Contents

0-8493-0052-5/00/$0.00+$.50

Chapter 1

Annotations for Productivity and Performance Portability

Boyana Norris

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, norris@mcs.anl.gov

Albert Hartono

Dept. of Computer Science and Engineering, The Ohio State University, 2015 Neil Avenue, Columbus, OH 43210, hartonoa@cse.ohio-state.edu

William Gropp

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, gropp@mcs.anl.gov

1.1 Introduction

In many scientific applications, significant time is spent in tuning codes for a particular high-performance architecture. Multiple approaches to such tuning exist, ranging from the relatively nonintrusive (e.g., by using compiler options) to extensive code modifications that attempt to exploit specific architecture features. In most cases, the more intrusive code tuning is not easily reversible and thus can result in inferior performance on a different architecture or, in the worst case, in wholly nonportable code. Readability is also greatly reduced in such highly optimized codes, resulting in lowered productivity during code maintenance.

We introduce an extensible annotation system that aims to improve both performance and productivity by enabling software developers to insert annotations into their source code that trigger a number of low-level performance optimizations on a specified code fragment. The annotations are special struc-

0-8493-0052-5/00/$0.00+$.50

```
void axpy_1(int n, double *y,
            double a, double *x)
{
  /*@ begin Variable (x[],y[]) @*/
  int i;
  for (i=0; i < n; i++)
    y[i] = y[i] + a * x[i];
  /*@ end @*/
}
```

```
void axpy_1(int n, double *y,
            double a, double *x)
{
  /*@ begin Variable (x[],y[]) @*/
  #pragma disjoint (*x, *y)
  if ((((int)(x)|(int)(y)) & 0xF) == 0) {
    __alignx(16,x);
    __alignx(16,y);
    int i;
    for (i=0; i < n; i++)
      y[i] = y[i] + a * x[i];
  } else {
    int i;
    for (i=0; i < n; i++)
      y[i] = y[i] + a * x[i];
  }
  /*@ end @*/
}
```

FIGURE 1.1: Memory-related annotation example: annotated code (left) and resulting generated code with optimized Blue Gene/L pragmas and alignment intrinsic calls (right).

tured comments inside the source code and are processed by a precompiler to produce optimized code in a general-purpose language, such as C, C++, or Fortran. In order to maximize the performance tuning opportunities, annotations are designed to support both architecture-independent and architecture-specific code optimizations. Given the annotated code as input, the annotation tool generates many tuned versions of the same operation, using different optimization parameters. The best-performing version can subsequently be used in production application runs.

Figure 1.1 shows a simple annotation example for the Blue Gene/L that targets memory alignment optimizations. Here, the annotations are shown as C comments starting with `/*@`. The `Variable` annotation directive results in the generation of architecture-specific preprocessor directives, such as pragmas, and calls to memory alignment intrinsics, including a check for alignment. Even these simple optimizations can lead to potentially significant performance improvements, such as gains of up to 60% in memory bandwidth with annotations (discussed in more detail in Section 1.3).

What makes annotations especially powerful is that they are not limited to certain operations and can be applied to complex computations involving many variables. Thus, annotations can be used for arbitrary expressions, exploiting the developer's understanding of the application to perform low-level code optimizations. Such optimizations may not be produced by general-purpose compilers because of the necessarily conservative nature of program analysis for languages such as Fortran and C/C++. These optimizations include low-level tuning of array operations for deep memory hierarchies, through loop blocking, tiling, and unrolling, as well as composing linear algebra operations and invoking specialized algorithms for key com-

putations. More advanced optimizations on other data structures, such as matrices, would present even greater opportunities for cache optimizations. Our aim is to use existing tools for performing such code optimization transformations where possible; the examples here merely illustrate the sorts of transformations that are sometimes necessary for performance and, because they are both ugly and system specific, are rarely performed in application codes.

The remainder of this paper is organized as follows. Section 1.2 describes the annotation language and the design and implementation of the code transformation system. Section 1.3 presents preliminary performance studies on the Blue Gene/L. Section 1.4 reviews related work. Section 1.5 contains a summary and a brief outline of future work.

1.2 Design and Implementation

In this section, we describe the design and current implementation of our annotation software system. The annotation language is embeddable in a general-purpose language, such as C/C++ and Fortran. Our ultimate goal is to construct an annotation system that is general, flexible, and easily extensible with new annotation syntax and corresponding code optimizations. In the following subsections, we describe the overall design of the system, and give an overview of the annotation language syntax and existing code generation modules.

1.2.1 Annotation System Design

Performance annotations are expressed through semantic comments, inserted into application source code. These annotations allow programmers to simultaneously describe the computation and specify various performance-tuning directives. Annotations are treated as regular comments by the compiler but are recognized by the annotation system as syntactical structures that have particular meaning.

Figure 1.2 depicts at a high level the structure and operation of the annotation system. The system first scans the annotated application source code, subdividing it into annotated code regions. Each region is then passed to the corresponding code generator for potential optimizations. Finally, target language code (C, C++, or Fortran) with various applied optimizations is generated for the annotated regions.

The annotation system consists of one or more code generators, each im-

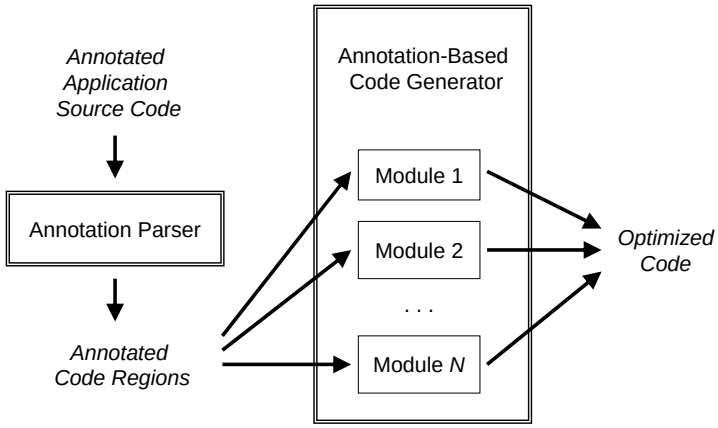

FIGURE 1.2: Overview of the annotation system.

plemented as a Python module.[1] Modules can be added to the system at any time without requiring modifications to the existing infrastructure. Each code generation module can define new syntax or extend the syntax of an existing annotation definition. Using the information supplied in the annotated region, each module performs a distinct optimization transformation prior to generating the optimized code. These optimizations can span different types of code transformations that are not provided by compilers in some cases, such as memory alignment, loop optimizations, various architecture-specific optimizations, high-level algorithmic optimizations, and distributed data management.

1.2.2 Annotation Language Syntax

Annotations are specified by programmers as comments and do not affect the correctness of the original program. We specify annotations using stylized C/C++ comments that start with `/*@` and end with `@*/` (in Fortran, comments starting with `![--` or `c[--` and ending with `--]` are used). These markers are called opening and closing *annotation delimiters*, respectively. For example, the annotation `/*@ end @*/` (or `![-- end --]` in Fortran) is used to indicate the end of an annotated code region.

Table 1.1 shows the simple grammar of the annotation language syntax. The structure of an *annotated code region* consists of three main parts: a *leader annotation*, an *annotation body block*, and a *trailer annotation*. An annotation body block can either be empty or contain C/C++ source code

[1]While the current implementation is in Python, we plan to add language-independent interfaces that would allow new modules to be added in a number of different languages.

TABLE 1.1: Annotation language grammar excerpt.

annotated-code-region	::=	*leader-annotation*
		annotation-body-block
		trailer-annotation
leader-annotation	::=	`/*@ begin` *module-name*
		`(` *module-body-block* `)`
		`@*/`
annotation-body-block	::=	
	|	*non-annotation-code annotation-body-block*
	|	*annotated-code-region annotation-body-block*
trailer-annotation	::=	`/*@ end @*/`

```
1.  void axpy_4(int n, double *y, double a1, double *x1, double a2, double *x2,
2.              double a3, double *x3, double a4, double *x4)
3.  {
4.    /*@ begin Variable (x1[],x2[],x3[],x4[],y[]) @*/
5.    int i;
6.    for (i=0; i < n; i++)
7.      y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
8.    /*@ end @*/
9.  }
```

FIGURE 1.3: Example of annotated application source code.

that may include other annotated regions. A leader annotation contains the *name* of the code generation module, which is loaded dynamically by the annotation system to optimize and generate the annotated application code. A high-level description of the computation and several performance hints are specified in the *module body block* inside the leader annotation and are used as input during the optimization and code generation phases. A trailer annotation closes an annotated code region, designated by `/*@ end @*/` (or the equivalent Fortran comment).

An example annotated application code can be seen in Fig. 1.3, where lines 4–8 contain the annotated code region with lines 4 and 8 as the leader and trailer annotations, respectively, and lines 5–7 as the annotation body block. The name of the annotation code generation module in this example is `Variable`, and the module input is the string '`x1[],x2[],x3[],x4[],y[]`'.

1.2.3 Current Annotation Modules

As we discussed in Section 1.2.2, given the module name in the leader annotation, the annotation system dynamically loads the corresponding code generation module and uses it to transform and generate the code in the annotation body block. If the pertinent module cannot be found in the modules directories, an error message is produced, and the annotation system process is terminated. The name-based dynamic loading provides flexibility and

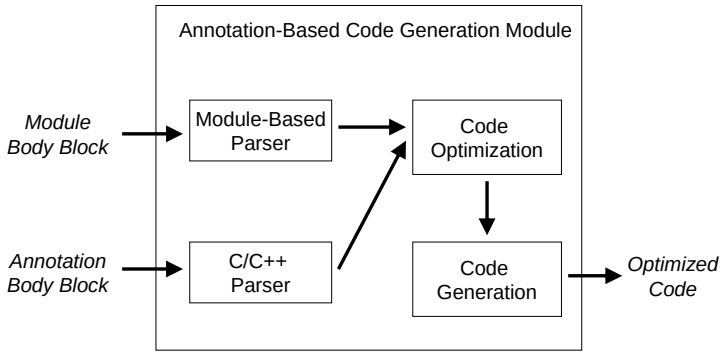

FIGURE 1.4: Structure of a code generation module.

easy extensibility without requiring detailed knowledge or modification of the existing annotation software.

1.2.4 Code Generation Module

Figure 1.4 portrays the general structure of an annotation-based code generation module. In order to generate optimized code, each module takes two kinds of input parameters: the module parameters specified in the *module body block* and the code contained in the *annotation body block*. The module body normally includes information that is essential for performing code optimization and generation, such as multidimensional array variables, loop structures, and loop unrolling factors. In order to process this information, new language syntax and a corresponding parser component must be implemented for each code generation module. In addition, the annotation body code, currently expressed in a language that is a slightly restricted version of C, must be parsed and provided as input to the transformation module. New modules that use the same syntax for the code block can simply use an existing parser. Modules can also introduce new syntax for the annotation body and in that case must provide their own parser. While one could view the annotation body block as redundant (since the annotated code region already contains the same or similar statements), the motivation behind requiring application developers to include the computation itself as part of the annotation is twofold. First, basing the transformation modules on the actual application code would require a full-blown compiler infrastructure in the target language (e.g., C, C++, or Fortran). While open-source research projects for these languages exist, they do not support arbitrary codes reliably yet; furthermore, they have been ported to only a few architectures and on supported platforms they require a large number of prerequisite packages to be available. Requiring this complex compiler infrastructure to avoid the relatively small manual effort in creating annotations runs counter to our goal

of making the annotation system portable and easy to install and use. Second, requiring in effect a "rewrite" of the code to be optimized encourages simplification and enables the code optimization effort to start with a cleaner, rather than an already hand-tuned, version of the code. Furthermore, we are considering annotations that would allow the computation to be expressed by using domain-specific high-level languages, thus capturing the semantics without imposing tuning constraints resulting from the use of a general-purpose language.

We note that annotations can also be nested; that is, an annotation body block can contain other annotated regions. Hence, the optimization and code generation are carried out recursively by the annotation tool to handle nested annotations. Next we describe in more detail the design and implementation of the currently available code generation modules.

1.2.4.1 Memory Alignment Module

The objective of the memory alignment module is to exploit memory alignment optimizations on the Blue Gene/L architecture. The dual floating-point unit (Double Hummer) of the Blue Gene/L's PowerPC 440d processor can be controlled with special instructions for parallel floating-point computations [19]. Efficient use of the Double Hummer requires 16-byte alignment.

The IBM XL compiler attempts to pair contiguous data values on which it can operate in parallel. Therefore, performance can be improved by explicitly specifying floating-point data objects that reside in contiguous memory blocks and are correctly aligned. In order to facilitate such parallelization, the compiler requires additional directives to remove possibilities of aliasing and to check for data alignment [19].

We illustrate the implementation of this module using the simple example previously shown in Fig. 1.3. As we can see in the leader annotation segment, this module is named `Variable`. The module body contains a list of array variables. In this example, `x1[]`, `x2[]`, `x3[]`, `x4[]`, and `y[]` are the array variables to be aligned.

The resulting optimized version that corresponds to the example given above can be seen in Fig. 1.5. A `#pragma disjoint` directive (line 5) is injected into the optimized code to inform the compiler that none of the listed identifiers share the same storage within the scope of their use. This information enables the compiler to avoid the overhead of reloading data values from memory each time they are referenced, and to operate on values already resident in registers (this is similar to the role of the `restrict` C keyword). Note that this directive demands that the two identifiers be disjoint. If the identifiers in fact share the same memory address, the computation can be incorrect.

Furthermore, the Blue Gene/L architecture requires that the addresses of the two data values, which are loaded in parallel in a single cycle, be aligned such that the loaded values do not cross a cache-line boundary. If they cross

```
1.  void axpy_4(int n, double *y, double a1, double *x1, double a2, double *x2,
2.              double a3, double *x3, double a4, double *x4)
3.  {
4.    /*@ begin Variable (x1[],x2[],x3[],x4[],y[]) @*/
5.    #pragma disjoint (*x1, *x2, *x3, *x4, *y)
6.    if ((((int)(x1)|(int)(x2)|(int)(x3)|(int)(x4)|(int)(y)) & 0xF) == 0) {
7.      __alignx(16,x1); __alignx(16,x2); __alignx(16,x3); __alignx(16,x4);
8.      __alignx(16,y);
9.      int i;
10.     for (i=0; i < n; i++)
11.        y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
12.   } else {
13.     int i;
14.     for (i=0; i < n; i++)
15.        y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
16.   }
17.   /*@ end @*/
18. }
```

FIGURE 1.5: Optimized version of annotated code shown in Fig. 1.3.

TABLE 1.2: New language grammar of memory alignment module.

align-module-body-block	::=	*array-variable-list*
array-variable-list	::=	*array-variable*
	|	*array-variable* , *array-variable-list*
array-variable	::=	*array-variable-name dimension-list*
dimension-list	::=	*dimension*
	|	*dimension dimension-list*
dimension	::=	`[]`
	|	`[` *variable-name* `]`
	|	`[` *constant* `]`

this boundary, a severe performance penalty is imposed by the alignment trap generated by the hardware. Thus, testing for data alignment is important. In the optimized code example, checking for data alignment is executed in line 6. Lines 7–8 contain calls to the `__alignx` intrinsic functions. These function calls are used to notify the compiler that the arriving data is correctly aligned, so the compiler can generate more efficient loads and stores.

The complete grammar for the new language syntax of the memory alignment module is shown in Table 1.2. In addition to one-dimensional arrays, multidimensional arrays can be specified in the module body block. One example is `a[i][]`, which refers to the location of the ith row of the two-dimensional array `a`. The empty bracket is basically used to refer to the starting address, where a sequence of adjacent data to be computed is stored in the memory. Another valid example is `b[i][j][]`. However, the `c[][i][j]` specification is invalid in C/C++ codes because the use of row-major array storage would result in noncontiguous memory starting at that address. On the other hand, the `c[][i][j]` array variable specification is valid when used

TABLE 1.3: Overview of the language structure of the loop optimization module.

loop-opt-module-body-block	::=	*statement-list*
statement-list	::=	*statement*
	|	*statement statement-list*
statement	::=	*labeled-statement*
	|	*expression-statement*
	|	*compound-statement*
	|	*selection-statement*
	|	*iteration-statement*
	|	*jump-statement*
	|	*transformation-statement*
transformation-statement	::=	`transform` *submodule-name* `(` *keyword-argument-list* `)` *statement*
keyword-argument-list	::=	*keyword-argument*
	|	*keyword-argument* `,` *keyword-argument-list*
keyword-argument	::=	*keyword-name* `=` *expression*

to annotate Fortran source because Fortran employs column-major array allocation. Such data arrangement rules can be enforced easily by this module using simple semantic analysis.

We note that the statements in the annotation body block (lines 5–7 in Fig. 1.3) are simply reproduced by this module without any transformation and thus require no parsing. A complete C/C++ parser component is therefore not needed, simplifying the implementation of this module.

1.2.4.2 Loop Optimization Module

The primary goal of the loop optimization module is to provide extensible high-level abstractions for expressing generic loop structures in conjunction with a variety of potential low-level optimization techniques, such as loop unrolling, skewing, and blocking for cache, and including some architecture-specific optimizations. Two optimization strategies that have been constructed and integrated into the annotation system are *loop unrolling* and *automated simdization*.

An overview of the new language syntax introduced by the loop optimization module is given in Table 1.3. Essentially, *a subset of C statements* and a newly defined *transformation statement* constitute the language grammar of this module. For compactness, further details on each of the C statement clauses are not given in this grammar. Many C language features, such as declarations, variable pointers, switch statements, enumeration constants, and cast expressions, are excluded from the language grammar selection in order to reduce the implementation complexity of this module. Some of these

```
1.  void ten_reciprocal_roots(double* x,
2.                                double* f)
3.  {
4.    int i;
5.    /*@ begin LoopOpt(
6.         transform Loop(unroll=4,
7.                 index=i, lower_bound=0,
8.                 upper_bound=10, step=1)
9.           f[i] = 1.0 / sqrt(x[i]);
10.   ) @*/
11.   for (i = 0; i < 10; i++)
12.     f[i] = 1.0 / sqrt(x[i]);
13.   /*@ end @*/
14. }
```

```
1.  void ten_reciprocal_roots(double* x,
2.                                double* f)
3.  {
4.    int i;
5.    /*@ begin LoopOpt(
6.         transform Loop(unroll=4,
7.                 index=i, lower_bound=0,
8.                 upper_bound=10, step=1)
9.           f[i] = 1.0 / sqrt(x[i]);
10.   ) @*/
11.   #if ORIGLOOP
12.     for (i = 0; i < 10; i++)
13.       f[i] = 1.0 / sqrt(x[i]);
14.   #else
15.     for (i = 0; i <= 10 - 3; i += 4)
16.     {
17.       f[i] = 1.0 / sqrt(x[i]);
18.       f[i + 1] = 1.0 / sqrt(x[i + 1]);
19.       f[i + 2] = 1.0 / sqrt(x[i + 2]);
20.       f[i + 3] = 1.0 / sqrt(x[i + 3]);
21.     }
22.     for (; i <= 10; i += 1)
23.       f[i] = 1.0 / sqrt(x[i]);
24.   #endif
25.   /*@ end @*/
26. }
```

FIGURE 1.6: Cache optimizations annotation example: annotated code (left) and resulting generated code with unrolled loop body (right).

will be added as the module evolves, increasing the variety of codes that can be annotated easily.

A new transformation statement clause is added into the grammar to achieve the flexibility of extending the loop optimization module with new *transformation submodules*. Using the provided submodule name, the loop optimization module dynamically searches for the corresponding submodule and then uses it to transform the transformation statement body. Additional data specified in the keyword argument list serve as input to the transformation submodule.

The example in Fig. 1.6 demonstrates how to annotate an application code with a simple portable *loop unrolling* optimization that aims to increase cache hit rate and to reduce branching instructions by combining instructions that are executed in multiple loop iterations into a single iteration. The keyword used to identify the loop optimization module is `LoopOpt`. The `Loop` name denotes the transformation submodule, whose basic function is to represent general loop structures. Four fundamental parameters are used to create a loop structure: the index variable name (`index`), the index's lower bound value (`lower_bound`), the index's upper bound value (`upper_bound`), and the iteration step size (`step`). For instance, the simple loop structure

```
for (i = 0; i <= n-1; i++)
  x[i] = x[i] + 1;
```

can be represented by using the following transformation statement.

```
transform Loop(index=i, lower_bound=0, upper_bound=n-1, step=1)
```

```
x[i] = x[i] + 1;
```

Annotating a loop structure for loop unrolling optimizations is straightforward: we add another keyword argument of the form "`unroll = n`", where `n` signifies how many times the loop body will be unrolled/replicated in the generated code. In the example in Fig. 1.6, the loop body is unrolled four times, resulting in the unrolled loop structure (lines 15–21). The final loop (lines 22–23) is generated for any remaining iterations that are not executed in the unrolled loop. Additionally, the generated code includes the original loop (lines 12–13) that can be executed through setting the `ORIGLOOP` (line 11) preprocessor variable accordingly.

As mentioned in Section 1.2.4.1, on Blue Gene/L, the IBM's XL C/C++ and XL Fortran compilers enable us to speed computations by exploiting the PowerPC 440d's Double Hummer dual floating-point unit (FPU) to execute two floating-point operations in parallel. Furthermore, there are quad-word load/store instructions (lfpd, stfpd) that can double the bandwidth between L1 and registers. The XL compilers support a set of highly optimized *built-in functions* (Oedipus instructions) [19] that have an almost one-to-one correspondence with the Double Hummer instruction set. These functions are designed to efficiently manipulate complex-type variables and include functions that convert noncomplex data to complex types. Hence, programmers can manually parallelize their code by using these built-in functions.

We have observed that the XL compilers automatically generate Double Hummer instructions for relatively simple expressions involving complex or double numbers. In many cases, however, the compiler-generated code utilizes only a single FPU, such as for the assignment statement below.

```
z[0] = a[0] + b[0] + 8.5 * c[0];
z[1] = a[1] + b[1] + 8.5 * c[1];
```

One approach to parallelizing the expression on the right-hand side of these assignments is first to divide the complex expression into a sequence of simple arithmetic expressions and then to translate each simple operation to its corresponding intrinsic functions. We refer to this process as *automated simdization*. We can transform the expression example using an intermediate temporary variable `t` to perform the following two-step computation:

```
t[0] = b[0] + 8.5 * c[0];
t[1] = b[1] + 8.5 * c[1];
z[0] = a[0] + t[0];
z[1] = a[1] + t[1];
```

which can be automatically converted into the following parallel code fragment.

```
double _Complex t, _t_1, _t_2, _t_3, _t_4;
_t_1 = __lfpd(&b[0]);
_t_2 = __lfpd(&c[0]);
t = __fxcpmadd(_t_1, _t_2, 8.5);
```

```
_t_3 = __lfpd(&a[0]);
_t_4 = __fpadd(t, _t_3);
__stfpd(&z[0], _t_4);
```

We have developed a simdization transformation module named `BGLSimd` as an extension of the loop optimization module. A complete example of the use of the `BGLSimd` annotation is shown in Fig. 1.7. This annotated code example shows the case when the statement to be simdized occurs inside the body of loop that will be unrolled. Therefore, the simdization and unrolling transformations are applied simultaneously. In this coupled transformation process, each simdized statement must be associated to a particular unrolled loop. To create this association, a keyword argument that has `loop_id` keyword identifier (lines 6 and 8) must be included. Loop identification is especially necessary when the statement to be simdized occurs in multiple nested unrolled loops.

We note that automated simdization requires that the associated loop has unit stride access (i.e. `step = 1`). Another important semantic constraint in this case is that, given the fact that the number of parallel floating-point units of Blue Gene/L is two, the associated loop unrolling factor must be divisible by two.

1.3 Performance Studies

In this section we present performance results for annotation-based optimization of some operations for which tuned library implementations do not exist or perform inadequately.

1.3.1 STREAM Benchmark

Preliminary results from employing simple annotations for uniprocessor optimizations are given in Table 1.4. These data describe the performance of an example array operation from the STREAM benchmark [11], also shown in Fig. 1.1. This computation is similar to some computational kernels in accelerator modeling codes, such as VORPAL's particle push methods [12]. The achieved memory bandwidth of the compiler-optimized version is significantly lower than that of the annotated version. The latter includes annotations specifying that the array variables are disjoint and should be aligned in memory, if possible, and that the loop should be unrolled. The same compiler options were used for both the original and the annotated versions. Even these simple optimizations can lead to potentially significant performance improvements. Table 1.4 shows gains of up to 60% in memory bandwidth with annotations.

```
1.  void vector_op(double* x, double* a,  double *b, double* d,
2.                 double c1, double c2, int n)
3.  {
4.    int i;
5.    /*@ begin LoopOpt(
6.        transform Loop(loop_id=lp1, unroll=4, index=i,
7.                       lower_bound=0, upper_bound=n-1, step=1)
8.          transform BGLSimd(loop_id=lp1)
9.            x[i] = a[i] - c1 * b[i] + c2 * d[i];
10.   ) @*/
11.   for (i = 0; i < n; i++)
12.     x[i] = a[i] - c1 * b[i] + c2 * d[i];
13.   /*@ end @*/
14. }
```

```
1.  void vector_op(double* x, double* a,  double *b, double* d,
2.                 double c1, double c2, int n)
3.  {
4.    int i;
5.    /*@ begin LoopOpt(
6.        transform Loop(loop_id=lp1, unroll=4, index=i,
7.                       lower_bound=0, upper_bound=n-1, step=1)
8.          transform BGLSimd(loop_id=lp1)
9.             x[i] = a[i] - c1 * b[i] + c2 * d[i];
10.   ) @*/
11.   #if ORIGLOOP
12.     for (i = 0; i < n; i++)
13.       x[i] = a[i] - c1 * b[i] + c2 * d[i];
14.   #else
15.     for (i = 0; i <= n - 1 - 3; i += 4)
16.     {
17.       {
18.         double _Complex _t_11, _t_12, _t_13, _t_14, _t_15;
19.         _t_11 = __lfpd(&a[i]);
20.         _t_12 = __lfpd(&d[i]);
21.         _t_13 = __fxcpmadd(_t_11, _t_12, c2);
22.         _t_14 = __lfpd(&b[i]);
23.         _t_15 = __fxcpnmsub(_t_13, _t_14, c1);
24.         __stfpd(&x[i], _t_15);
25.       }
26.       {
27.         double _Complex _t_11, _t_12, _t_13, _t_14, _t_15;
28.         _t_11 = __lfpd(&a[(i + 2)]);
29.         _t_12 = __lfpd(&d[(i + 2)]);
30.         _t_13 = __fxcpmadd(_t_11, _t_12, c2);
31.         _t_14 = __lfpd(&b[(i + 2)]);
32.         _t_15 = __fxcpnmsub(_t_13, _t_14, c1);
33.         __stfpd(&x[(i + 2)], _t_15);
34.       }
35.     }
36.     for (; i <= n - 1; i += 1)
37.       x[i] = a[i] - c1 * b[i] + c2 * d[i];
38.   #endif
39.   /*@ end @*/
40. }
```

FIGURE 1.7: Example of automatic simdization for the Blue Gene/L: annotated code (top) and resulting generated code with simdized and unrolled loop body (bottom).

TABLE 1.4: Memory bandwidth of $a = b + ss * c$ on the Blue Gene/L, where a, b, and c are arrays of size m, and ss is a scalar.

Array Size m	**No Annotations (MB/s)**	**Annotations (MB/s)**
10	1920.00	2424.24
100	3037.97	6299.21
1000	3341.22	8275.86
10000	1290.81	3717.88
50000	1291.52	3725.48
100000	1291.77	3727.21
500000	1291.81	1830.89
1000000	1282.12	1442.17
2000000	1282.92	1415.52
5000000	1290.81	1446.48

1.3.2 AXPY Operations

We consider generalized $AXPY$ operations of the form $y = y + a_1x_1 + \cdots + a_nx_n$, where $a_1, \ldots, a_n$ are scalars and $y, x_1, \ldots, x_n$ are one-dimensional arrays. These operations are more general forms of the triad operation discussed in the previous section. Figure 1.8 shows the performance of this computation for various array sizes when $n = 4$ on the Blue Gene/L at Argonne National Laboratory. Included are timing and memory bandwidth results for five versions of the code: a simple loop implementation without any library calls (labeled "Original"), two BLAS-based implementations that use the Goto BLAS library [5, 6] and the ESSL [3], respectively, and two annotated versions. The first annotated version contains only variable alignment and loop unrolling annotations, while the second additionally contains a BGLSimd annotation similar to the one illustrated in Fig. 1.7. For our earliest experiments, the ESSL was the only BLAS library available; Goto BLAS was added more recently. All versions were compiled with the same aggressive compiler optimization options. The performance improvement of the annotated version over the simple loop (original) version is between 78% and 488% (peaking for array size 100). SIMD operations were significantly effective only for certain array sizes, resulting in a factor of 6 improvement over the simple loop version. ESSL exhibited very poor performance compared to Goto BLAS. Both annotated versions outperformed the Goto BLAS version by 33% to 317% depending on the array sizes. Improvement over BLAS can be typically expected in most cases where several consecutive interdependent calls to BLAS subroutines are made. The AXPY and similar computations dominate certain types of codes, such as some automatically generated Jacobian computations, but tuned library implementations do not support such operations directly;

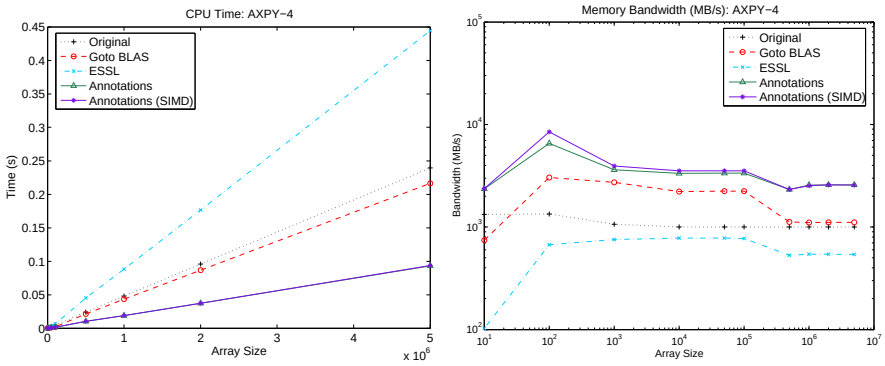

FIGURE 1.8: Performance on the Blue Gene/L for AXPY-4 operations: wall-clock time (left) and memory bandwidth (right).

hence, annotation-driven optimization can have significant positive impact on performance. Implementations that rely on calls to multiple tuned library subroutines suffer from loss of both spatial and temporal locality, resulting in inferior memory performance.

1.4 Related Work

In this section we present a brief overview of other approaches to performance optimization that are based on some higher-level semantic information (either user-defined or derived via compiler analysis).

Active libraries. Active libraries [15, 2] such as ATLAS [18], unlike traditional libraries, are geared to the generation and optimization of executable code. Some active libraries, such as the Blitz++ library [14], rely on specific language features and exploit the compiler to generate optimized code from high-level abstractions. The Broadway [10] compiler can be viewed as a specific instance of a system for supporting active libraries. Broadway gives domain-specific compiler optimizations based on user-specified annotation files expressing domain expertise.

Metaprogramming techniques. *Expression templates* furnish a C++ metaprogramming technique for passing expressions as function arguments [13]. The Blitz++ library [14] employs expression templates to generate customized evaluation code for array expressions. This approach remedies performance problems due to the noncomposability of operations when using traditional libraries such as the BLAS or language features such as operator overloading.

Programmable syntax macros [17] deliver a portable mechanism for extend-

ing a general-purpose compiler; they enable the person writing the macro to act as a compiler writer. The macro language is C, extended with abstract syntax tree (AST) types and operations on ASTs. While programmable syntax macros are general and powerful, the software developer must have significant compiler writing expertise in order to implement desired language extensions.

A *metaobject protocol* (MOP) [9, 1] is an object-oriented interface for programmers enabling them to customize the behavior and implementation of programming languages. MOPs enable control over the compilation of programs. For example, a MOP for C++ can provide control over class definition, member access, virtual function invocation, and object creation.

Our annotations approach differs from these metaprogramming techniques in that it is meant to be easily extensible without requiring a developer to have compiler expertise. Because of its generality and extensibility, it is not specific to a particular library, domain, or programming language.

Domain-specific languages and compilers. Domain-specific languages (DSLs) provide specialized syntax that raises the level of abstraction for a particular problem domain. Examples of DSLs include YACC for parsing and compilers, GraphViz for defining directed graphs, and Mathematica for numerical and symbolic computation. DSLs can be stand-alone and used with an interpreter or compiler, or they can be embedded in a general-purpose language (e.g., as macros) and preprocessed into the general-purpose language prior to compilation. At a higher level of abstraction, the Telescoping Languages project [4, 8, 7] defines a strategy for generating high-performance compilers for scientific domain languages. To date, these efforts have focused on extensions to Matlab as defined by domain-specific toolboxes. Our annotations approach includes the use of an embedded language, but it is a more general, extensible language, not a domain-specific one.

Unlike compiler approaches, we do not implement a full-blown compiler or compiler generator; rather, we define a precompiler that parses the language-independent annotations and includes code generation for multiple general-purpose languages, such as C and Fortran.

User annotations are used for other performance-related purposes not directly related to code optimization. One example is *performance assertions* [16], which are user annotations for explicitly declaring performance expectations in application source code. A runtime system gathers performance data based on the user's assertion and verifies this expectation at runtime. Unlike our annotations system, this approach does not guide or perform any code modifications; rather, it automates the testing of performance properties of specific portions of complex software systems. The Broadway [10] compiler mentioned earlier also employs annotations to guide the generation of library calls. Thus, annotation files are associated with a particular *library*, and each library specifies its own analysis problems and code transformations. Significant compiler expertise is needed in order to create an annotation file for a given library. By contrast, the performance annotations we describe in this chapter are more general, with a simpler syntax, and are meant to be associated with partic-

ular, usually small, *code fragments* within arbitrary applications. Little or no compiler expertise is required of the program developer in order to use performance annotations to specify code optimization hints.

1.5 Summary and Future Directions

We have described the initial implementation of an annotation-based performance tuning system that is aimed at improving both performance and productivity in scientific software development. The annotations language is extensible and embeddable in general-purpose languages. We have demonstrated performance improvements in several computational kernels. We are working with a few application developers to apply annotation-based tuning to their applications and plan to release the annotations tool for general use in the very near future.

The annotation work described in this chapter is at an early stage, and our design and implementation are evolving in several directions. We are currently incorporating support for automated generation and execution of multiple tuned versions of code (e.g., for different loop unrolling factors), which requires interaction with multiple job schedulers. We are working on full support for Fortran code generation and are considering new architecture-specific optimizations for platforms other than the Blue Gene/L. We also plan to expand and improve existing code generation modules; for example, we can further speed the Blue Gene/L simdized code by exploiting *common subexpression elimination* (CSE), a typical compiler optimization approach used to reduce the number of operations, where intermediates are identified that can be computed once and stored for use multiple times later. We have already developed an exhaustive CSE algorithm that is guaranteed to find optimal solutions. However, the exponential growth of its search time makes an exhaustive search approach prohibitively expensive for solving complex arithmetic equations. Therefore, one of our objectives is to develop one or more heuristic CSE algorithms that are able to find a near-optimal solution in polynomial time. Longer-term, our research objectives include the addition of new types of annotations that enable high-level specification and tuning of tensor operations and other domain-specific computations.

In addition to code optimizations targeting single-processor performance, we plan to expand our annotation language with syntax for distributed operations and data structures commonly used in scientific computing, such as parallel grid updates for problems discretized on a regular grid. In that case, the user annotation will describe the grid at a very high level, using global dimensions and the type and width of the stencil used. Then, we will define high-level annotations for initialization and point update using global

grid coordinates (i.e., using basically sequential code). The job of the annotation processor would be to take the annotated source code and generate efficient parallel implementation of the distributed operations expressed in the annotations and global-coordinate code. An advantage of annotations over other language-based approaches is that the data structure support can be customized to the application. For example, support for staggered grids or C-grids (semi-regular grids with special properties, particularly at the boundaries) can be added quickly with an annotations-based approach.

Acknowledgments

This material is based on work supported by the U.S. Defense Advanced Research Projects Agency and by the U.S. Department of Energy under Contract DE-AC02-06CH11357. We thank Dinesh Kaushik of Argonne National Laboratory for performing some of the early performance studies of annotation-based performance optimization. We also thank Gail Pieper of Argonne National Laboratory for her comments and corrections.

References

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

[2] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Proceedings of the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39, Berlin, 2000. Springer-Verlag.

[3] Engineering scientific subroutine library (ESSL) and parallel ESSL. `http://www-03.ibm.com/systems/p/software/essl.html`, 2006.

[4] Ken Kennedy et al. Telescoping Languages Project description. `http://telescoping.rice.edu`, 2006.

[5] Kazushige Goto. High-performance BLAS by Kazushige Goto, 2007. `http://www.tacc.utexas.edu/~kgoto/`.

[6] Kazushige Goto and Robert van de Geijn. High-performance implementation of the Level-3 BLAS. Technical Report TR-2006-23, The University of Texas at Austin, Department of Computer Sciences, 2006.

[7] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of IPDPS 2000*, May 2000. CD-ROM Proceedings.

[8] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

[9] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), 1991.

[10] Calvin Lin and Samuel Z. Guyer. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, July 2005.

[11] John McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/`, 2006.

0-8493-0052-5/00/$0.00+$.50

[12] Peter Messmer and David L. Bruhwiler. A parallel electrostatic solver for the VORPAL code. *Comp. Phys. Comm.*, 164:118, 2004.

[13] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[14] Todd L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*. Birkhauser (Springer-Verlag), Boston, 1997.

[15] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, Computer Science Department, May 2004.

[16] J. Vetter and P. Worley. Asserting performance expectations. In *Proceedings of the SC2002*, 2002.

[17] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

[18] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. `http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Whaley814/INDEX.HTM`, 1998.

[19] Using the XL compilers for Blue Gene. `http://www-1.ibm.com/support/docview.wss?uid=pub1sc10431000`, 2006.