3
Floating-point Units	2
Peak IPC	8
Peak Floating-point IPC	4
Peak Mflops	1500
Registers	32
Data TLB	128 entries
Data TLB Associativity	128-way
L1 Data Cache Size	32 KB
L1 Instr Cache Size	32 KB
L1 Latency	1 cycle
L1 Line Size	128 B
L1 Associativity	128-way
L2 Size	8 MB
L2 Associativity	4-way
L2 Latency	7-10 cycles
Memory	16 GB
Memory Latency	20-50 cycles

Table 2: Hardware specification.

IBM HPM hardware performance monitor Tool Kit [5]. Due to the duration of production runs, we do not account for application, mesh, or optimization configuration time, which we expect to be amortized over many time steps. We collected the results while running on the dedicated POWER3 node described in Table 2 [25]. In addition to the architectural features listed above, the POWER3 implements a hardware-based prefetch engine that detects sequential instruction and data accesses and prefetches up to four streams simultaneously.

We compile `testhydro` with the `KCC` front-end optimizing compiler, passing it the `+K3` optimization flag to instruct it to perform branch simplification, loop unrolling, small object optimization, and function inlining. `KCC` produces intermediate C code that is compiled by IBM's `xlc` back-end compiler. We pass `xlc -O2`, as well as `arch=pwr3` to enable POWER3-specific optimizations and `ignerrno` to allow the compiler to emit the `sqrt` instruction. The input data set, `ellipsoid`, is the largest provided with `KOLAH`, with 70K zones, 383K faces, 530K corners, 1.5M sides, 195K edges, and 66K nodes.

Tables 3 and 4 summarize the results. Table 3 lists the results relative to the baseline run and Table 4 shows the raw data as reported by the HPM Tool Kit. Each column provides results with respect to the optimization it names. Entries in Table 4 are counts reported in millions of instances, except load latency, which is in cycles; memory stalls, which is in millions of cycles; percent idle cycles, which is a percentage; elapsed time, which is in seconds; and computational intensity, which is a ratio. In most cases, a reduction in the metric reported indicates an improvement due to optimization, such as with number of load instructions. In these cases, the entries in Table 3 list the baseline result divided by the optimized result. In the few cases in which an increase signals an improvement (e.g., computational intensity), we instead report the optimized result divided by the baseline. Thus, any entry greater than one indicates that an optimization provided some benefit to the respective metric.

The optimizations are generally applied serially, from left to right across the table heading. This is not the case for the three data and computation reordering optimizations: data packing, iteration-space partitioning, and multiple-constraint reordering. Because these are conflicting optimizations, they are alternatives; we apply each on top of the iteration-space narrowing optimization.

```
for (SideIterator sideIt = mesh->sides();
     sideIt != NULL; ++sideIt)
{
  // extract the edge from the side
  EdgeIterator edgeIt = sideIt->edge();

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

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

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

  rho1 = volumeWeightedAvg(zoneMass,node1It);
  rho2 = volumeWeightedAvg(zoneMass,node2It);

  ZoneIterator zoneIt = sideIt->zone();
  soundSpeed = sqrt( inGamma * pressure[*zoneIt] /
    max(rho1, rho2) );

  map<int,<pair<int,int> >::iterator mapIt =
    edgeMap.find(edgeIt->getID());

  int leftEdgeIndex = (*mapIt).second.first;
  if( leftEdgeIndex > -1 ) {
    // ...
  }

  // ... mem ops, method calls, sqrts, etc.

  int signDotProd = sign * S.dot(deltaV);
  if (signDotProd < 0.0) {
    edgeForcing = doPhysics(deltaV, ...);
  } else {
    edgeForcing.Zero();
  }

  // write back the results
  nodeForcing[*node1It] += edgeForcing;
  nodeForcing[*node2It] -= edgeForcing;
}
```

Figure 3: Iteration over sides.

Table 4 presents the baseline results. Notice the relative paucity of floating point instructions: they constitute only 3% of the total instructions and are nearly 1.5 and 14 times less plentiful than branches and memory operations, respectively.

5. TRADITIONAL OPTIMIZATIONS

The following subsections describe optimizations well known in compiler optimization lore, but inapplicable in `KOLAH` for lack of sufficient analysis. In particular, the compiler is unable to perform code motion and common subexpression elimination because of its inability to guarantee that object-based indexing does not produce side effects. With our knowledge that this operation is side-effect free, we are safely able to apply the optimizations manually.

5.1 Code Motion of Control Structures

The code fragment in Figure 3 calculates an artificial viscosity and is one of several dominant loops in the `testhydro`

Metric	Code Motion (Section 5.1)	Common Sub-expression Elimination (Section 5.2)	Mesh Pre-computation (Section 6.1)	Iteration-space narrowing (Section 6.2)	Data Packing (Section 7.1)	Iteration-space Partitioning (Section 7.2)	Multiple-constraint Reordering (Section 7.3)
Load instrs	1.13	1.62	2.15	6.62	6.62	8.37	7.46
L1 load misses	1.06	1.08	1.21	1.50	1.51	1.70	1.59
L2 misses	1.00	1.02	1.15	1.11	1.12	1.12	1.12
Load latency	0.98	0.78	0.70	0.39	0.44	0.28	0.41
Loads & stores	1.14	1.59	2.08	7.30	7.30	9.42	8.40
Useful prefetches†	1.01	0.97	0.76	1.37	1.80	0.74	1.68
Memory stalls	1.19	1.35	1.48	2.86	2.88	3.29	3.00
Branches	1.27	1.69	2.33	6.84	6.83	7.54	7.22
Mispredictions	2.18	1.86	3.96	4.86	4.73	5.18	4.69
Flips	1.10	1.07	1.05	1.30	1.30	1.37	1.29
Mflips/s†	1.08	1.37	1.71	3.10	3.35	2.93	3.61
Instr completed	1.17	1.61	2.13	7.32	7.32	9.30	8.42
Comp intensity†	1.03	1.49	1.97	5.62	5.62	6.86	6.47
Elapsed time	1.19	1.46	1.80	4.02	4.34	4.01	4.67

Table 3: Reduction across metrics after optimization with respect to baseline. †In cases where an increase is desirable, the entry represents the optimized result divided by the baseline result.

Metric	Baseline (Section 4)	Code Motion (Section 5.1)	Common Sub-expression Elimination (Section 5.2)	Mesh Pre-computation (Section 6.1)	Iteration-space narrowing (Section 6.2)	Data Packing (Section 7.1)	Iteration-space Partitioning (Section 7.2)	Multiple-constraint Reordering (Section 7.3)
Load instrs	73065	64691	44999	34024	11033	11039	8728	9791
L1 load misses	885	832	821	733	592	587	519	558
L2 misses	348	346	342	303	314	310	310	310
Load latency	1.24	1.26	1.59	1.78	3.21	2.81	4.42	2.99
Loads & stores	110491	96909	69458	53194	15126	15129	11730	13156
Useful prefetches	83	84	81	63	114	149	61	139
Memory stalls	7856	6599	5829	5295	2745	2725	2389	2614
Branches	11581	9119	6843	4977	1693	1694	1536	1604
Mispredictions	585	268	315	148	120	124	113	125
Flips	8042	7314	7550	7652	6208	6207	5872	6211
Mflips/s	12.80	13.88	17.59	21.93	39.71	42.83	37.45	46.16
Instr completed	238519	204317	147905	111973	32598	32598	25651	28325
% idle cycles	51.98	52.01	56.76	59.79	75.24	73.40	80.38	75.16
Comp intensity	0.07	0.07	0.11	0.14	0.41	0.41	0.50	0.47
Elapsed time	628.40	526.79	429.21	348.99	156.35	144.95	156.77	134.55

Table 4: Raw data. Entries are counts in millions, except load latency, which is in cycles; memory stalls, which is in millions of cycles; percent idle cycles, which is a percentage; computational intensity, which is a ratio; and elapsed time, which is in seconds.

time step. Careful inspection of this loop reveals that the conditional of line 39 could be evaluated earlier. The argument to the conditional is evaluated at line 38, but the dependences of that statement are computed by line 18. The ellipsis abstracts 23 intervening statements; these include memory accesses, another conditional, method invocations, and long-latency square root instructions. Lifting the conditional and its dependent instructions, such that they are evaluated as early as is feasible, avoids the unnecessary execution of the independent instructions when the condition is false.

In order to lift the conditional so that it follows line 18, compiler analysis needs to determine that all intervening statements are side-effect free. `KCC` inlines `volumeWeightedAvg`, so that the only obstacles are the overloaded indexing operator of line 25 and the invocation of STL's `find` on line 28, which can not be completely inlined to determine its side-effect behavior.[2] Semantics of object-based field indexing and an additional annotation on `find` instruct the compiler that both are side-effect free and that it may safely perform the optimization.

Of the 1,500,000 total iterations of the loop, lifting the conditional results in fewer dynamic instruction executions in the 183,000 iterations in which the conditional evaluates to false. As shown in Table 3, the 17% reduction in dynamic instructions translates to a 19% performance improvement.

5.2 Common Subexpression Elimination

`KOLAH` frequently dereferences iterators to access physical fields, as illustrated by the code fragments in Figures 2 and 3. The brevity of mesh field access reflects the elegance of the object-oriented interface, not the simplicity of its implementation: dereferencing an iterator involves three method invocations, evaluating a conditional, three member field accesses, and at least two pointer dereferences; object-based

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

indexing leads to five method invocations, two pointer dereferences, and one array access.

The code of Figure 3 dereferences `node1It` three times: on lines 13, 18, and 46. The tight loop of Figure 2, with its repeated dereferencing of the same iterator variable, is a pattern that occurs even more frequently. We manually apply common subexpression elimination to such dereferences; by leveraging the mesh field semantics discussed in Section 3, we pledge that the expression is constant throughout the loop, a guarantee the compiler is unable to make.

Eliminating the dereferences does not address the more costly aspect of field access: the overhead induced by object-based indexing. Object-based indexing extracts an integer `MeshID` from the object through a series of method invocations that culminates in a call to `getID`. This integer ultimately indexes into a vector representing the field. Because indexing a field with an object is logically equivalent to indexing it with the `MeshID`, we replace the iterator deference by an invocation of `getID`. Rather than eliminating the deference as above, we may now eliminate the call to `getID` to remove additional overhead: `getID` executes two loads in addition to the three loads required by iterator dereferencing. This transformation sacrifices the expressive power of a mesh-independent construct for the efficiency of one that is intimately aware of mesh internals. Automating this optimization within a semantics-aware compiler would improve performance without imposing on the programmer.

The convenience of field access comes at the expense of eight loads. Aside from the access to the underlying array and the five accesses required to satisfy the dereference and object-based indexing, sampling a single value from a mesh field requires two additional accesses to calculate the array base: one to negotiate `KOLAH`'s field abstraction and the other to traverse a shared pointer abstraction. These last two accesses are loop-invariant expressions that could be hoisted from the loop were it not for the single, opaque invocation of `getID` which prevents the compiler from recognizing the safety of this operation. Because it involves eliminating mesh field internals that are not visible until being inlined, we do not manually apply this optimization. These array base calculations will be eliminated once we have automated semantics-based optimizations, since our research compiler performs inlining.

We explore an alternate means of removing the two accesses involved in the base address calculation, as well as the one remaining invocation of `getID`, that exploits the common motif represented by the loops of Figures 2 and 3: iteration over all mesh objects of a particular class. Where elsewhere an iterator hides potentially non-strided access, here the iteration is a simple traversal over every mesh element. A local side-effect analysis, complemented by our semantic assertions that object-based indexing is side-effect free and that mesh iterators do not revisit elements, ensures us that there are no loop-carried dependences and that the loop may be reordered. This allows us to replace the opaque iterator with an integer induction variable that covers the same set of elements, though possibly in a different order. The integer induction variable eliminates the invocation of `getID`, which makes it safe to hoist the calculation of the array's base address out of the loop.

Table 3 summarizes the results of these techniques, all of which exploit mesh semantics to ensure correctness where compiler analysis is too conservative. Their additional 14-30% reduction in executed instructions, loads, and branches leads to a 19% improvement over code motion and a cumulative 46% improvement over the baseline.

6. EARLY EVALUATION

`KOLAH`'s design stresses flexibility over performance: its interface facilitates numerical programming by providing convenient access to mesh entities, but also encourages unnecessary and redundant computation. Section 6.1 shows that re-evaluating static mesh connectivity information during traversal is wasteful, but avoidable. Section 6.2 discusses *narrowing* an iteration space that re-computes expensive operations to one that avoids this redundancy.

6.1 Mesh Precomputation

Section 5.2 found that dereferencing a mesh iterator entails three memory accesses. Such overheads are often incurred when traversing a mesh entity as an intermediary when accessing an entity of a different type. For example, the code for the gradient operator shown in the top third of Figure 4 iterates over zones and faces to reach connected sides without otherwise accessing data associated with the faces. Such overhead is avoidable.

Unstructured grid codes retain a static mesh throughout their execution. Adaptive mesh-based schemes reconstitute a mesh automatically when accuracy falls to unacceptable levels, but also hold the mesh static across a large number of iterations. This static property allows us to evaluate mesh connectivity metadata prior to performing computation over the mesh, during application initialization or immediately after remeshing in an adaptive scheme. This avoids unnecessary iteration or pointer chasing needed to recompute static properties, including the mesh entity interconnectivity of the gradient operator and the relationship of an edge to a side as expressed in line 5 of Figure 3.

Figure 4 shows our intuitive approach to mesh precomputation. `setupGrad` mimics the original loop structure of `grad` to precompute and store for subsequent retrieval only those target mesh entities needed for the calculation, rather than those intermediate mesh entities required to access the target objects. In the case of `grad`, this requires storing the zone and nodes associated with a side in a vector, but does not require storing the face. `precomputedGrad` transforms the original three, perfectly nested loops into a single loop and accesses the stored objects linearly from the vectors to avoid the loop and indirection overheads inherent in mesh traversal.

Table 3 shows that precomputing mesh connectivity provides a 19% improvement over the semantic-aware optimizations of the previous section. Like those optimizations, it significantly reduces loads; however, a larger fraction of those loads removed by precomputation were responsible for L1 and L2 misses. The 2x reduction in branch mispredictions indicates the high overhead of mesh traversal and the large gains attained by avoiding it.

6.2 Iteration-space Narrowing

The connectivity between `KOLAH`'s mesh objects provides latitude in object traversal and algorithm design. Computation operating on all mesh entities of a given type, such as the code in Figure 2, is best implemented as a simple iteration over those elements. Other computation, including the program fragment of Figure 3, is a complex function of multiple iteration spaces—those ranging over zones, side, edges, and nodes. In such cases, the choice of iteration space or

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

void setupGrad(Field& field, Mesh& mesh) {
   ZoneIterator zi;
   for (zi = mesh.ZoneBegin();
        zi != mesh.ZoneEnd(); ++zi) {
      FaceIterator fi;
      for (fi = zi->faceBegin();
           fi != zi->faceEnd(); ++fi) {
         SideIterator si;
         for (si = fi->sideBegin();
              si != fi->sideEnd(); ++si) {
            sideIts.push_back(si);
            zoneIts.push_back(*si->zone());
            node1Its.push_back(*si->node1());
            node2Its.push_back(*si->node2());
         }
      }
   }
}

void precomputedGrad(Field& field, Field& grad) {
   for (int i = 0; i < zoneIts.size(); ++i) {
     SideIterator si = sideIts[i];

     Vector ds = field[zoneIts[i]] * si->areaNormal();
     grad[node1Its[i]] += ds;
     grad[node2Its[i]] -= ds;
   }
}
```

Figure 4: Gradient operators.

spaces is not obvious since one mesh entity domain can be reached from any other through their interconnections. This flexibility allows a programmer to implement all operations involving a logical computation within a single loop, rather than distributing them over multiple iteration spaces.

While such flexibility facilitates scientific programming, the resulting implementation is potentially inefficient. The lack of a bijection between mesh entity domains means that iteration spaces that uniquely visit sides, for example, may revisit any mesh entity they access from a side. Such is the case in Figure 3 which revisits nodes, since the same node may be associated with different sides. The original loop structure makes 3,000,000 invocations of `volumeWeightedAvg`, though only 66,351 of those invocations access unique nodes. Thus the vast majority of these calls incur the unnecessary loop and memory accesses of `volumeWeightedAvg`.

Iteration-space narrowing eliminates redundancy by extracting a function that is re-evaluated with the same arguments and executing it within an iteration space that uniquely visits those arguments. In the above example, iteration-space narrowing instantiates a loop that iterates over nodes, invokes `volumeWeightedAvg` on each, and memoizes the results for subsequent access in the original loop. Memoizing results of `volumeWeightedAvg` within the original loop is less efficient, because it requires a conditional to check whether the result has already been calculated. Such branches degrade performance directly by introducing pipeline stalls and indirectly by complicating compiler- or hardware-directed prefetching.

Table 3 shows that iteration-space narrowing is the most powerful optimization we consider. By eliminating the many repeated invocations on `volumeWeightedAvg`, it provides a 3× reduction in executed instructions and loads over the previous mesh precomputation optimization. The result is a doubling in performance.

The legality of the transformation follows directly from the mesh semantics we use above to allow code motion and to determine that loops are side-effect free. More interesting is the question of profitability; we offer an approach to infer the presence of redundant execution automatically based on a knowledge of mesh semantics.

To recognize that elements are being revisited we need to characterize the domain of a loop nest and of the statement succinctly. Ahmed *et al.* [1] describe a statement iteration space, based on the loops surrounding a statement, that characterizes the dynamic instances of a statement as a set of points in an iteration space induced by affine functions of the loop indices. Since iterators introduce non-affine expressions into loops, this approach is not applicable. Strout *et al.* [24] avoid this problem by describing dependences using Presburger arithmetic with uninterpreted functions and resolving the dependences at run time. This inspector/executor-inspired approach [6] could also be used to determine whether a loop revisits entities by simply traversing the iteration space and keeping a record of any accessed element.

We propose a symbolic approach that codifies the relations between mesh entities, e.g., that an edge is associated with two nodes. This allows a compiler to determine statically whether iteration-space narrowing is likely to be profitable. For example, given the previous assertion, a compiler can infer that a function on nodes invoked within a loop over edges will be repeatedly executed with the same arguments. If the compiler determines that the function is sufficiently complex and that its computation is side-effect free and independent of the surrounding loop, it can re-instantiate it within a loop that directly iterates over its arguments.

7. DATA AND COMPUTATION REORDERING

Data packing strategies that reorder the layout of data elements have been successful in improving locality and reducing bandwidth consumption [13]. We show in Section 7.1 that `KOLAH` benefits from this traditional use of data packing. In addition, we propose using packing to manipulate iteration spaces, making them amenable to code restructuring. Section 7.2 introduces a loop partitioning scheme that reorders computation and creates several partially evaluated versions of a loop to facilitate data reuse through blocking. Finally, Section 7.3 demonstrates that the two above approaches have different iteration-space ordering preferences and that resolving them leads to better performance than the application of either in isolation.

Each of the following computation reordering strategies relies on mesh semantics to complement traditional analyses to ensure correctness. In each case, mesh and field semantics developed above are sufficient to ensure that no loop-carried dependences exist and that reordering is safe. Other attempts to overcome limitations of compiler analyses to reorder or, equivalently, to parallelize loops have leveraged specialized language constructs or run-time dependence analysis. Titanium is a Java dialect that includes a `foreach` construct that does not specify the execution order of body instances [27]. Unfortunately, given the entrenched nature of scientific codes, we can not mandate new programming interfaces. Run-time data dependence analysis [22, 24] and simple run-time checks that select between safe and optimized code versions [3] are powerful approaches, but incur run-time overheads and increase code size to accommodate alternate code versions.

7.1 Data Packing

The triply-nested loop structure that implements performance-critical gradient and divergence operators leads to non-strided memory accesses. The implementation of `grad` in Figure 4 shows that the iteration space on sides traversed in the inner loop is defined by a face, whose iteration space is in turn defined by an enclosing loop over zones. Each of these loops accesses a subvolume of the mesh. Since these accesses do not match memory order, they result in a non-strided access pattern.

Consecutive packing [7] reduces the impact of this non-sequential behavior by linearizing mesh entities in memory according to their access order within the loop, to the extent allowed by repeated accesses. Thus mesh entities accessed consecutively in time are more likely to be stored consecutively in memory. This effectively increases spatial locality for small objects. Unfortunately, a cache line is not large enough to accommodate multiple mesh entities; consecutive accesses do not enjoy spatial reuse of a cache line. Thus, Table 3 shows packing sides does not affect L1 and L2 misses, though it should reduce TLB misses by providing page-level reuse.[3]

Although `KOLAH` does not benefit from fine-grained spatial reuse, consecutive packing produces a sequence of addresses more amenable to stream prefetching than those resulting from coarse-grained packing strategies, such as bucket tiling [18]. Data packing transforms the address stream such that it has many short sequences of strided addresses that can be identified and exploited by the POWER3's hardware-based prefetcher. By doing so, it increases the number of useful prefetches by 30% over iteration-space narrowing. The result is a reduction in load latency from 3.2 cycles prior to data packing to 2.8 cycles afterwards and a 7% performance improvement.

7.2 Iteration-space Partitioning and Loop Specialization

Execution within mesh algorithms is often conditionalized on geometric properties. For example, the loop in Figure 3 performs additional computation if the conditional on line 32 evaluates to true, indicating that the edge has a "left" neighbor. The complete loop has a symmetric test for the "right" neighbor. Such conditionals reduce basic block size, making it more challenging for the architecture to effectively schedule instructions. This degrades performance since accommodating the frequent memory accesses and long-latency floating-point instructions common to unstructured grid codes requires a balanced instruction mix [9].

Though these properties are not known until run time, after the mesh has been constructed, many of them remain static after initialization. This knowledge allows us to remove conditionals, just as an awareness of static mesh connectivity helps eliminate unnecessary recomputation. In this example, we create four versions of the loop corresponding to the cross product of the results from the two branches: both taken, neither taken, left branch taken, or right branch taken. In all cases, the conditionals have been removed and their bodies have been inlined or removed as appropriate. For example, in the specialized loop corresponding to existence of the right neighbor only, we replace the two conditionals with the inlined body for the right neighbor.

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

The loop corresponding to edges with neither a left nor a right neighbor admits further partial evaluation, and in fact can be removed entirely. Through constant folding and aggressive inlining after both conditionals and their bodies have been removed, a compiler should be able to determine that line 38 sets `signDotProd` to zero, so that the conditional on line 39 fails and line 42 sets `edgeForcing` to the zero vector. Since adding a zero vector to `nodeForcing` has no effect[4], the entire loop is side-effect free and may be eliminated. Eliminating this loop reduced the number of executed loop instances by 29,800 out of 1,500,000.

This transformation to statically evaluate and remove conditionals is a specific instance of *iteration-space partitioning and loop specialization*. Mellor-Crummey *et al.* [16] also recognized that data packing reorders computation when the data is stored in an array that is accessed without indirection both prior to and after packing. By using a space-filling curve to reorder computation, they attained significant cache miss reductions. Our approach differs since it reorders computation according to some property of the induction variable to facilitate subsequent optimizations. This subsequent specialization and restructuring of loop bodies contrasts with computation reordering optimizations that only reorder iteration spaces or introduce additional loop nests. Optimizations that introduce temporal locality illustrate these differences.

Gropp *et al.* [10] reorder a loop over edges in the unstructured mesh code FUN3D to introduce locality within loop body statements operating on nodes. By sorting the edges according to the identifier of the node at either end, they move loop body instances accessing the same node temporally close to one another so that they reuse data in cache. In their study of irregular scientific applications, Mellor-Crummey *et al.* [16] extend blocking used in dense-matrix calculations to interaction lists in molecular dynamics appli-

[3] We do not report results from the POWER3 TLB miss counter because it does not measure TLB misses directly.

[4] Menon *et al.* [17] discuss a compiler framework that incorporates a semantic understanding of vectors and matrices. In this case, such knowledge is not required because the code is inlined as scalars, which the compiler is able to analyze.

cations. They do so by first assigning a block number to each particle based on its memory location and then accessing particles by iterating over blocks.

The loop over sides in Figure 3 also exhibits temporal reuse; edges are revisited since they are not unique to a given side. We sort the sides based on their edge's identifier; this makes sides sharing an edge contiguous in the iteration space and provides temporal reuse of cached data. We can further exploit this iteration space reordering to tile the loop. We do so by lifting all statements that are dependent solely on an edge before any statements dependent on the side induction variable. Because edges are reused across consecutive loop instances, we introduce an inner loop over all sides sharing an edge. This register tiling over edges ensures an edge and computation on that edge are reused across sides sharing it.

Despite the significant reduction in loads and instructions, Table 3 shows that iteration-space partitioning does not perform as well as data packing. Loop partitioning induces an order that interferes with the POWER3 hardware prefetching mechanism, reducing the number of useful prefetches per load from 0.013, in the case of data packing, to 0.007. This results in an average load latency of 4.4 cycles for iteration-space partitioning, whereas data packing maintains a significantly lower 2.8-cycle load latency.

7.3 Packing Order Conflict Resolution

Applications studied in previous work [7] have a single dominant loop that provides an obvious packing order; the above two sections demonstrate that this is not the case in `KOLAH`, where packing orders inspired by different concerns induce different performance. The presence of multiple packing order preferences implies that orders that balance data packing's reduction in load latency with iteration-space partitioning's reduction in dynamic instructions can improve overall performance.

Table 3 shows that a compromise with superior performance does exist: multiple-constraint reordering both allows code restructuring to eliminate instructions and successfully exploits the hardware prefetcher to reduce the load latency to 3 cycles. It outperforms all previous optimizations and results in a final cumulative speedup of 4.7× over the baseline.

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

8. RELATED WORK

We broadly group related work into studies characterizing the performance limitations of scientific codes and frameworks for expressing and leveraging domain-specific semantics. Other work related to proposed optimizations is discussed in the respective sections.

Performance studies of scientific codes: Several reports indicate the significant performance impact of indirect memory accesses on unstructured grid applications. Anderson *et al.* [2] concentrate on minimizing memory references in the Fortran77 unstructured mesh code FUN3D. In their performance evaluation of scientific codes, Vetter and Yoo [26] study the unstructured mesh code UMT. They find that UMT suffers poor cache performance and significant stalls due to loads. The regularity metric defined by Mohan *et al.* [19] lends insight to this poor cache performance. They quantify a code's regularity as the number of memory accesses that occur within a strided stream divided by the total number of accesses. They determine that UMT has a relatively low regularity of 0.44, a result consistent with its heavy use of indirection. Semantics-enabled common subexpression elimination and mesh precomputation reduce the indirect memory accesses responsible for the overhead of mesh entity iterator dereferencing and mesh traversal to avoid exacerbating the effect of indirection in these codes.

Jin and Mellor-Crummey [14] also take the approach of manually optimizing a scientific library. To optimize stencil computation in SMG98, they target *hypre*, a library that provides abstractions of Cartesian grids, grid hierarchies, and iterators for use in creating multigrid applications.

Semantics-aware compilers and optimizations: Several projects have the goal of extending the base language with an awareness of user-defined abstractions [8, 12, 15, 21]. ROSE [21] is a source-to-source translator written to optimize C++ libraries and user-defined abstractions. Like Magik [8], it allows users to directly access and manipulate the compiler's IR. ROSE supports a Magik-like IR-based transformation scheme, as well as a string-based interface that requires less intimate knowledge of compiler technology. Users write analyses and transformations that traverse, query, and transform the abstract syntax tree. Broadway [12] and Telescoping Languages [15] share the goal of exploiting semantic information to optimize user-defined libraries, but do not focus on providing a framework for writing translators as do ROSE and Magik.

Recent work within the ROSE project [28] describes an array abstraction interface used to explicitly communicate array semantics to the compiler. The interface maps standard, abstracted operations on arrays, such as `len()`, to the concrete implementation within the particular array class, such as `this.getLength()`. Thus, the compiler operates at the level of the array abstractions until it needs to emit code. We are extending this work to encompass mesh abstractions.

9. CONCLUSION

We proposed a small set of semantics, tailored to unstructured mesh abstractions but independent of their implementations. These semantics complement traditional analyses; by providing information where alias and side-effect analysis proved too conservative to optimize mesh abstractions, they enabled traditional optimizations including code motion and common subexpression elimination, and improved performance by 46%. Recognition of these semantics both inspired and enabled novel optimizations that led to speedups of 4.7× and 4.9× on POWER3 and Pentium 4 platforms, respectively. Because the semantics apply to any application or library using common mesh and field abstractions, we expect them to enable optimizations, with similar gains, across mesh implementations.

Recognizing the limits of traditional compiler analyses when confronted with mesh abstractions and defining a set of semantics to overcome them were necessary steps towards the automation of the optimizations that exploit these semantics. We are in the process of implementing these mesh optimizations within the ROSE [21, 23] compiler framework. We have incorporated a parser for the Broadway annotation lan-

guage [11] into ROSE and intend to adapt it to describe mesh semantics. We will use this language to express domain-specific semantic information [12] where our alias and side-effect analysis are insufficient to determine the correctness of a desired optimization.

10. ACKNOWLEDGMENTS

Support was provided by a DOE CSGF fellowship under grant number DE-FG02-97ER25308 and by the National Science Foundation under award ITR/NGS-0325536. Additionally, portions of this work were performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. LLNL Document Number UCRL-CONF-212479.

11. REFERENCES

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

[2] W. Anderson, W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of IEEE/ACM Supercomputing '99*, Nov. 1999.

[3] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic loop transformations and parallelization of java. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, May 2000.

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

[5] I. Corporation. HPM Tool Kit — Home Page. http://www.alphaworks.bim.com/tech/hpmtoolkit, Jan. 2005.

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

[7] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, May 1999.

[8] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the 1st Conference on Domain-specific Languages*, pages 103–118, 1997.

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

[10] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Performance modeling and tuning of an unstructured mesh cfd application. In *Proceedings of IEEE/ACM Supercomputing '00*, Nov. 2000.

[11] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-specific Languages*, pages 39–52, 1999.

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

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

[14] G. Jin and J. Mellor-Crummey. Experiences tuning smg98—a semicoarsening multigrid benchmark based on the *hypre* library. In *Proceedings of the International Conference on Supercomputing*, pages 305–314, June 2002.

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

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

[17] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the International Conference on Supercomputing*, pages 434–443, June 1999.

[18] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 192–202, 1999.

[19] T. Mohan, B. de Supinski, S. McKee, F. Mueller, A. Yoo, and M. Schulz. Identifying and exploiting spatial regularity in data memory references. In *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.

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

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

[22] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of the International Conference on Supercomputing*, pages 274–284, June 2002.

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

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

[25] N. Trickett, T. Nakagawa, R. Mani, and D. Gfroerer. *Understanding IBM e-server pSeries Performance and Sizing*. IBM, Feb. 2001. POWER3 node specs.

[26] J. S. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. In *Proceedings of IEEE/ACM Supercomputing '02*, Nov. 2002.

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

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