

Annotation-Based Empirical Performance Tuning Using Orio

Albert Hartono
Dept. of Computer Science and Engg.
Ohio State University
Columbus, Ohio 43210-1277
Email: hartonoa@cse.ohio-state.edu

Boyana Norris
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844
Email: norris@mcs.anl.gov

P. Sadayappan
Dept. of Computer Science and Engg.
Ohio State University
Columbus, Ohio 43210-1277
Email: saday@cse.ohio-state.edu

Abstract—For many scientific applications, significant time is spent in tuning codes for a particular high-performance architecture. Tuning approaches range from the relatively nonintrusive (e.g., by using compiler options) to extensive code modifications that attempt to exploit specific architecture features. Intrusive techniques often result in code changes that are not easily reversible, and can negatively impact readability, maintainability, and performance on different architectures. We introduce an extensible annotation-based empirical tuning system called Orio that is aimed at improving both performance and productivity. It allows software developers to insert annotations in the form of structured comments into their source code to trigger a number of low-level performance optimizations on a specified code fragment. To maximize the performance tuning opportunities, the annotation processing infrastructure is designed to support both architecture-independent and architecture-specific code optimizations. Given the annotated code as input, Orio generates many tuned versions of the same operation and empirically evaluates the alternatives to select the best performing version for production use. We have also enabled the use of the Pluto automatic parallelization tool in conjunction with Orio to generate efficient OpenMP-based parallel code. We describe our experimental results involving a number of computational kernels, including dense array and sparse matrix operations.

I. MOTIVATION

The size and complexity of scientific computations are increasing at least as fast as the improvements in processor technology. Programming such scientific applications is hard, and optimizing them for high performance is even harder. This results in a potentially large gap between the achieved performance of applications and the peak available performance, with many applications achieving 10% or less of the peak (e.g., [1], [2]). A greater concern is the inability of existing languages, compilers, and systems to deliver the available performance for the application through fully automated code optimizations.

Delivering performance without degrading productivity is crucial for the success of scientific computing. Scientific code developers generally attempt to improve performance by applying one or more of the following three approaches: manually optimizing code fragments, using tuned libraries for key numerical algorithms, and, less frequently, using compiler-based source transformation tools for loop-level optimizations. Manual tuning is time-consuming and impedes readability and performance portability. Tuned libraries often deliver excellent

performance without requiring significant programming effort, but can only provide limited functionality. General-purpose source transformation tools for performance optimizations are few and have not yet gained popularity among computational scientists, at least in part because of poor portability and steep learning curves.

II. RELATED WORK

Ideally, a developer should only have to specify a few simple command-line options and then rely on the compiler to optimize the performance of an application on any architecture. Compilers alone, however, cannot fully satisfy the performance needs of scientific applications. First, compilers must operate in a black-box fashion and at a very low level, limiting both the type and number of optimizations that can be done. Second, static analysis of general-purpose languages, such as C, C++, and Fortran, is necessarily conservative, thereby precluding many possible optimizations. Third, in the process of transforming a mathematical model into a computer program, much potentially useful (for optimization purposes) information is lost since it cannot be represented by the programming language. Finally, extensive manual tuning of a code may prevent certain compiler optimizations and result in worse performance on new architectures, resulting in loss of performance portability.

An alternative to manual or automated tuning of application codes is the use of tuned libraries. The two basic approaches to supplying high-performance libraries include providing a library of hand-coded options (e.g., [3]–[5]) and generating optimized code automatically for the given problem and machine parameters. ATLAS [6], [7] for BLAS [3] and some LAPACK [8] routines, OSKI [9] for sparse linear algebra, PhiPAC [10] for matrix-matrix products, and domain-specific libraries such as FFTW [11] and SPIRAL [12] are all examples of the latter approach. Most automatic tuning approaches perform empirical parameter searches on the target platform. FFTW uses a combination of static models and empirical techniques to optimize FFTs. SPIRAL generates optimized digital signal processing libraries by an extensive empirical search over implementation variants. GotoBLAS [5], [13], on the other hand, achieves near-peak performance on several architectures by using hand-tuned data structures and kernel

operations. These auto- or hand-tuned approaches can deliver performance that can be five times as fast as that produced by many optimizing compilers [7]. The library approach, however, is limited by the fact that optimizations are highly problem- and machine-dependent. Furthermore, at this time, the functionality of the currently available automated tuning systems is quite limited.

General-purpose tools for optimizing loop performance are also available. LoopTool [14] supports annotation-based loop fusion, unroll/jamming, skewing and tiling. The Matrix Template Library [15] uses template metaprograms to tile at both the register and cache levels. A new tool, POET [16] also supports a number of loop transformations. POET offers a complex template-based syntax for defining transformations in a language-independent manner. Other research efforts whose goal, at least in part, is to enable optimizations of source code to be augmented with performance-related information include the X language [17] (a macro C-like language for annotating C code), the Broadway [18] compiler, and telescoping languages [19]–[21], and various meta-programming techniques [22]–[25].

Emerging annotation-based tools are normally designed by compiler researchers and thus the interfaces are not necessarily based on concepts accessible to computational scientists. The complexity of existing annotation languages and lack of common syntax for transformations (e.g., loop unrolling) result in steep learning curves and the inability to take advantage of more than one approach at a time. Furthermore, at present, there is no good way for users to learn about the tools available and compare their capabilities and performance.

III. ORIO DESIGN AND IMPLEMENTATION

Orio [26] is an empirical performance tuning system that takes annotated C source code as input, generates many optimized code variants of the annotated code, and empirically evaluates the performance of the generated codes, ultimately selecting the best performing version to use for production runs.

The Orio annotation approach differs from existing compiler- and annotation-based systems in the following significant ways. First, by not committing to a single general-purpose language, we can define annotation grammars that *restrict* the original syntax, enabling more effective performance transformations (e.g., disallowing pointer arithmetic in a C-like annotation language); furthermore, it enables the definition of new high-level languages which retain domain-specific information that is normally lost in low-level C or Fortran implementations. This in turn expands the range of possible performance-improving transformations. Second, Orio was conceived and designed with the following requirements in mind: portability (which precludes extensive dependencies on external packages), extensibility (new functionality must require little or no change to the existing Orio implementation and interfaces that enable integration with other source transformation tools must be defined), and automation (ultimately Orio should provide tools that manage *all* the steps

of the performance tuning process, automating each step as much as possible). Finally, Orio is usable with real scientific applications without requiring reimplementations. This ensures that the significant investments in the development of complex scientific codes is leveraged to the greatest extent possible.

Figure 1 contains a high-level graphical depiction of the code generation and tuning process implemented in Orio. Orio can be used to improve performance by source-to-source transformations such as loop unrolling, loop tiling, and loop permutation. The input to Orio is C code containing structured comments that include a simplified expression of the computation, as well as various performance-tuning directives. Orio scans the marked-up input code and extracts all annotation regions. Each annotation region is then processed by transformation modules. The code generator then produces the final C code with various optimizations that correspond to the specified annotations. Unlike compiler approaches, we do not implement a full-blown C compiler; rather, we use a precompiler that parses only the language-independent annotations.

Orio can also be used as an automatic performance tuning tool. The code transformation modules and code generator produce an optimized code version for each distinct combination of performance parameter values. Then each optimized code version is executed and its performance evaluated. After iteratively evaluating all code variants, the best-performing code is selected as the final output of Orio. Because the search space of all possible optimized code versions can be huge, a brute force search strategy is not always feasible. Hence, Orio provides various search heuristics for reducing the size of the search space and thus the empirical testing time.

A. Annotation Language Syntax

Orio annotations are embedded into the source code as structured C comments that start with `/*@` and end with `@*/`. For example, the annotation `/*@ end @*/` is used to indicate the end of an annotated code region. A simple grammar illustrating the basic syntax of Orio annotations is depicted in Figure 2. An annotation region consists of three main parts: leader annotation, annotation body, and trailer annotation. The annotation body can either be empty or contain C code that possibly includes other nested annotation regions. A leader annotation contains the name of the code transformation module used to transform and generate the annotated application code. A high-level description of the computation and several performance hints are coded in the module body inside the leader annotation. A trailer annotation, which has a fixed form (i.e., `/*@ end @*/`), closes an annotation region.

B. Orio Input Example

Figure 3 shows a concrete annotation example that empirically optimizes a C function for the Blue Gene/P architecture. This is an instance of an AXPY operation, i.e., one that computes $y = y + a_1x_1 + \dots + a_nx_n$, where a_1, \dots, a_n are scalars and y, x_1, \dots, x_n are one-dimensional arrays.

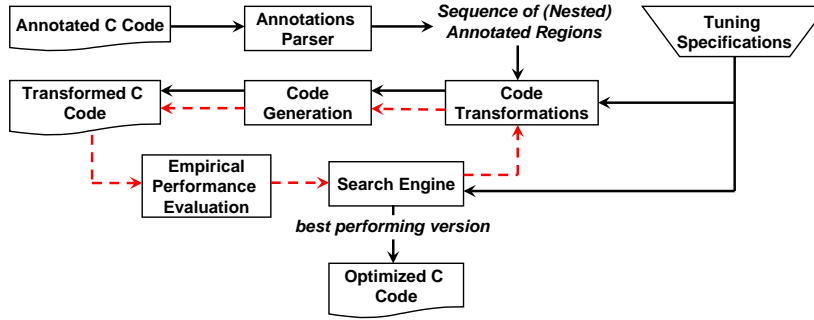


Fig. 1. Overview of Orio’s code generation and empirical tuning process.

```

<annotation-region> ::= <leader-annotation> <annotation-body> <trailer-annotation>
<leader-annotation> ::= /*@ begin <module-name> ( <module-body> ) @*/
<trailer-annotation> ::= /*@ end @*/
  
```

Fig. 2. Annotation language grammar excerpt.

The specific AXPY operation considered in this example corresponds to $n = 4$. The first annotation contains the `BGP_Align`¹ directive, which instructs Orio to dynamically load its memory-alignment optimization module and then generate preprocessor directives, such as pragmas and calls to memory alignment intrinsics, including a check for data alignment. The main purpose of these alignment optimizations is to enable the use of the dual floating-point unit (Double Hammer) of the Blue Gene/P, which requires 16-byte alignment. As discussed later in Section V-A, even these simple alignment optimizations can lead to potentially significant performance improvements. This example also shows the use of Orio’s loop transformation module (named `Loop`) to optimize the AXPY-4 loop by unrolling and generating OpenMP parallelization directives for exploiting multicore parallelism. In addition to the simple source transformations in this example, Orio also supports other optimizations, such as register tiling and scalar replacement.

Whereas the `BGP_Align` and `Loop` annotations in this example guide the source-to-source transformations, the purpose of the `PerfTuning` annotation is primarily to control the empirical performance tuning process. Details of the tuning specifications for optimizing the AXPY-4 code on Blue Gene/P are shown in the right-hand side of Figure 3. The tuning specification contains data required for building, initializing, and running experiments, including input variable information, the search algorithm, performance counting technique, performance parameters values, and execution environment details. The tuning specifications can be either integrated in the source code or defined in a separate file, as in this example.

The annotated AXPY-4 code (left side of Figure 3) uses two performance parameters whose values are defined in the tuning specification: the unroll factor (UF) and the boolean variable PAR, which is used to activate or deactivate OpenMP paral-

lization. These parameters are used by Orio to determine at runtime whether it is beneficial to parallelize the loop, i.e., whether there is enough work per thread to offset the OpenMP overhead or not.

Achieving the best performance for different input problem sizes may require different tuning approaches; thus, the entire tuning process is repeated for each specified problem size. In the AXPY-4 example, the search space includes seven different input problem sizes (variable N).

C. Annotation Parsing and Code Generation

The Orio system consists of several optimization modules, each implemented as a Python module. As mentioned earlier in Section III-B, given the module name in the leader annotation, Orio dynamically loads the corresponding code transformation module and uses it for both annotation parsing and code generation. If the pertinent module cannot be found in the transformation modules directory, an error message is emitted and the tuning process is terminated. This name-based dynamic loading provides flexibility and easy extensibility without requiring detailed knowledge or modification of the existing Orio software. Therefore, varied approaches to code transformations ranging from low-level loop optimizations for cache performance to composed linear algebra operations and new specialized algorithms can easily be integrated into Orio.

After parsing the annotation, each module performs a distinct optimization transformation prior to generating the optimized code. The transformation module can either reuse an existing annotation parser or define new language syntax and a corresponding parser component.

In some cases, the annotation is seemingly redundant, containing a version of the computation that is very similar to the original code. As mentioned earlier, we took this approach so that the annotation language can be defined in a way that enables more effective transformations (through restrictions or high-level domain information).

¹Architecture-specific annotations are simply ignored when the code is being tuned on an architecture that doesn’t support them.

```

void axpy4(int N, double *y,
double a1, double *x1, double a2, double *x2,
double a3, double *x3, double a4, double *x4) {

/*@ begin PerfTuning (
import spec axpy4_tune_spec;
) @*/

register int i;

/*@ begin BGP_Align (x1[],x2[],x3[],x4[],y[]) @*/
/*@ begin Loop (
transform Unroll (ufactor=UF, parallelize=PAR)
for (i=0; i<=N-1; i++)
y[i] += a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];
) @*/

for (i=0; i<=N-1; i++)
y[i] += a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];

/*@ end @*/
/*@ end @*/
/*@ end @*/
}

```

```

spec axpy4_tune_spec {
def build {
arg build_command =
'mpixlc_r -O3 -qstrict -ghot -qsmpr=omp:noauto';
arg batch_command =
'qsub -n 128 -t 20 --env "OMP_NUM_THREADS=4"';
arg status_command = 'qstat';
arg num_procs = 128;
}
def performance_counter {
arg method = 'basic timer';
arg repetitions = 10000;
}
def performance_params {
param UF[] = range(1,33);
param PAR[] = [True,False];
}
def input_params {
param N[] = [10,100,1000,10**4,10**5,
10**6,10**7];
}
def input_vars {
decl dynamic double y[N] = 0;
decl dynamic double x1[N] = random;
decl double a1 = random;
decl double a2 = random;
# ... omitted ...
}
def search {
arg algorithm = 'Exhaustive';
arg time_limit = 20;
}
}

```

Fig. 3. Orio input example; annotated AXPY-4 source code (left) and tuning specification for the Blue Gene/P (right).

```

/*@ begin Loop (
transform UnrollJam(ufactor=Ui)
for (i=0; i<=M-1; i++)
transform UnrollJam(ufactor=Uj)
for (j=0; j<=N-1; j++)
transform UnrollJam(ufactor=Uk)
for (k=0; k<=O-1; k++)
A[i][j] += B[i][k]*C[k][j];
) @*/
for (i=0; i<=M-1; i++)
for (j=0; j<=N-1; j++)
for (k=0; k<=O-1; k++)
A[i][j] += B[i][k]*C[k][j];
/*@ end @*/

```

```

def performance_params {
param Ui[] = range(1,33);
param Uj[] = range(1,33);
param Uk[] = range(1,33);
constraint reg_capacity = Ui*Uj+Ui*Uk+Uk*Uj<=32;
}
def input_params {
param M[] = [10,50,100,500,1000];
param N[] = [10,50,100,500,1000];
param O[] = [10,50,100,500,1000];
constraint square_matrices = (M==N) and (N==O);
}

```

Fig. 4. Example of specifying parameter constraints in Orio; annotated code for matrix-matrix multiplication (left) and constraint specification (right).

Current optimizations supported by Orio span different types of code transformation that are not provided by production compilers in some cases. Available optimizations include simple loop unrolling, memory alignment optimization, loop unroll/jamming, loop tiling, loop permutation, scalar replacement, register tiling, loop-bound replacement, array copy optimization, multicore parallelization (using OpenMP), and other architecture-specific optimizations (e.g., generating calls to SIMD intrinsics on Intel and Blue Gene/P architectures).

D. Search Space Exploration and Evaluation

As briefly discussed earlier, our empirical tuning approach systematically measures the performance costs of automatically-generated code variants in order to find the most optimal available version. In the context of empirical optimization, code variants are alternative, semantically equivalent implementations of the same computation. Each implementation variant is associated with a collection of different optimization parameters that correspond to source-to-source code transformations such as, for instance, unroll factors, tile

sizes, and loop permutation order. So, each coordinate in the search space of empirical tuning problem represents a distinct combination of performance parameter values. Both the dimension and the size of the search space depend on the total number and the value ranges of used performance parameters, respectively.

The conceptually straightforward approach to exploring the space of the parameter values is to use an exhaustive search procedure that is guaranteed to find the optimal code version. Normally the size of the search space is too large, making full coverage impractical. Thus, in addition to supporting exhaustive search, we have implemented several search heuristics. The simplest search heuristic is a random search, which picks a random coordinate in the search space at each step and then measures its performance; random search is not guaranteed to return close-to-optimal results. We have also developed two other search heuristics to effectively narrow the search space for close-to-optimal performance, including a heuristic based on the *Nelder-Mead simplex* method [27], [28], a popular non-derivative direct search method for optimization, and *simulated*

annealing [29]. Similarly to the implementation of Orio’s optimization modules, each search technique is implemented as an independent Python module in Orio and is dynamically loaded given only the algorithm’s name as one of the fields in the tuning specification.

To improve the quality of the search result further, each search heuristic is enhanced by applying a local search after the global search completes. The local search compares the best performance with neighboring coordinates. If a better coordinate is discovered, the local search continues recursively until no further performance improvement is possible or a user-specified termination criterion is met.

Further pruning of the search space is enabled through user-specified constraints in the tuning specifications. We use loop unroll/jamming transformation to make the discussion more concrete. Loop unroll/jamming [30] (coupled with scalar replacement) is mainly intended to increase data reuse at the register level. So, when unroll/jamming is applied to multiple loops, the values of unroll factors must be such that register locality is maximized while satisfying the register capacity constraints to avoid unnecessary register spills. Figure 4 shows an example of analytically enforcing register capacity constraints on matrix-matrix multiplication code that is optimized with loop unroll/jamming, assuming 32 registers. U_i, U_j, U_k are unroll factors for loops i, j, k , respectively. With the specified parameter constraints, the search space of performance parameter values is drastically trimmed down from 32,768 points (i.e., $32 * 32 * 32$) to only 65 points. This example also demonstrates that a set of constraints can also be imposed on input parameters to decide what input problem sizes to consider.

Orio also supports parallel search when parallel resources are available, e.g., when tuning on the Blue Gene/P. The parallel Orio driver concurrently executes multiple independent code variants in the same parallel job. After Orio submits a parallel job, each node in the target machine executes a distinct generated code variant to collect the code performance. The resulting performance results are collected and stored in a temporary file, which is later processed by Orio to determine the best performing variant. Figure 3 has an example of using parallel search by specifying the number of nodes to use (per job) with the `num_procs` variable. The remaining fields used by the parallel search are the `batch_command` and the `status_command` fields, which specify how Orio should submit a parallel job and query its status, respectively.

IV. PLUTO-ORIO INTEGRATION

A number of source-to-source transformation tools for performance optimization exist. Using these tools to achieve (near) optimal performance on different architectures, however, is still a nontrivial task that requires significant architectural and compiler expertise. By combining code transformation tools with an empirical tuning system, such as Orio, we can reduce the amount of manual effort and automatically determine the transformations that result in the best performance. We show in this section how Orio has been extended

with an external transformation program, called Pluto [31]. This integration also demonstrates the easy extensibility of Orio and the ability to leverage other source transformation approaches.

Pluto is a source-to-source automatic transformation tool aimed at optimizing a sequence of nested loops for data locality and coarse-grained parallelism simultaneously. Pluto employs a polyhedral model of arbitrary loop nests, where the dynamic instance (iteration) of each statement is viewed as an integer point in a well-defined space called the statement’s polyhedron. This statement representation and a precise characterization of data dependences enable Pluto to construct mathematically correct complex loop transformations. Pluto’s polyhedral-based transformations result in improved cache locality and loops parallelized for multicore architecture.

Figure 5 outlines the overall structure of the Pluto-Orio integrated system, which is implemented as a new optimization module in Orio. Initially, the transformation process takes two kinds of input: the loop code to be optimized and the optimization parameters code. The loop code is embedded in the annotation body block, while the performance parameters code is written in the module body block. A parser implemented inside the new module parses the performance parameters code to extract their values. The extracted values of the performance parameters include tile sizes, unroll factors, loop order (permutation), as well as several boolean values for triggering OpenMP parallelization, scalar replacement, and auto-vectorization. Using these parameter values, Pluto then performs polyhedral transformations (e.g., two-level tiling and OpenMP parallelization) on the input loop code. The resulting generated code is subsequently passed to the widely-available profiling tool `gprof` [32] for hotspot detection. The statistical information produced by `gprof` provides accurate locations (line numbers) of the loop nests where the Pluto-generated code spends most of its execution time. The identified hotspot loop nests are then decorated with Orio’s performance annotations for complementary syntactic transformations (e.g., loop permutation, loop unroll/jamming, scalar replacement, and explicit auto-vectorization). Finally, Orio optimizes the annotated hotspots as described in Section III.

Pluto also performs syntactic loop unroll/jam in a post-processing pass. The target loops, however, are limited only to innermost loops with a maximum depth of two (i.e., 1-D unrolling and 2-D unroll/jamming). So we choose to use the loop unroll/jam transformation already available in Orio, which is more flexible because it can be applied to loop nests of depths larger than two.

At present Orio does not employ any data dependence analysis when performing syntactic transformations. Therefore, there is no guarantee that the code generated after syntactic transformations is correct. Because of this, the tuning process using the integrated Pluto-Orio system is currently semi-automatic—user involvement is required to decide whether it is safe to apply a syntactic transformation on an identified hotspot.

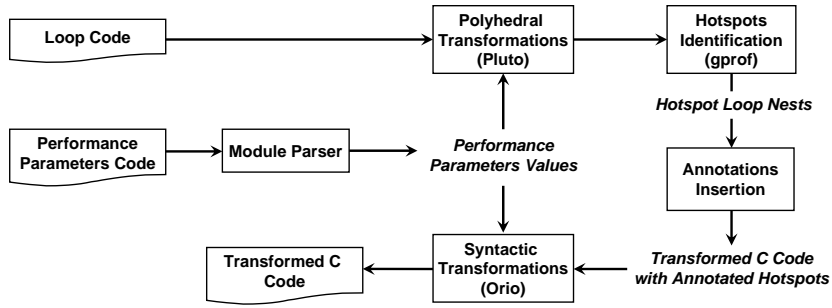


Fig. 5. Integration of Pluto into Orio’s code transformation module.

V. EXPERIMENTAL RESULTS

In this section, we discuss the performance results of several experiments on a multicore Intel Xeon workstation and a Blue Gene/P (both at Argonne). The workstation has dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64). Each compute node of the Blue Gene/P is equipped with four 850 MHz IBM PowerPC 450 processors with a dual floating-point unit and 2 GB total memory per node, private L1 (32 KB) and L2 (4 MB) caches, a shared L3 cache (8 MB), and running a proprietary lightweight operating system. We used version 10.1 of the Intel and version 9.0 of the IBM XL C/C++ V9.0 compilers on the Xeon and Blue Gene/P, respectively. Because of space considerations, here we discuss a limited number of computational kernels; more performance data for different computations is available at the Orio project website [26].

A. Sequence of Linear Algebra Operations

In this experiment, we tuned the performance of the AXPY-4 operation (see Figure 3) on a single node of the Blue Gene/P machine using the IBM xlc compiler. We measured the performance for two scenarios: using a single core per node and using all four cores. The results are shown in Figures 7(a) and 7(b), respectively. The single-core scenario was compiled with the following options: `-O3 -qstrict -qarch=450d -qtune=450 -qhot -qsmp=noauto`; the multicore scenario differs in the use of `-qsmp=auto` for the non-Orio versions. The parallel Orio version contains OpenMP parallelization directives in the generated code, thus necessitating the use of the `-qsmp=omp:noauto` compiler option. Included are performance numbers for four code variants: a simple loop implementation without any library calls (labeled “Compiler-optimized”), two BLAS-based implementations that use Goto BLAS [5] and the ESSL [4] libraries, and the Orio-tuned version.

The performance results shown in Figure 7 indicate that the code tuned by Orio consistently outperforms the other three versions for both the sequential and parallel cases. We observe that even for a simple algebraic operation, such as

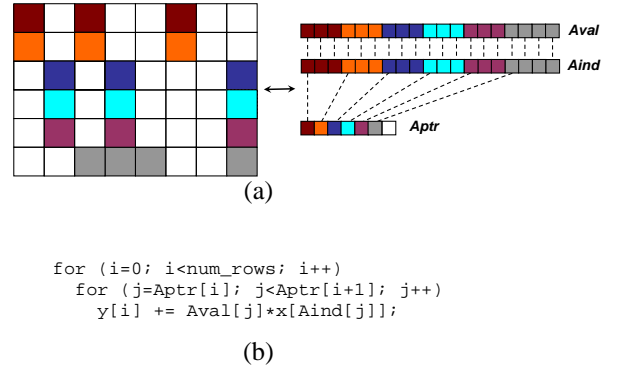


Fig. 6. (a) Compressed sparse row (CSR) format; (b) Basic implementation of CSR-based SpMV.

the composed AXPY routines, the compiler alone is unable to yield performance comparable to the empirically tuned version. Moreover, implementations that rely on calls to multiple tuned library routines (e.g., Goto BLAS and ESSL) suffer from loss of both spatial and temporal localities, resulting in inferior memory performance.

B. Sparse Matrix Computations

In this section we examine the effectiveness of Orio in optimizing key computations in PETSc [33], a toolkit for the parallel numerical solution of partial differential equations, by empirically tuning one of its heavily used kernels, sparse matrix-vector multiplication (SpMV). SpMV dominates the performance of various scientific applications; yet, traditional implementations of sparse kernels exhibit relatively poor performance because of the use of indirect addressing for accessing the matrix data elements.

The SpMV operation computes $\forall A_{i,j} \neq 0 : y_i \leftarrow y_i + A_{i,j} \cdot x_j$, where A is a sparse matrix, and x, y are dense vectors. Each element of A is used only once, and element reuse is possible only for x and y . Thus, to optimize SpMV, one should use compact data structures for A and try to maximize temporal reuse of x and y . One of the most commonly used data structures for storing a sparse matrix is compressed sparse row (CSR) storage [34], illustrated in Figure 6(a). Elements in each row of A are packed together in a dense array, $Aval$,

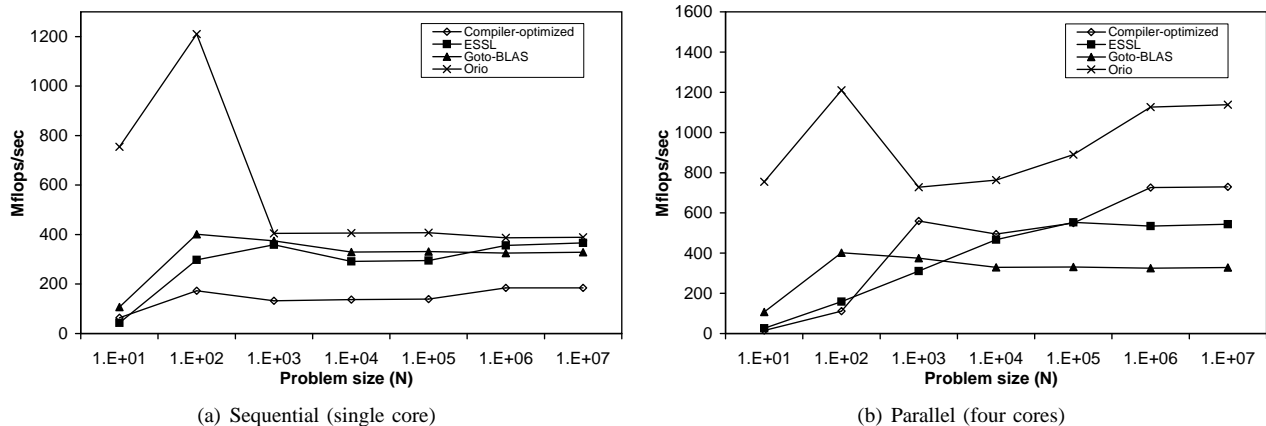


Fig. 7. Performance of AXPY-4 operations on Blue Gene/P.

and a corresponding array of integers `Aind` stores the column indices. The `Aptr` array designates where each sparse row begins in `Aval` and `Aind`. An implementation of SpMV using CSR storage is shown in Figure 6(b).

PETSc’s implementation of SpMV used in this experiment exploits matrix structure by using *inodes*, which represent rows with identical nonzero structure. PETSc detects inodes automatically in arbitrary sparse matrices. For example, the sparse matrix shown in Figure 6(a) contains three inodes. The first inode is of size two because it holds two adjacent rows (i.e., the first and the second rows) with the same nonzero structure. The second inode contains three consecutive rows with identical nonzero structure, and the last inode has only a single row. Knowing the properties of each inode at runtime enables maximum reuse of vector x since multiple elements of x can be loaded and used exactly once for all rows in the same inode structure. To accomplish this, one must use a register-blocking transformation. To optimize the inode-based SpMV routine, we incorporated a new transformation module inside Orio that implements various optimization strategies including register blocking, SIMDization, memory alignment optimization, loop-control optimization, accumulator expansion, and thread-level parallelization (with OpenMP). We then used Orio to automatically select the best version of the optimized inode SpMV. Only the outer loop that iterates over inodes was parallelized by using OpenMP.

To evaluate the performance of the tuned SpMV routine, we conducted the experiment using a 2-D driven cavity flow simulation application [35] (SNES ex27 in PETSc), on both the Xeon and Blue Gene/P. The “Compiler-optimized” label is used as a base case that represents the performance of the naive implementation of SpMV (shown in Figure 6(b)) optimized by the native compiler. We also tested the performance of the hand-tuned (by PETSc developers) SpMV code, which is included in PETSc releases.

Figure 8 shows the performance results on the Intel Xeon. The problem size is the number of grid points in the x and y directions, not the size of the sparse input matrix. The matrix dimension is based on the grid size and the

number of field components computed for each grid point; for example, an 8×8 problem involves sparse matrices of dimension 900 with 17,040 nonzero entries. In this experiment, we ran in three multicore modes: SMP (one MPI process, eight threads/process), dual (four MPI processes, two threads/process), and virtual node or VN (eight MPI processes, one thread/process). The code tuned by Orio consistently outperforms the others. Furthermore, the performance of the hand-tuned code is almost equivalent to that of the icc-optimized simple code. Performance differences between the Orio version and the hand-tuned version become more significant as more threads per process are employed to facilitate OpenMP parallelization. The SMP and VN results show that optimizations using loop-level parallelism (through OpenMP) achieve much better performance than using coarse-grained MPI parallelism.

The Blue Gene/P results in Figure 9 again show that empirical optimization using Orio produces the best performance for all cases. On this architecture, the performance gap between the xlc-optimized and the hand-tuned codes is now larger than in the Intel experiment. Similarly, by exploiting thread-level parallelism, the Orio-optimized code performs better than the hand-tuned version when the number of threads per process increases and the number of nodes decreases. For the single-thread cases (VN mode), thread-level parallelism is not exploited; nevertheless, the Orio version still performs better than the other two implementations.

C. Evaluation of the Pluto-Orio Integrated System

This section discusses the performance evaluation of the integrated Pluto-Orio system (discussed in Section IV) on the multicore Intel Xeon by using a number of application kernels that are nontrivial to optimize and parallelize. We compare the performance of the code tuned by Orio with the base code and the Pluto-generated codes. The Pluto code was obtained by running the base code with Pluto-0.3.0 [36] using `--tile` to employ loop tiling for L1 cache, `--l2tile` to employ loop tiling for L2 cache, and `--unroll` to employ loop unrolling, and for parallel code generation, an additional `--parallel`

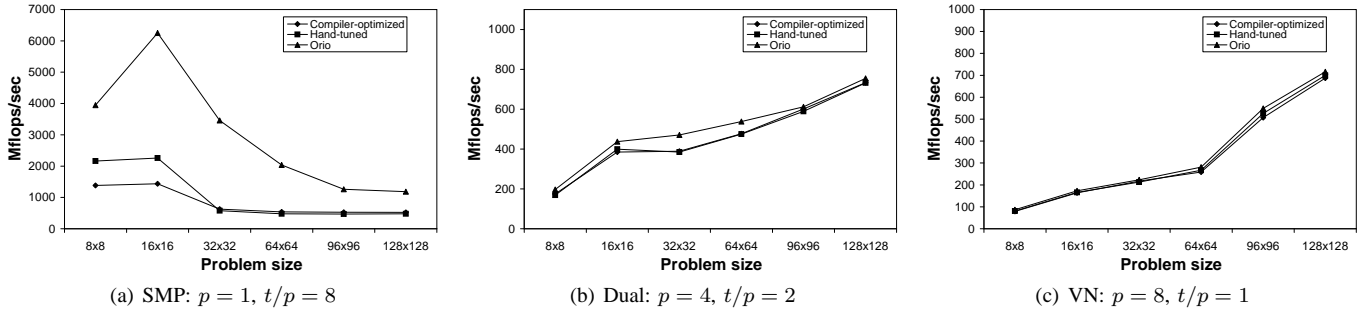


Fig. 8. Performance of inode SpMV on eight-core Intel machine; p is the number of processes, and t/p is the threads per process.

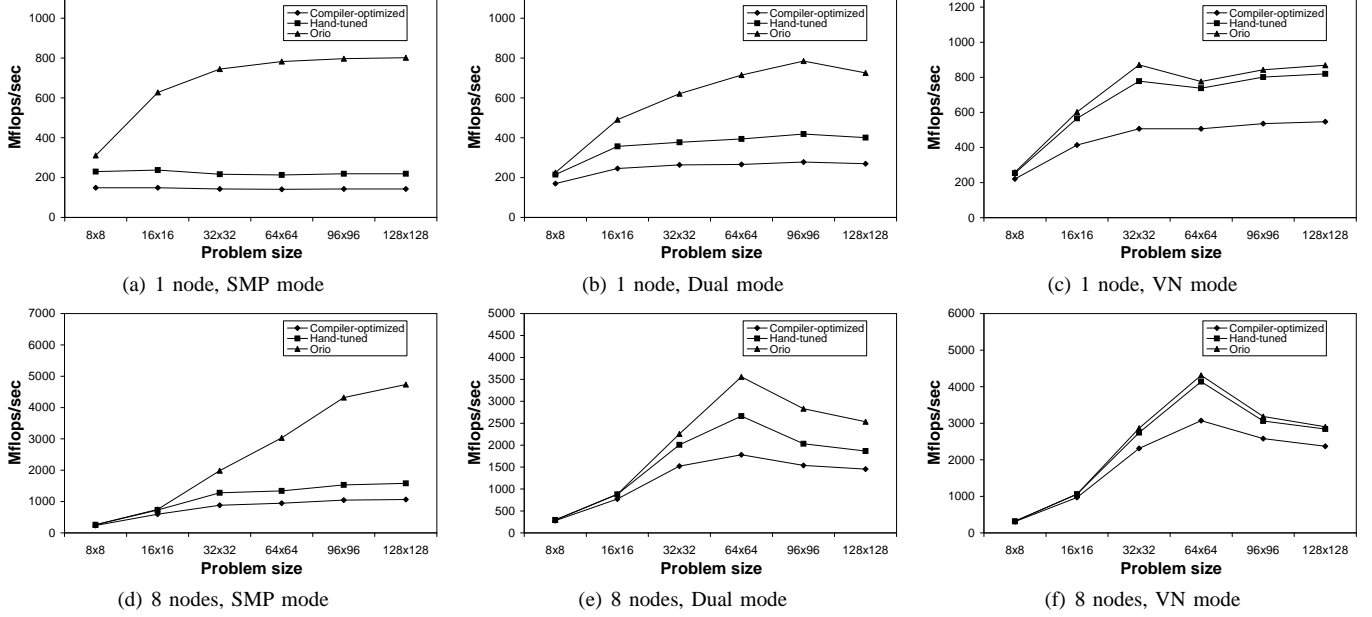


Fig. 9. Performance of inode SpMV on Blue Gene/P for 1 and 8 nodes.

option. We used Pluto’s default tile sizes (L1: 32x32 or 32x32x32; L2: 256x256 or 256x256x256) and default unroll factors (64 for 1-D unrolling and 8x8 for 2-D unroll/jamming). All codes were compiled with the Intel C compiler using the `-O3` optimization flag to enable auto-vectorization and other advanced optimizations, and `-parallel` (or `-openmp` in the case of manual OpenMP parallelization) to enable code parallelization.

In the following sections, we refer to the `icc`-optimized base code as “ICC,” the Pluto-generated code with L1 tiling and unroll/jamming as “Pluto (L1 tiling),” and the Pluto-generated code with L1 and L2 tilings and unroll/jamming as “Pluto (L1+L2 tiling).” The code tuned by Orio is referred to as “Pluto+Orio”.

1) *2-D Finite-Difference Time-Domain Method for Computational Electromagnetics*: We consider the two-dimensional finite difference time domain (FDTD) algorithm, a popular method for solving the time-dependent Maxwell’s equations in the context of computational electrodynamic problems. As shown in Figure 13, the 2-D FDTD method is implemented

```

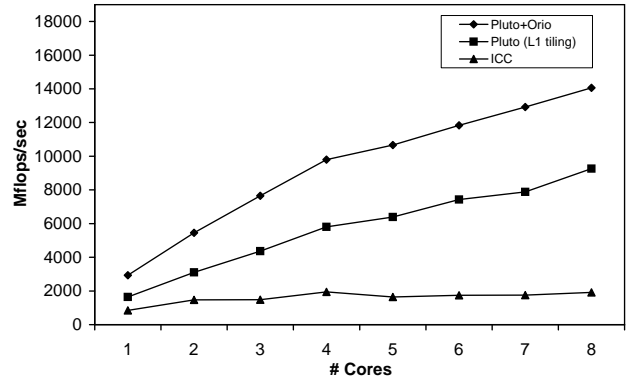
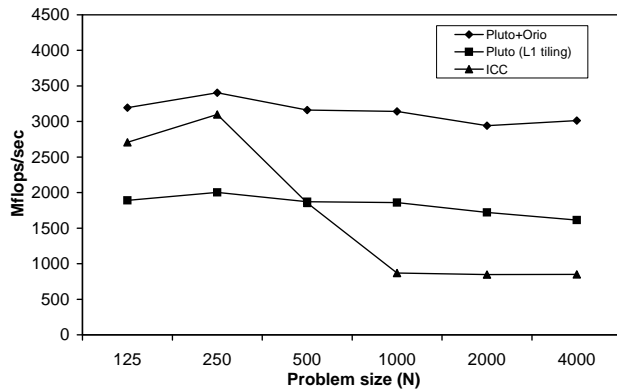
for(t=0; t<tmax; t++) {
  for (j=0; j<ny; j++) ey[0][j] = t;
  for (i=1; i<nx; i++)
    for (j=0; j<ny; j++)
      ey[i][j] -= 0.5*(hz[i][j] - hz[i-1][j]);
  for (i=0; i<nx; i++)
    for (j=1; j<ny; j++)
      ex[i][j] -= 0.5*(hz[i][j] - hz[i][j-1]);
  for (i=0; i<nx; i++)
    for (j=0; j<ny; j++)
      hz[i][j] -= 0.7*(ex[i][j+1] - ex[i][j]
        + ey[i+1][j] - ey[i][j]);
}

```

Fig. 13. 2-D FDTD code.

as an outer iteration over time containing four imperfectly nested loops. The arrays ex and ey denote the electric field components, and the array hz denotes the magnetic field.

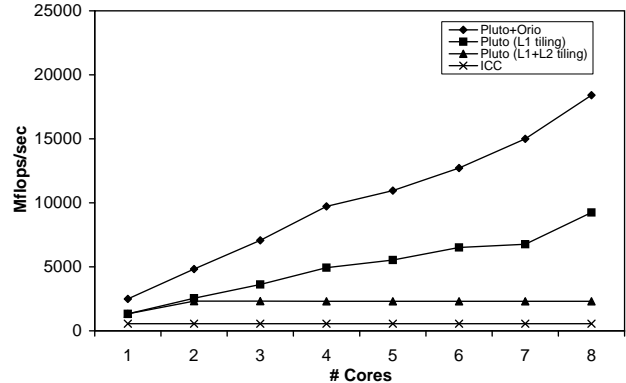
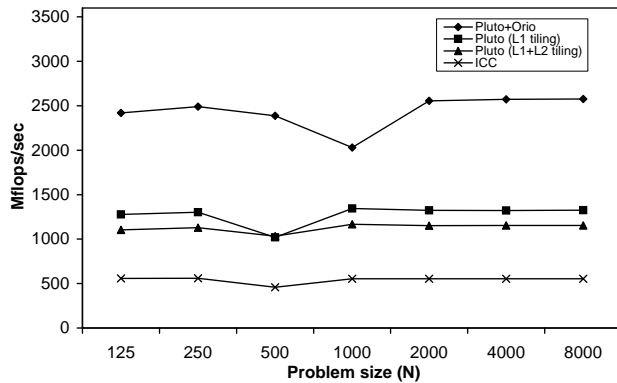
The performance of the sequential 2-D FDTD code for $tmax = 500$ and $nx = ny$ is shown in Figure 10(a). The base code optimized by `icc` alone performs better than Pluto for small problem sizes since all input arrays fit in the L2 cache (insufficient computation to offset Pluto’s tiling overhead).



(a) Sequential (T=500)

(b) Parallel (T=500, N=2000)

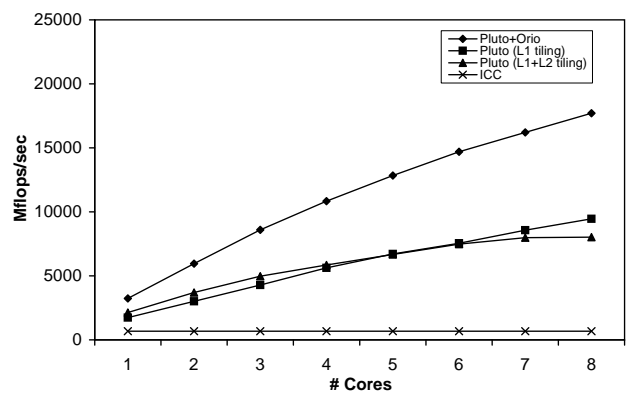
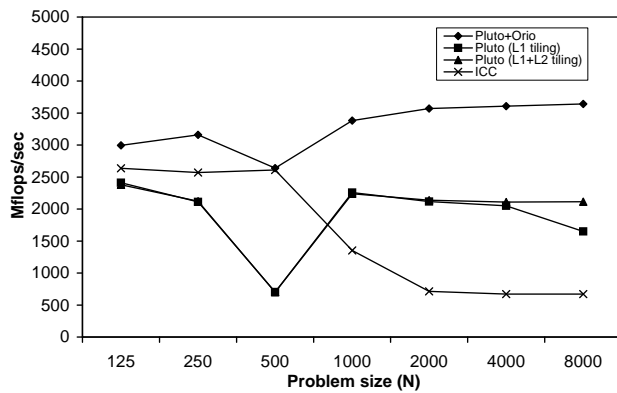
Fig. 10. 2-D FDTD performance on eight-core Intel Xeon.



(a) Sequential (T=500)

(b) Parallel (T=500, N=4000)

Fig. 11. 3-D Gauss-Seidel performance on eight-core Intel Xeon.



(a) Sequential

(b) Parallel (N=4000)

Fig. 12. LU decomposition performance on eight-core Intel Xeon.

```

for (t=0; t<=T-1; t++)
  for (i=1; i<=N-2; i++)
    for (j=1; j<=N-2; j++)
      A[i][j] = (A[i-1][j-1] + A[i-1][j]
                + A[i-1][j+1] + A[i][j-1] + A[i][j]
                + A[i][j+1] + A[i+1][j-1] + A[i+1][j]
                + A[i+1][j+1]) / 9.0;

```

Fig. 14. 3-D Gauss-Seidel code

```

for (k=0; k<=N-1; k++) {
  for (j=k+1; j<=N-1; j++)
    A[k][j] /= A[k][k];
  for (i=k+1; i<=N-1; i++)
    for (j=k+1; j<=N-1; j++)
      A[i][j] -= A[i][k]*A[k][j];
}

```

Fig. 15. LU decomposition code.

As the array sizes increase, the lack of data reuse impairs the base code’s performance, whereas the Pluto performance remains about the same. When the input arrays are small, Orio discovers that applying Pluto’s polyhedral transformations is not beneficial, and therefore it employs only its syntactic transformations on the original FDTD code. For large problem sizes, Orio exploits some of the Pluto’s code transformations and enhances these further with its syntactic optimizations, resulting in performance consistently and significantly higher than both the base and Pluto codes (up to 86% over Pluto).

Figure 10(b) shows the multicore performance obtained for $tmax = 500$ and $nx = ny = 2000$. The results indicate that whereas `icc` is unable to auto-parallelize the code, Pluto detects the existence of pipelined parallelism and then successfully parallelizes the code. Locality-exploiting optimizations by Orio additionally improve the Pluto performance by up to 78%. Moreover, we observe that because of memory contention between the two quad-core Intel processors, the speedup of both the Pluto and Orio codes slightly decreases when the number of cores used is greater than four.

2) 3-D Gauss-Seidel Successive Overrelaxation Method:

Figure 14 shows the 3-D Gauss-Seidel computation, which is sometimes referred to as *successive displacement method*, indicating the dependence of the iterations on the ordering. If the ordering is changed, the components of the new iterations will change as well.

Figure 11(a) contains the sequential performance results for $T = 500$, which show that applying Pluto’s polyhedral tiling on the original code always delivers performance boosts, which range from 126% to 142%. When using Pluto, performing two-level tiling (for both L1 and L2 caches) yields 13% lower performance than performing only L1 tiling. This is also reflected in the best sequential code found by Orio, where one-level tiling (for L1 cache only) is always found to be better than two-level tiling. The sequential speedup obtained from tuning the Pluto code with Orio is significant, ranging from 51% to 133%.

The parallel performance results for $T = 500$ and $N = 4000$ are shown in Figure 11(b). Again we observe that `icc` fails to parallelize the code, whereas Pluto is able to parallelize it and Orio improves the performance of the one-level tiled

Pluto code further by up to 128% (and up to 548% for the two-level tiled Pluto code). We observe that tiling over both L1 and L2 can result in worse performance than L1-only tiling because the number of tiles can be less than the number of cores.

3) *LU Factorization*: LU factorization or decomposition is a numerical method for the solution of linear systems of equations; a simple implementation is shown in Figure 15.

The sequential performance results are shown in Figure 12(a). Similarly to the 2-D FDTD results, the `icc`-optimized code is more efficient than the Pluto-generated codes when the input arrays are small and fit in the L2 cache. Because of better data locality, however, the performance of the Pluto-tiled codes is better than that of the base code as the input arrays get larger. Consequently, Orio employs Pluto’s polyhedral transformations only for large input sizes prior to applying its own syntactic transformations. Compared to the two Pluto codes, the Orio-tuned code yields performance improvements ranging between 26% and 277%.

The parallel performance results shown in Figure 12(b) were obtained for $N = 4000$. We observe that `icc` is not able to parallelize the code, whereas Pluto achieves higher performance by exploiting multicore parallelism. Furthermore, both Pluto-generated codes have comparable performance, with slightly better scalability exhibited by the single-level tiled code. Orio further improves the performance of the Pluto-generated versions by a factor of 51% to 120%.

VI. CONCLUSIONS AND FUTURE WORK

We have described the design and implementation of Orio, an extensible Python software system for defining annotation-based performance-improving transformations. Our experiments with a number of different types of computations on two different architectures show that Orio can deliver performance improvements when used alone or in conjunction with other source transformation tools.

Orio is a new tool under active development; future work includes (but is not limited to) providing support for annotating and generating Fortran code, defining new annotation languages and corresponding transformation modules, e.g., using matrix notation for linear algebra operations, and integration with other source transformation tools through new optimization modules.

ACKNOWLEDGMENT

The authors would like to thank Uday Bondhugula of IBM Research for many productive discussions and his valuable help with Pluto. This work was supported in part by the U.S. Dept. of Energy under Contract DE-AC02-06CH11357, the National Science Foundation through awards 0403342, 0509467, and 0811781, and a State of Ohio Development Fund.

REFERENCES

- [1] H. Simon, L. Oliker, A. Canning, J. Carter, S. Ethier, and J. Shalf, “Evaluation of leading scalar and vector architectures for scientific computations,” <http://repositories.cdlib.org/lbnl/LBNL-55291>, Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-55291, 2004.

- [2] J. Carter, L. Oliker, and J. Shalf, "Performance evaluation of scientific applications on modern parallel vector systems," in *VECPAR*, ser. Lecture Notes in Computer Science, M. J. Daydé, J. M. L. M. Palma, A. L. G. A. Coutinho, E. Pacitti, and J. C. Lopes, Eds., vol. 4395. Springer, 2006, pp. 490–503.
- [3] J. J. Dongarra, J. D. Cruz, I. S. Duff, and S. Hammarling, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 16, pp. 1–17, 1990.
- [4] "Engineering Scientific Subroutine Library (ESSL) and parallel ESSL," www-03.ibm.com/systems/p/software/essl.html, 2006.
- [5] K. Goto and R. van de Geijn, "High-performance implementation of the level-3 BLAS," The University of Texas at Austin, Department of Computer Sciences, Tech. Rep. TR-2006-23, 2006.
- [6] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27.
- [7] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.
- [8] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK's user's guide*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.
- [9] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005*, ser. Journal of Physics: Conference Series, vol. 16. Institute of Physics Publishing, June 2005, pp. 521–530.
- [10] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PhiPAC: A portable, high-performance, ANSI C coding methodology," in *International Conference on Supercomputing*, 1997, pp. 340–347.
- [11] M. Frigo, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the ICASSP Conference*, vol. 3, 1998, p. 1381.
- [12] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," in *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, Feb 2005, pp. 216–231.
- [13] K. Goto, "GotoBLAS," <http://www.tacc.utexas.edu/resources/software/>, 2007.
- [14] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent, "HPCVIEW: A tool for top-down analysis of node performance," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 81–104, Aug 2002.
- [15] J. G. Siek and A. Lumsdaine, "A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library," in *ECOOP Workshops*, ser. Lecture Notes in Computer Science, S. Demeyer and J. Bosch, Eds., vol. 1543. Springer, 1998, pp. 468–469.
- [16] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized optimizations for empirical tuning," in *Workshop on Performance Optimization of High-Level Languages and Libraries (POHLL)*. IEEE Computer Society, March 2007, pp. 1–8.
- [17] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali, "Language for the compact representation of multiple program versions," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC05)*, ser. Lecture Notes in Computer Science. Germany: Springer-Verlag, 2006, no. 4339, pp. 136–151.
- [18] C. Lin and S. Z. Guyer, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, July 2005.
- [19] K. Kennedy *et al.*, "Telescoping languages project description," <http://telescoping.rice.edu/>, 2006.
- [20] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey, "Telescoping languages: A system for automatic generation of domain languages," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 387–408, 2005.
- [21] K. Kennedy, "Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems," in *Proceedings of International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, May 2000, pp. 297–304.
- [22] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, June 1995.
- [23] D. Weise and R. Crew, "Programmable syntax macros," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1993, pp. 156–165.
- [24] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Meta-Object Protocol*. Cambridge (MA): MIT Press, 1991.
- [25] S. Chiba, "A metaobject protocol for C++," in *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Oct 1995, pp. 285–299.
- [26] "Orio project," trac.mcs.anl.gov/projects/performance/orio, 2008.
- [27] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the Nelder-Mead simplex method in low dimensions," *SIAM Journal of Optimization*, vol. 9, pp. 112–147, 1998.
- [28] R. M. Lewis, Michael, and W. Trosset, "Direct search methods: Then and now," *Journal of Computational and Applied Mathematics*, vol. 124, pp. 200–0, 2000.
- [29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.
- [30] V. Sarkar, "Optimized unrolling of nested loops," *Int. J. Parallel Program.*, vol. 29, no. 5, pp. 545–581, 2001.
- [31] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2008.
- [32] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," *SIGPLAN Notices*, vol. 17, no. 6, p. 120, Jun. 1982.
- [33] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Users Manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 2.1.5, 2004.
- [34] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, December 2003.
- [35] T. S. Coffey, C. T. Kelley, and D. E. Keyes, "Pseudo-transient continuation and differential-algebraic equations," *SIAM J. Sci. Comput.*, vol. 25, no. 2, pp. 553–569, 2003.
- [36] "The Pluto automatic parallelizer," sourceforge.net/projects/pluto-compiler.