MODULARIZING SCIENTIFIC LIBRARIES WITH ASPECT-ORIENTED AND GENERATIVE PROGRAMMING TECHNIQUES

Suman ROYCHOUDHURY*, Jeff GRAY**, Jing ZHANG***, Purushotham BANGALORE**, Anthony SKJELLUM**

*School of Information Technology, International University in Germany, Bruchsal 76646, Germany, Email: Suman.Roychoudhury@i-u.de

** Department of Computer and Information Sciences, University of Alabama at Birmingham, 115A Campbell Hall, 1300 University Boulevard, Birmingham, AL, USA, 35295-1170, tel. (+1) 205 934 2213, E-mail: { gray, puri, tony}@cis.uab.edu

*** Software and Middleware Research, Motorola Applied Research and Technology Center 1295 E. Algonquin Road, Schaumburg, IL, USA 60196 E-mail: j.zhang@motorola.com

ABSTRACT

Scientific computing libraries represent complex software that are often difficult to understand, evolve, and maintain. As systems become larger and more complex with additional requirements, they are subject to decay over a period of time, making it increasingly difficult to address changing stakeholder requirements. New approaches for software engineering and programming language design, such as aspect-oriented software development and generative programming, have been investigated recently as effective techniques for improving modularization of software. In particular, aspects have the potential to interact with many different kinds of language constructs in order to modularize crosscutting concerns. This paper presents an analysis of Blitz++, which is a well-known C++ class library for scientific computing. The analysis demonstrates through various examples how aspect-oriented and generative programming techniques can be applied in modularizing crosscutting concerns that have been identified in Blitz++. This improves software maintainability by reducing code size and de-coupling crosscutting concerns to enable easier change evolution.

Keywords: *aspect-oriented programming, generative programming, scientific computing, software engineering.*

1. INTRODUCTION

To support software adaptation and evolution, new approaches, such as Aspect-Oriented Software Development (AOSD) [1], have shown initial promise in assisting a developer in isolating points of variation and configurability. It has been recognized within the past decade that software modularization constructs available in traditional object-oriented programming languages are inadequate for capturing certain behaviors in a software system that are crosscutting [1, 4]. Although objects can separate certain kinds of concerns (e.g., data or behavior), they fail to encapsulate concerns that overlap, interact with, and cut across the dominant modules in a system. Such concerns are said to be crosscutting and their representation is scattered and tangled among the description of numerous other concerns.

Crosscutting concerns are treated as second class citizens in most languages as there is no explicit representation for their modularization. For example, the common example of logging the method entry and exit points in a very large system may lead to scattering of the logging concern across other useful features present in the code base. This may introduce unnecessary cohesion in the system resulting in poor modularization. In addition to logging, examples of other crosscutting concerns that are difficult to modularize using traditional object-oriented languages are security checks, transaction management, pre-fetching and disk quota operations [3]. It is often desirable to have a way to create a single separate module that describes all of the functionality of a crosscutting concern.

Aspect-oriented techniques provide new language constructs to cleanly separate concerns that crosscut the modularization boundaries of an implementation. In a fundamentally new way, aspects permit a software developer to quantify, from a single location, the effect of a concern across a body of code, thus improving the modularization of crosscutting concerns. Some of the constructs typical in Aspect-Oriented Programming (AOP) approaches, such as AspectJ [4], include the following:

Join Point: Specific execution point of a program, such as a method invocation or a particular exception being thrown.

Pointcut: Means of identifying a set of join points through a predicate expression.

Advice: Defines actions to be performed at associated join points.

Aspect: A modularization of a concern for which the implementation might otherwise cut across multiple boundaries; generally defined by pointcuts and advice.

An aspect weaver is a translator that is responsible for merging the separated aspects with the base code. AspectJ [4] is an aspect weaver for Java. As an example, Listing 1 shows a simple logging aspect in AspectJ. The pointcut `log` captures all join points that correspond to calls on `FooObject` methods (the * represents a wildcard). The `before` and the `after` advice binds the pointcut to specific actions to be performed just before and after each join point is reached (i.e., before and after `FooObject's` method call invocation).

The entire crosscutting concern is captured in a single aspect called `Logger`. The different constructs introduced in this example illustrate the benefit of AOP in modularizing systems that exhibit such crosscutting. Thus, instead of a scattered representation of the logging

concern which may spread throughout the code base, aspects help to modularize concerns that are crosscutting in nature. This directly improves software maintainability by reducing the code size and improving the comprehensibility. Separation of crosscutting concerns makes the system easier to change and evolve. As an indirect effect, reduced code may lead to a smaller memory footprint that can increase the performance of scientific computing software.

```
1. aspect Logger {
2.   pointcut log():
3.     call(public * FooObject.*(..));
4.   before(): log() {
5.    System.out.println("before " +
6.        "calling FooObject methods");
7.    }
8.   after(): log() {
9.     System.out.println("after " +
10.       "calling FooObject methods");
11.  }
12.}
```

Listing 1 AspectJ specification to capture logging in FooObject Methods

As a complementary approach to AOP, generative programming is a software engineering technique that is used to automatically synthesize end-products from a set of primitive reusable components that can be appropriately configured from high-level specifications [2]. Generative programming utilizes automated source code creation through meta-programming, generic classes, templates, aspects, and code generators to improve programmer productivity.

Scientific computing was an initial application domain for the early examples of AOP [5]. However, aside from an application of AspectJ [4] to an implementation of JavaMPI [6], AOP has not been applied or investigated deeply within the area of scientific computing. This is primarily due to the fact that such applications are typically written in FORTRAN, C, or C++, but the center of AOP research has largely remained focused on Java-based implementations. Nevertheless, there is a strong potential for impact if aspects can be used to improve the modularization of scientific computing applications written in languages other than Java.

In this paper, we focus on C++ template libraries that are tailored for scientific computing applications. Such libraries typically rely heavily on parametric polymorphism to specialize mathematical operations on vectors, arrays, and matrices [7, 8]. To demonstrate the benefits of applying aspect-oriented and generative programming techniques to applications written in C++ templates, this paper presents an investigation of applying AOP to a well-known scientific computing library called Blitz++ [8]. The paper illustrates several examples of crosscutting concerns in Blitz++ and explains how they can be encapsulated using aspects. In addition to AOP related examples, we also show how generative programming techniques can be applied to generate large code blocks that represent several arithmetic operations in Blitz++.

1.1. Overview of paper contents

The paper is organized as follows. Section 2 introduces Blitz++ for discussing crosscutting concerns that exist in scientific computing libraries. Section 3 illustrates how generative programming techniques can be used to synthesize several mathematical functions that are required by the Blitz++ library. Section 4 presents a summary of results highlighting the benefits of achieving improved modularization. A comparison to related work is covered in Section 5. A conclusion offers summary remarks and a vision for future work.

2. ASPECTS IN SCIENTIFIC LIBRARIES

This section focuses on the benefits of modularizing open-source libraries written in the scientific computing domain. The contribution highlights the crosscutting features of a template library for scientific computing and describes improved modularization using aspect-oriented and generative programming techniques. In particular, this section illustrates several aspects that are identified in Blitz++ [8], which is a C++ template library that supports high performance scientific computing.

2.1. Background: Crosscutting in Blitz++

Optimizing performance, while preserving the benefits of programming language abstractions, is a major hurdle faced in scientific computing [7, 9, 10]. Object-Oriented Programming Languages (OOPLs) have popularized useful features (e.g., inheritance and polymorphism) in the development of complex scientific problems. However, the performance bottleneck associated with OOPLs has been a major concern among High-Performance Computing (HPC) researchers. Alternatively, languages such as FORTRAN have dominated the numerical computing domain, even though the primitive programming constructs in such languages make applications difficult to maintain and evolve.

Compiler extensions (e.g., High Performance C++ [11] and High Performance Java [12]) and scientific libraries (e.g., POOMA [13], MTL [14], and Blitz++ [8]) have been developed to extend the benefits of object-oriented programming to the scientific domain. In particular, Blitz++ is a popular scientific package that has specific abstractions (e.g., arrays, matrices, and tensors) that support parametric polymorphism through C++ templates. The goal of the Blitz++ project was to develop techniques that enable C++ to compete or exceed the speed of FORTRAN for numerical computing. Blitz++ arrays offer functionality and efficiency, but without any language extensions. The Blitz++ library is able to parse and analyze array expressions at compile-time and perform loop transformations. Blitz++ currently provides dense vectors and multidimensional arrays, in addition to matrices, random number generators, and tiny vectors. The overall size of the Blitz++ library is approximately 115K source lines of code (SLOCs). Moreover, there are several additional source code directories that serve as benchmarks and test cases.

Although Blitz++ makes extensive use of templates for array and vector implementation, the issue addressed in

this paper is the ability to apply AOP concepts to large scientific template libraries like Blitz++. This section contains a description of some of the array and vector implementation templates in Blitz++, and identifies several crosscutting features in the current Blitz++ implementation. The general approach could be applied to other libraries that use parametric polymorphism implemented in languages such as Ada or Java.

The first example (Section 2.2) represents the common case of a debugging precondition that appears in array-impl.h and resize.cc. These files primarily represent arrays whose dimensions are unknown at compile-time and require resizing during runtime. In addition, there are several methods that perform block reduction operations and conversions to and from matrix and vector.

A second crosscutting feature in array-impl.h is `setupStorage`, which is used for initial memory allocation for arrays and appears in both array-impl.h and resize.cc.

The third example (Section 2.3) is based on redundant assertion checks on the lower and upper bounds of an array during instantiation. It appears 46 times in array-impl.h and is similar in concept to redundant assertion checking described by Lippert and Lopes [15].

Section 3 examines AOP combined with generative programming techniques [2]. In particular, this section explores the various binary and unary operations on vectors that use templatized mathematical functions. These functions crosscut the vector operations. For example, many mathematical functions (e.g., sin, cos, tan, abs) appear multiple times in both vecuops.cc and vecbops.cc. Although Blitz++ currently generates these templates, an alternative approach is shown that uses program transformation rules to generate source code. Using this approach, over 12K SLOCs are generated from just 60 lines of code in a base template.

2.2. Precondition and `setupStorage` aspects

The Blitz++ library has a debugging mode that is enabled by defining the preprocessor directive `BZ_DEBUG`. In this mode, an application executes slowly because Blitz++ performs precondition and bounds checking on the array index. Under this condition, if an error or fault is detected by the system, the program halts and displays an error message. Listing 2 shows a sample precondition check for an array implementation. The rank of the vector influences the precondition to be checked.

Another aspect that crosscuts the array implementation boundaries is `setupStorage`. The method is called to allocate memory for any new array. However, any missing length arguments will have their value taken from the last argument in the parameter list. For example, `Array<int,3> A(32,64)` will create a 32x64x64 array, which is handled by the routine `setupStorage`. Both the `BZPRECONDITION` (lines 7 and 15 of Listing 2) and `setupStorage` (lines 9 and 17) can be individually considered as two different pieces of advice applied to the same pointcut (the former as `before` advice and the latter as `after` advice).

Listing 3 presents the corresponding aspect specification for the crosscutting concern identified in Listing 2. This allows the separation of crosscutting concerns from the base code (Listing 2) and encapsulates them as aspects (Listing 3) to be woven using a low-level translator and weaver.

The `BZPRECONDITION` statement (line 4 in Listing 3) and the `setupStorage` statement (line 7 in Listing 3) form part of the `before` and the `after` advice. Note that the `before` and the `after` advice are similar to the `Logger` example shown in Listing 1, but with a different objective.

The pointcut `ArrayConstructor` refers to execution of all `Array` constructors defined in any `Array` type (denoted by the wildcard *). However, if it is desired to match only arrays of type `Array<int>`, more selective pointcuts can be used. The function call `thisJoinPoint.getArgs().length` will return the length of the parameter list in the `Array` constructor.

```
1.  template<typename T_expr>
2.    _bz_explicit Array (_bz_ArrayExpr<T_expr> expr);
3.  Array(int length0, int length1,
4.        GeneralArrayStorage<N_rank> storage = GeneralArrayStorage<N_rank>())
5.   : storage_(storage)
6.   {
7.     BZPRECONDITION(N_rank >= 2);
8.     // implementation code omitted
9.     setupStorage(1);
10.  }
11.  Array(int length0, int length1, int length2,
12.        GeneralArrayStorage<N_rank> storage = GeneralArrayStorage<N_rank>())
13.  : storage_(storage)
14.  {
15.    BZPRECONDITION(N_rank >= 3);
16.    // implementation code omitted
17.    setupStorage(2);
18.  }
```

Listing 2 Precondition check and setpupStorage in Blitz++ array implementation

```
1. aspect InsertBZPreCon_MemAllocation {
2.  pointcut ArrayConstructor(): execution(Array<*>::Array(..));

3.  before(): ArrayConstructor() {
4.     BZPRECONDITION(N_rank  >= thisJoinPoint.getArgs().length());
5.  }
6.  after(): ArrayConstructor() {
7.     setupStorage(thisJoinPoint.getArgs().length()-1);
8.  }
9. }
```

Listing 3 Aspect specification for precondition and memory allocation in templates

2.3. Redundant assertion checking

Another crosscutting feature present in Blitz++ is assertion checking, which is used to evaluate the size or range of array instances. To detect errors in ranges, each array allocation makes an implicit call to `assertInRange`, which checks the lower and upper bounds of an array instance.

This particular assertion is defined in all array template specifications, according to a general pattern as shown in Listing 4 (e.g., `assertInRange` in lines 3 and 7). However, note that the number of index parameters passed to the `assertInRange` routine implicitly depends on the size of the TinyVector. For example, as presented in Listing 4, to allocate a TinyVector of size 1 requires only one parameter (i.e., `index[0]`) to be passed to `assertInRange`. Similarly, for a different allocation size of N, the range will be checked on `index[0]`, `index[1]`, … , `index[N-1]`. This kind of assertion is repeated 46 times in array-impl.h and is context-dependent on the size of each template container.

To avoid the crosscutting assertion checking in every definition of an array implementation, the aspect specification (as defined in Listing 5) will weave this concern into the template code. The `operator` pointcut refers to all operator methods in the array implementation class. The `getParamList` special construct (line 7 of Listing 5) returns the list of index parameters for each call to `assertInRange`.

2.4. Low-Level Implementation using Program Transformation

In our approach, the low-level aspect weaving is achieved using a program transformation engine called the Design Maintenance System (DMS) [16]. The work is similar to our previous investigation of aspects with ObjectPascal [17]. However, the contribution described in this paper targets a different language domain that is applied to scientific computing. Figure 1 presents an overview of the automated process for implementing aspect weaving for C++ templates.

```
1. template<int N_rank2> T_numtype operator()
2.                       (TinyVector<int,1> index) const {
3.     assertInRange(index[0]);
4.     return data_[index[0] * stride_[0]];
5. }
6. T_numtype operator() (TinyVector<int,2> index) const {
7.     assertInRange(index[0], index[1]);
8.     return data_[index[0] * stride_[0] +  index[1] * stride_[1]];
9. }
```

Listing 4 Redundant assertion check on base template specification

```
1. aspect AssertInRange {
2.
3.   pointcut operator ():
4.    execution(Array<*>::operator()(..));
5.
6.   before(): operator() {
7.     assertInRange(thisJoinPoint.getParamList());
8.   }
9. }
```

Listing 5 Aspect specification for redundant assertion checking

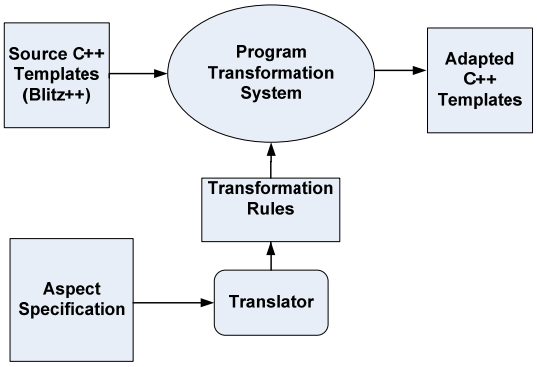

Fig. 1 Low-level infrastructure to perform aspect weaving for C++ template libraries

One of the major processes involved in the implementation is the translator (bottom of Figure 1), which parses and translates a high-level aspect language into low-level transformation rules. These rules are specific to the underlying transformation system and perform the actual weaving.

However, as these rules are synthesized by the translator, their existence is oblivious to end-users who can specify their intention using the high-level aspect language. The heart of the weaving process is the DMS transformation engine, which takes the source C++ template and translated rule as input and generates the adapted C++ file as output. In our work, aspect mining and removal of the original crosscutting concern was performed manually, although the actual weaving is automated using the high-level aspect specification and the low-level transformation infrastructure shown in Figure 1.

3. CROSSCUTTING GENERIC FUNCTIONS

This section discusses the combination of AOP with other generative programming techniques [2]. In Blitz++, templates such as binary and unary operations for arrays and vectors are synthesized from a code generator implemented in several C++ routines. For consideration in this section, attention is focused on a specific set of unary vector (mathematical) operations in a template specification, which are generated to the vecuops.cc source file in the Blitz++ library containing approximately 12K SLOCs. Most of these mathematical operations (e.g., log, sqrt, sin, floor, fmod) have the same syntactic structure and can be specified concisely in the form of a pattern. An analysis of the generation process revealed that the entire template specification is essentially a cross-product between the set of defined mathematical operations (λ) and a base template (β) that represents the general pattern structure. As observed, the set of mathematical functions crosscut the entire unary vector general pattern.

If λ1,λ2,...λn represent the set of mathematical operations (e.g., log, sin, sqrt) that crosscut the partial base template structure β (whole of Listing 6), then the code generated as the cross-product of λ and β can be represented as λ1β + λ2β +... + λnβ. The partial string identifier `OPERATION` (highlighted in bold in Listing 6) identifies the locations in the partial base template structure where the mathematical operations must be woven to generate the whole template structure (i.e., ∑ λ x β = 12k SLOCs). The concept here is somewhat different than standard AOP practice and more analogous to generative programming, but the idea of a cross-product between a set of mathematical operations and a base pattern is germane to the overall process of template weaving. Although this example is based on vector operations using mathematical functions, similar situations (e.g., operations on Blitz++ arrays) exist in several other generated template specifications in the Blitz++ library.

The transformation rule describing the weaving of the mathematical functions with the base pattern is shown in Listing 7. The first line of the rule identifies the programming language (base domain) of the transformed source, which is C++ in Blitz++. Lines 3-10 use patterns for matching a syntax tree with a specified structure. The rule as shown in Line 13 describes a directed pair of corresponding syntax trees.

Such rules are typically provided as a rewrite specification that maps from a left side (source) syntax tree expression to a right side (target) syntax tree expression. The right side of the rule specification uses an external function (i.e., `generate_template_code` in Line 17) to generate code. The first parameter to this external function is a template definition (β). The second and third parameters are the two annotated markers in the base tree that need to be replaced with the enumerated mathematical operations.

```
1.    template<class P_numtype1>
2.    inline _bz_VecExpr   <_bz_VecExprUnaryOp <VectorIterConst<P_numtype1>,
3.                          _bz_OPERATION<P_numtype1>>>
4.
5.    OPERATION(Vector<P_numtype1>& d1)
6.    {
7.      typedef bz_VecExprUnaryOp <VectorIterConst<P_numtype1>,
8.                                 _bz_OPERATION<P_numtype1>> T_expr;
9.      return _bz_VecExpr <T_expr> (T_expr (d1.begin()));
10.   }
```

Listing 6 Subset of base pattern used to generate the vector operation template

```
1  default base domain Cpp.
2
3  pattern aspect_op():
4          identifier = "OPERATION".
5  pattern aspect_bz_op():
6          identifier = "_bz_OPERATION".
7
8  pattern op1(): identifier = "log".
9  pattern op2(): identifier = "sin".
10 pattern op3(): identifier = "sqrt".
11 ...
12
13 rule generate_vec_template
14   (td:template_declaration):
15   declaration_seq -> declaration_seq
16 = td ->
17   generate_template_code(
18       td, aspect_op(), aspect_bz_op(),
19       op1(), op2(), op3(),...
20    )
```

Listing 7 Rules used to generate mathematical operations using a base template definition

The fourth and subsequent parameters are the set of generic mathematical operations (e.g., log, sin, sqrt) to be woven into the base pattern during code generation. Using the above rule specification and the base pattern as shown in Listing 6, nearly 12K source lines of code are generated which resembles the entire set of unary mathematical operations present in the Blitz++ library.

4. SUMMARY OF RESULTS

In this section, we briefly summarize the results of modularizing Blitz++. The first example (Section 2.2) showed how a "precondition check" on array indices could be encapsulated into a single aspect, which otherwise appeared 25 times in array-impl.h and in 57 places in resize.cc.

The second crosscutting example described in that section was `setupStorage`, which was scattered throughout the implementation details of array-impl.h and resize.cc in over 23 separate locations in each file. Although memory allocation for arrays is an altogether different concern, coupling it with array implementation deeply reduces the chances of the system to evolve over time. For example, if memory allocation for arrays tends to be different for a particular platform, the call to `setupStorage` needs to be updated 23 times in both template classes.

The third example (Section 2.3) illustrated how assertion checks on the lower and upper bounds of an array could be modularized using aspect-oriented techniques. Note that it was redundantly called 46 times in array-impl.h, but AOP techniques assisted in isolating this concern into one single module that captured the same functionality.

Section 3 demonstrated the benefits of generative programming. As shown in that section, over 12K SLOCs resembling several unary mathematical operations on vectors were generated from a base pattern and a transformation rule which together represented just 60 lines of code and transformation rules. The above techniques not only helped to reduce the code size for better maintainability and improved modularization, but also offered an indirect effect of reducing the memory footprint of the Blitz++ library.

In addition to the examples described in this paper, there are several other bottlenecks in Blitz++ that can benefit from AOP and generative programming. For example, another crosscutting concern represents a mapping of array indices. The helper class used to implement this mapping is specialized for ranks 1, 2, 3, ..., 11. However, instead of redundantly writing the specialization code for each individual rank, the specialization code is itself captured in a single aspect and applied universally over all array ranks. Likewise, in addition to unary mathematical operations on vectors, binary operations could be similarly generated in a manner described in Section 3.

5. RELATED WORK

A discussion of templates and aspects in AspectC++ within the context of generative programming is discussed in [18]. The focus of the AspectC++ work is on the interesting notion of incorporating parametric polymorphism into the bodies of advice. In contrast, the focus of our contribution is a discussion of the complimentary idea of weaving crosscutting features into the implementation of scientific template libraries.

Within the scientific computing domain, ROSE provides optimizations using source-to-source transformation of ASTs for C++ applications [10]. The transformations are expressed using a domain-specific language [19]. The type of transformations performed by ROSE are focused solely on optimization issues of scientific libraries and are not applicable to the kinds of transformations we advocate in this paper to improve the modularization of crosscutting concerns in scientific code bases.

AspectJ provides support for generics and parameterized types in pointcuts and intertype declarations for Java [22]. In order to restrict matching of patterns within given parameter types (for methods and constructors), return types (for methods) and field types, an appropriate parameterized type pattern is specified in the signature pattern of a pointcut expression. However, since AspectJ is bound to the Java programming language, it does not support scientific libraries written in non-Java based applications.

6. CONCLUSION

The crosscutting concerns present in Blitz++ and introduced in Section 2 highlight the benefits that aspect-orientation can bring to scientific computing libraries. Because our weaver is based on source-to-source translation at pre-compile-time, there was no impact on the performance of the Blitz++ library as a result of aspect weaving. Given the historical tendency of languages to evolve by adopting new paradigms, it is reasonable to assume that aspect-oriented and generative programming concepts will be integrated into many more applications of scientific computing.

The work described in this paper primarily focussed on modularizing libraries written in C++ templates, which has received the least amount of attention from the software engineering community. Furthermore, it is our belief that the concepts and examples described here apply to scientific computing libraries other than Blitz++, as well as libraries written in other languages that support parametric polymorphism (e.g., Java and Ada). Future directions will involve extending the focus to other scientific libraries that are implemented in C++ (e.g., POOMA [13], MTL [14]). An interesting notion of crosscutting concern that may apply within the scientific computing domain is to identify parallelism within blocks of sequential code. In [20], Chalabine and Kessler have suggested seven different forms of interdependent concerns that are necessary to introduce parallelism within sequential programs. Future work will explore this idea in other scientific libraries.

Another interesting topic serving as future work is library-independent aspects that may exist within a specific domain, such as scientific computing, but applicable to several different libraries. In particular, the research will focus on modularizing High Performance Linpack (HPL) libraries designed for benchmarking high performance distributed memory computers [21].

ACKNOWLEDGEMENT

This work is sponsored in part by NSF grant CCF-0350463.

REFERENCES

[1] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2004.

[2] Krzysztof Czarnecki and Ulrich Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[3] Yvonne Coady and Gregor Kiczales, “Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code”, International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, March 2003, pp. 50-59.

[4] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, “Getting Started with AspectJ,” Communications of the ACM, October 2001, pp. 59-65.

[5] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman, “Aspect-Oriented Programming of Sparse Matrix Code,” International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Springer-Verlag LNCS 1343, Marina del Ray, CA, December 1997, pp. 249-256.

[6] Bruno Harbulot and John Gurd, “Using AspectJ to Seperate Concerns in a Parallel Scientific Java Code,” International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, March 2004, pp. 122-131.

[7] Anthony Skjellum, Purushotham Bangalore, Jeff Gray, and Barrett Bryant, “Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software,” ICSE 2004 Workshop: International Workshop on Software Engineering for High Performance Computing System (HPCS) Applications, Edinburgh, Scotland, May 2004.

[8] Todd Veldhuizen, “Arrays in Blitz++,” 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Springer-Verlag LNCS 1505, Santa Fe, NM, December 1998, pp. 223-230.

[9] Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik, “Parallel Object-Oriented Framework Optimization,” Concurrency: Practice and Experience, February-March 2004, pp. 293-302.

[10] Todd Veldhuizen and Dennis Gannon, “Active Libraries: Rethinking the Roles of Compilers and Libraries,” SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, Yorktown Heights, NY, October 1998.

[11] Elizabeth Johnson and Dennis Gannon, “HPC++: Experiments with the Parallel Standard Template Library,” International Conference on Super-computing, Vienna, Austria, July 1997, pp. 124-131.

[12] Vladimir Getov, Susan Flynn Hummel, and Sava Mintchev, “High-performance Parallel Programming in Java: Exploiting Native Libraries,” Concurrency: Practice and Experience, September-November 1998, pp. 863-872.

[13] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn, “POOMA: A Framework for Scientific Simulations of Paralllel Architectures,” in Gregory V. Wilson and Paul Lu, ed., Parallel Programming Using C++, MIT Press, 1996.

[14] Jeremy Siek and Andrew Lumsdaine, “The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra,” Computing in Object-Oriented Parallel Environments (ISCOPE), Springer-Verlag LNCS 1505, Santa Fe, NM, December 1998, pp. 59-70.

[15] Martin Lippert and Cristina Lopes, “A Study on Exception Detection and Handling Using Aspect-Oriented Programming,” International Conference of Software Engineering (ICSE), Limmerick, Ireland, June 2000, pp. 418-427.

[16] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, “DMS: Program Transformation for

Practical Scalable Software Evolution," International Conference on Software Engineering (ICSE), Edinburgh, Scotland, May 2004, pp. 625-634.

[17] Jeff Gray and Suman Roychoudhury, "A Technique for Constructing Aspect Weavers Using a Program Transformation System," International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, March 2004, pp. 36-45.

[18] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk, "Generic Advice: On the Combination of AOP with Generative Programming in AspectC++," Generative Programming and Component Engineering (GPCE), Springer-Verlag LNCS 3286, Vancouver, BC, October 2004, pp. 55-74.

[19] Markus Schordan and Daniel Quinlan, "A Source-To-Source Architecture for User-Defined Optimizations," Joint Modular Languages Conference (JMLC), Springer-Verlag LNCS 2789, Klagenfurt, Austria, August 2003, pp. 214-223.

[20] Mikhail Chalabine and Christoph Kessler, "Crosscutting Concerns in Parallelization by Invasive Software Composition and Aspect Weaving," 39th Hawaii International Conference on System Sciences (HICSS-39), Kauai, HI, January 2006.

[21] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," Version 1.0a http://www.netlib.org/benchmark/hpl.

[22] AspectJ Project, http://www.eclipse.org/aspectj.

Received December 14, 2008, accepted August 1, 2009

BIOGRAPHIES

Dr. Suman Roychoudhury is a Research Associate in the School of Information Technology, International Univerity in Germany. He received his Ph.D. from the University of Alabama at Birmingham (UAB). His research interests are energy-aware embedded systems, aspect-oriented software development, service-oriented architecture and model-driven engineering.

Dr. Jeff Gray is an Associate Professor in the Computer and Information Sciences Department (CIS) at UAB, where he co-directs research in the Software Composition and Modeling (SoftCom) Laboratory. His research interests include model-driven engineering, aspect-orientation, and generative programming.

Dr. Jing Zhang is a research scientist at Motorola Applied Research and Technology Center, where she is responsible for conducting research on policy-based management system. She received her Ph.D. from the CIS Department at UAB. Her Ph.D. research was focused on techniques that combine model transformation and program transformation in order to assist in evolving large software systems.

Dr. Purushotham Bangalore is an Associate Professor in the CIS Department at UAB. He has a Ph.D. in Computational Engineering from Mississippi State University. As Director of the Collaborative Computing Laboratory, Dr. Bangalore undertakes research in the area of Grid Computing Programming Environments.

Dr. Anthony Skjellum is Professor and Chair of the CIS Department at UAB. He received his Ph.D. in Chemical Engineering from the California Institute of Technology. He specializes in reusable, scalable mathematical software, and message passing middleware for scalable, real-time, and fault-tolerant systems.