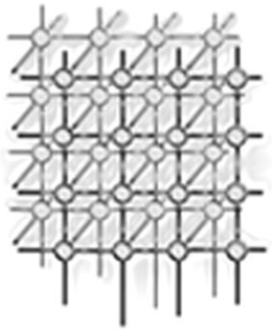
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:293–302 (DOI: 10.1002/cpe.775)



Parallel object-oriented framework optimization

Daniel J. Quinlan[1,*,†], Markus Schordan[1], Brian Miller[1] and Markus Kowarschik[2]

[1] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, U.S.A.*
[2] *System Simulation Group, Department of Computer Science, University of Erlangen–Nuremberg, Germany*

SUMMARY

Sophisticated parallel languages are difficult to develop; most parallel distributed memory scientific applications are developed using a serial language, expressing parallelism through third party libraries (e.g. MPI). As a result, frameworks and libraries are often used to encapsulate significant complexities. We define a novel approach to optimize the use of libraries within applications. The resulting tool, named ROSE, leverages the additional semantics provided by library-defined abstractions enabling library specific optimization of application codes. It is a common perception that performance is inversely proportional to the level of abstraction. Our work shows that this is not the case if the additional semantics can be leveraged. We show how ROSE can be used to leverage the semantics within the compile-time optimization. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: telescoping languages; AST restructuring tools; semantics-based transformations; object-oriented optimizations

INTRODUCTION

This paper demonstrates a compile-time approach using the semantics defined within library abstractions to optimize parallel applications. The specific abstraction chosen for optimization in our example is that of an array abstraction contained within an array class library; however, any library defining high-level abstractions could have been used, nothing in our approach is specific to the type of abstraction. The array abstraction is interesting because it has simple parallel semantics which we will show how to leverage at compile-time. The parallel semantics associated with our simple array

*Correspondence to: Daniel J. Quinlan, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, U.S.A.
†E-mail: dquinlan@llnl.gov

Contract/grant sponsor: U.S. Department of Energy; contract/grant number: W-7405-Eng-48

Copyright © 2004 John Wiley & Sons, Ltd.

Received 20 September 2001
Revised 29 May 2002
Accepted 1 July 2003

```
class Range {
public:
    Range ( int base, int bound, int stride );
    Range operator+ ( int i );
    Range operator- ( int i );
};
class doubleArray {
public:
    doubleArray ( int i, int j );
    doubleArray & operator= ( const doubleArray & X );
    friend doubleArray operator+ ( const doubleArray & X, const doubleArray & Y );
    doubleArray operator() ( const Range & I, const Range & J );
    friend doubleArray & sin ( const doubleArray & X );
};
int main () {
    doubleArray A(100,100), B(100,100);
    Range I(1,98,1), J(1,98,1);
    A(I,J) = B(I-1,J) + B(I+1,J) + B(I,J-1) + B(I,J+1);
}
```

Figure 1. Example: snippet of the header file and program to be optimized.

abstraction could not be expected to be optimized by any serial compiler, moreover all of the specific user-defined semantics defined by any library abstraction are ignored within compilation because the compiler only understands the semantics of the language in which the abstraction is implemented.

The potential for the optimization of abstractions and their interaction depends heavily upon the characteristics of the abstractions. The array abstraction which we use as an example benefits significantly from such optimizations. In this paper we demonstrate our approach using this example, explain what library designers must implement, explain how minimal the impact is on the application developer and include parallel performance results. By way of example, this paper lays out a general approach to the optimization of high-level abstractions found in other libraries.

MOTIVATING EXAMPLE

We use a motivating example from the A++/P++ array class library [1] to show how the ROSE framework can be used by the library writer to develop a preprocessor that optimizes the given example. The example uses two classes which are implemented twice; once in the serial A++ library and again in the parallel P++ library. Within our motivating example we consider the following trivial five-point stencil array operation. In Figure 1, `A` and `B` are multidimensional array objects of type `floatArray`. `I` and `J` are `Range` objects that together specify a two-dimensional index space of the arrays `A` and `B`. This running example is used in the following sections to demonstrate how to optimize a scientific application code. Figure 1 shows a greatly simplified header file associated with an array class library; this is sufficient for preprocessing since our example code only uses a small fraction of the A++/P++ library features, the implementation of these functions is not required for preprocessing.

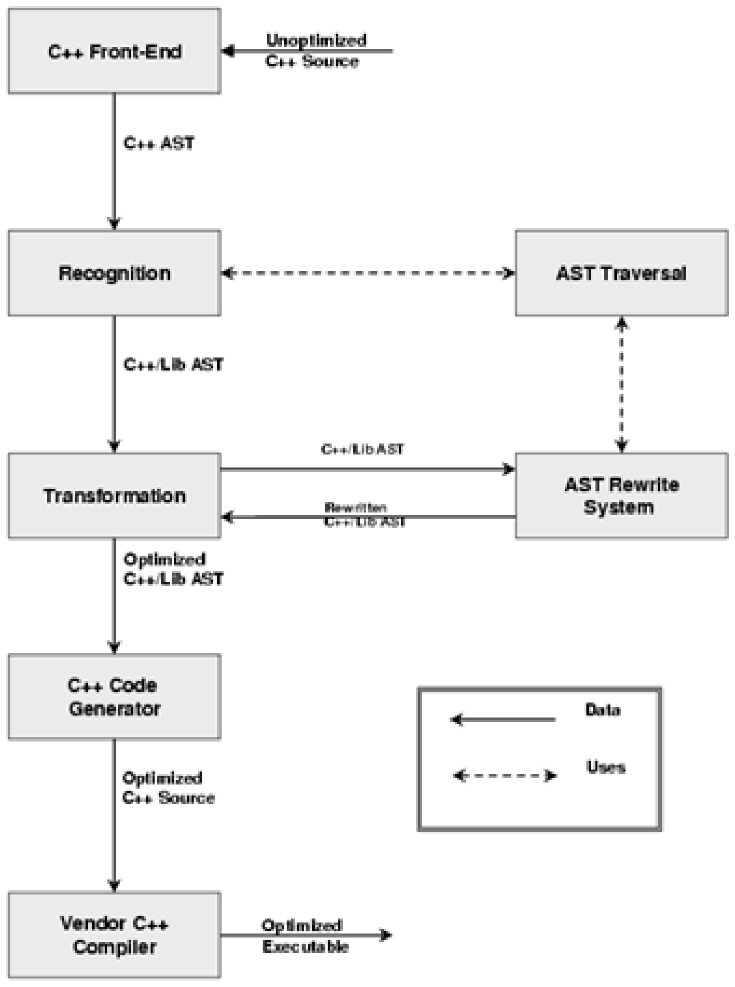


Figure 2. Processing phases of a preprocessor developed using the ROSE framework.

ROSE STRUCTURE

Figure 2 shows the execution order of the different phases of a preprocessor built using the ROSE framework. The ROSE framework allows library specific preprocessors to be built. We outline what the library designer implements and how the application developer uses the preprocessor separately.

- *What a library designer does*
  The library designer writes the preprocessor using the ROSE framework. Our approach

makes minimal demands of compiler design experience on the library designer. To build the preprocessor the library designer specifies all of the transformations (details in the section on the transformation phase) and provides the library's header files to ROSETTA to automatically build the high-level grammar specific to their library.

- *Implements transformation:* a transformation is expressed as a set of semantic actions. A semantic action is associated with a particular type of node in the abstract syntax tree (AST). The tree traversal mechanism within ROSE simplifies the implementation of the transformation by permitting the library designer to specify the semantic actions and allowing the tree traversal mechanism to execute them on each node of the AST. Depending upon the complexity of the transformation, synthesized and/or inherited attributes (see the section on the transformation phase) can be used to pass information up or down within the AST, respectively, throughout the traversal. Nested and/or recursive traversals are also possible and can simplify the implementation of the transformation.
- *Provides a library interface:* based on the information in the library's header files the recognition of library specific abstractions in the application code's AST is automated. Higher-level grammars are automatically built from the library's header files. Higher-level grammars identify the location of abstractions within the AST. The recognition phase marks library specific AST nodes and this simplifies the implementation of library specific transformations.

• *How an application developer uses the preprocessor*
The application developer uses the preprocessor provided by the library designer as a compiler substitution for the vendor's C++ compiler. The preprocessor calls the vendor's C++ compiler as a final step to generate object files. Thus the library designer's preprocessor is a complete replacement for the application developer's compiler and the use of the preprocessor only requires a change of the name of the compiler within an application's makefile. This approach simplifies the testing of the application with and without the preprocessor and supports its optional use within application development.

C++ FRONT END

We currently use the Edison Design Group (EDG) [2] C++ front-end for parsing C++. From the EDG intermediate representation we generate an object-oriented annotated AST and call this intermediate representation Sage III AST (which is based on Sage II [3]).

RECOGNITION PHASE

To simplify the development of transformations and make them compelling for library developers to write, we have separated the tedious aspects of recognizing the library's abstractions within the C++ AST from the transformation of the AST.

The recognition is automated such that nodes in the AST that represent the use of constructs (classes, functions, etc.) defined in the library are marked as library specific. With this greater degree of refinement of the AST, transformations can be written specific to abstractions without complex pattern recognition mechanisms.

The distinction and separation of library statements from non-library statements is often a subtle point. However, the library's interface provides all the information that is needed by ROSE. Figure 3 shows the AST representing the application code in Figure 1. Nodes representing the use of constructs defined in the library, `doubleArray` and `Range` constructs, are marked and shown as grey nodes. According to the different classes in the library different types of nodes are recognized. In the figure we show this by using dark gray for `doubleArray` nodes and light gray for `Range` nodes. However, not all nodes are recognized to be library specific. In Figure 3, `SgIntVal` nodes are not marked (white) because the type `int` is not defined in the library. We call an AST consisting of library specific nodes and C++ nodes a higher-level AST.

Another case arises when functions are used which are not defined in the library. Let `A`, `B` be of type `doubleArray` and `foo` be a function not defined in the library. Then a statement `A = B + foo();` would not be recognized as library specific. In contrast, if `foo` were defined in the library it would be recognized. In our example the overloaded operators `+`, `-` and `=` are functions which are defined in the library and, therefore, the nodes representing function calls of the overloaded functions are recognized to be library specific.

Please note that the use of a member function (i.e. member function call) that is not defined in the library is never marked as library specific—independent of its return type. However, if the function is defined in the library, the 'color' is defined by the class it belongs to. Non-member functions are treated as if they belong to one single class in the library.

The higher-level ASTs are defined by higher-level grammars which are automatically generated by adding rules to the abstract C++ grammar according to the class interfaces defined in the library. The input for this are the library header files (see Figure 1). Since the higher-level grammars are automatically generated, the recognition can be automated.

TRANSFORMATION PHASE

The only step required of the library writer is to define the transformation based on a 'colored' AST. The library specific constructs in a program are marked in the recognition phase. This essential information defines where a transformation can be applied. ROSE aids the library writer by providing a traversal mechanism that visits all the nodes of the AST in a predefined order and computes attributes. Based on a fixed traversal order, we provide inherited attributes for passing information down the AST (top-down processing) and synthesized attributes for passing information up the AST (bottom-up processing). Inherited attributes can be used to propagate context information along the edges of the AST whereas synthesized attributes can be used to compute values based on the information of the subtree. One function for computing inherited attributes and one function for computing synthesized attributes must be implemented when attributes are used. We provide different interfaces that allow that both, only one or no attribute is used; in the latter case it is a simple traversal with a visit method called at each node.

Hence, the traversal mechanism can be used to gather information about the AST, or 'query' the AST. Based on this information, existing source code can be added, replaced and deleted. Creating parts of an AST 'by hand' is not practical for complex transformations and ROSE offers a feature that has

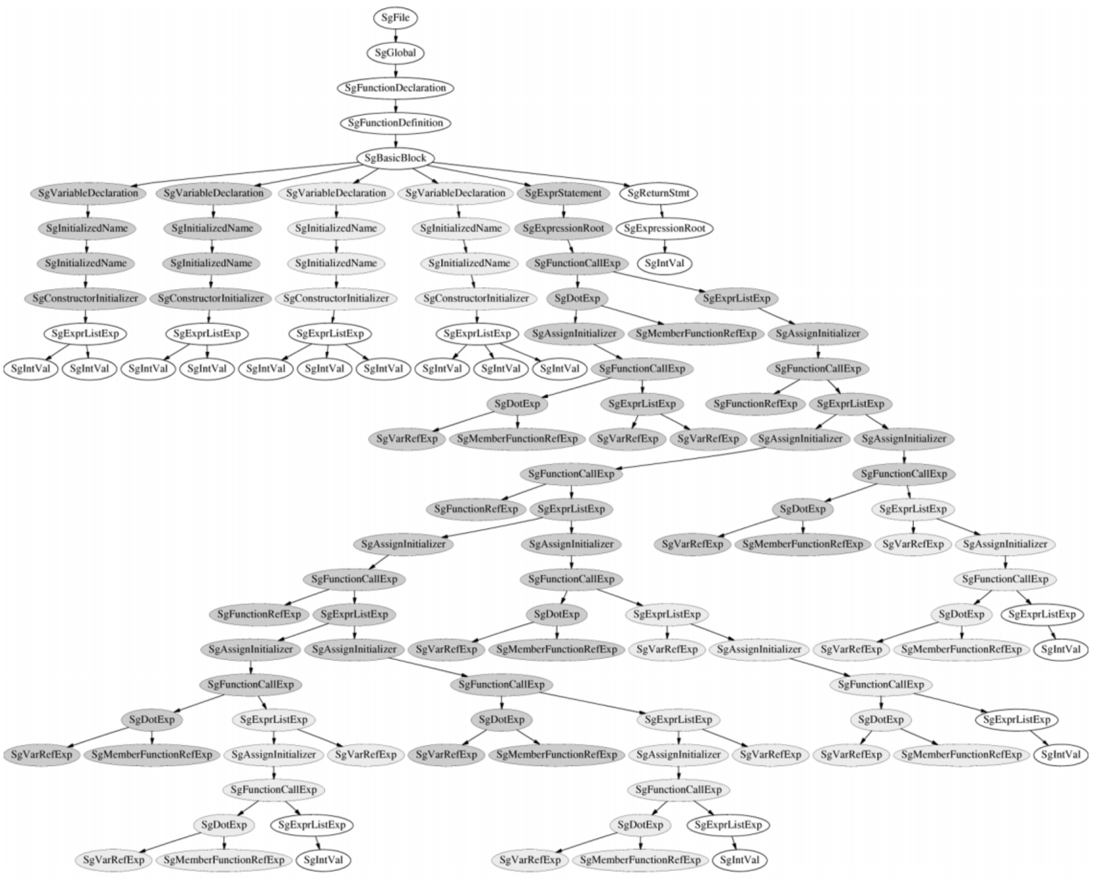


Figure 3. The AST associated with the input program using the array grammar.

proved to greatly simplify the rewriting of the AST. Source-strings can be used to define a subtree to be inserted. The source-string itself can be computed by the use of attributes. Alternatively source-strings may explicitly specify new source. A source-string is passed on to the rewrite system which calls the front-end to create an AST fragment to represent this source-string and inserts it into the AST. During a traversal, AST fragments can be inserted at any node of the AST. The default replace mechanism substitutes the subtree of the current node (with the node as its root node). For example, an identity transformation can be performed by computing the source-string represented by the subtree of the current node as a synthesized attribute and replacing this subtree with a new AST fragment which is generated by using the above-mentioned source-string feature.

The transformations are based on the semantics of the library. Thus, a limitation of our approach is that changes to the semantics of the library require adapting the transformations to the new library

```
ARRAY_OPERAND_UNIFORM_SIZE_DECLARATION_MACRO_D2();
double * restrict _A_I_J_pointer = ((double * )0);
double * restrict _B_I_J_pointer = ((double * )0);
int _1, _2;
int _length1 = 0, _length2 = 0;
class doubleArray A(100,100), B(100,100);
class Range I (1,98,1), J (1,98,1);
#define SC(x1,x2) /* case UniformSizeUnitStride */ (x1)+(x2)*_size1
ARRAY_OPERAND_UNIFORM_SIZE_INITIALIZATION_MACRO_D2(A);
_A_I_J_pointer = (A.getAdjustedDataPointer)(I,J);
_B_I_J_pointer = (B.getAdjustedDataPointer)(I,J);
ARRAY_TRANSFORMATION_LENGTH_INITIALIZATION_MACRO_D2(A);
for (_2 = 0; _2 < _length2; _2++) {
     for (_1 = 0; _1 < _length1; _1++) {
          _A_I_J_pointer[SC(_1,_2)] = (((_B_I_J_pointer[SC((_1 + 1),_2)] +
               _B_I_J_pointer[SC((_1 - 1),_2)]) + _B_I_J_pointer[SC(_1,(_2 - 1))]) +
               _B_I_J_pointer[SC(_1,(_2 + 1))]);
        }
  }
A.updateGhostBoundaries();
```

Figure 4. Example of the transformed A++/P++ code from Figure 1. This code uses macros for array subscript computation and variable initialization. Replacing the high-level array abstractions with direct access to the underlying pointers results in improved performance.

semantics. However, in practice the change of the semantics of a library should be avoided if possible because it may also require that any user code that uses the functionality with changed semantics needs to be adapted as well.

C++ CODE GENERATOR

The previous steps result in the automated generation of an optimized application code which uses the semantics of the library's abstractions as part of the optimization. Figure 1 showed the example application code (a trivial example code) using the array class library abstractions. Figure 4 shows the automatically transformed output code implementing the lower-level details required for superior performance. This transformation results in improved performance because the separate evaluation of the binary operators in the original code is fused into a simple loop in the transformed code, improving temporal locality. The transformation we have used is not particularly sophisticated and much more complex transformations have been implemented but are too large to present as examples (e.g. cache-based transformations).

RELATED WORK

Within ROSE we use *Sage III*, which we have developed as a revision of the *Sage II* [3] AST restructuring tool. *Nestor* [4] is a similar AST restructuring tool for Fortran 77, Fortran 90 and HPF2.0, which, however, does not attempt to recognize and optimize high-level user-defined abstractions. Work son *OpenC++* [5] has led to the development of a C++ tool which is also similar to Sage, but adds some additional capabilities for optimization. It neither addresses the sophisticated scale of abstractions that we target nor the transformations we target.

Related work on the optimization of libraries on *telescoping languages* [6] shares many of the same goals as our research work and we expect to work more closely with these researchers in the near future. Our approach so far is less ambitious than the *telescoping languages* research, but is in some aspects further along, although currently specific to high-level abstractions represented in C++. Further approaches are based on the definition of library-specific *annotation languages* to guide optimizing source code transformations [7] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations; see [8] for example.

Work at the University of Tennessee has led to the development of *Automatically Tuned Linear Algebra Software* (ATLAS) [9]. Within this approach numerous versions of a parameterized transformation are generated to define a search space and the performance of a given architecture is evaluated. The parameters associated with the best performing transformation are thus identified empirically. Preprocessors built using the ROSE framework could take significant advantage of this approach to identify the optimal parameters associated with transformations parameterized on architecture-specific details (e.g. cache-size).

The example shown is similar to what can be done using expression templates techniques [10]. Expression templates work by using the template mechanism defined by C++ and require no additional preprocessing step, but are exceedingly problematic. The general compile-time approach we present is superior, or at worst an alternative, because it provides for sophisticated program analysis, which our simple example does not require, but more complex transformations (e.g. loop fusion) most certainly require. Such program analysis (e.g. dependence analysis) is not possible within the expression template technique because of the limitations inherent in template meta-programming. More information about expression templates and its advantages, disadvantages and limitations can be found in [10–12].

RESULTS

To show the value in using such an automated source-to-source preprocessing tool, we compare the parallel performance of a ROSE-transformed C++ code to a HPF implementation solving the same problem. The C++ code is written using the P++ parallel array class library. Additional versions of this code have been written in HPF, C and using the A++ abstractions locally on each processor with explicit message passing. Using the abstractions available in the P++ library greatly simplifies code development with the resulting source code being quite compact and very easy to understand. We have used a preprocessor built using ROSE to automatically transform the high-level abstractions into code that can be highly optimized by the vendor compiler, thereby achieving high performance[‡].

Our ROSE transformed C++ implementation takes advantage of restricted pointers. That is, pointers are guaranteed to have no aliases. With this assumption, the resulting C code performs equivalently to the FORTRAN 77 implementation [13].

[‡]Minor manual corrections were required to fix current bugs in the automated code generation.

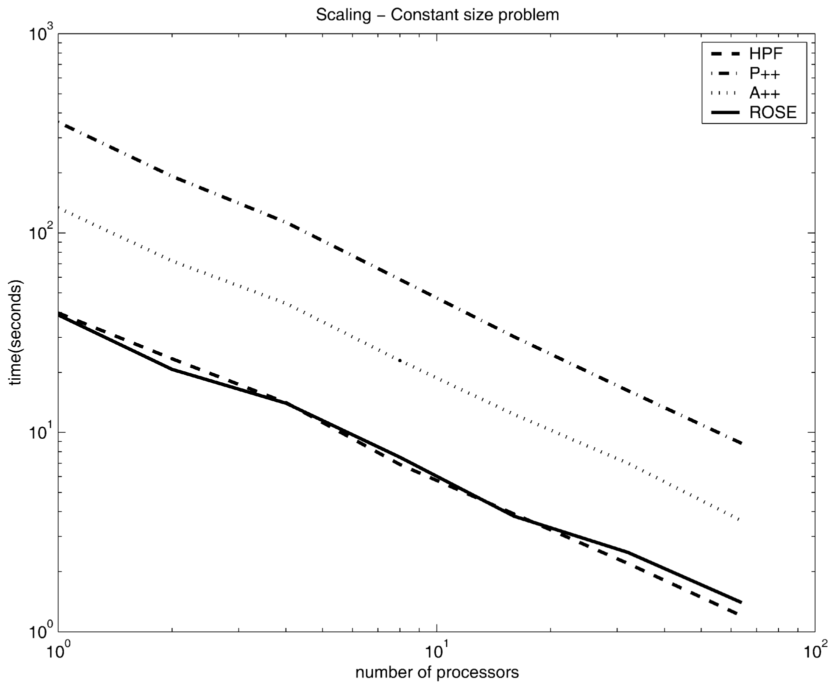


Figure 5. Results showing the performance of the parallel array abstractions before and after transformations.

The test machine is ASCI Blue Pacific at the Lawrence Livermore National Laboratory. This IBM machine consists of 256 compute nodes, each node containing four 332 MHz PowerPC 604e CPUs with 1.5 GB of RAM. We present a performance comparison of the numerical solution of a linear convection equation. Separate codes solving this equation have been written using HPF, P++, A++ and one generated from the A++/P++ code by a preprocessor built using ROSE.

We present a scaling study of these codes in Figure 5. In this test we fix the array size as the number of processors grows from 1 to 64. We see that all versions of the code scale similarly, but only the code containing the ROSE preprocessor generated transformations performs at the same level as its FORTRAN counterpart.

CONCLUSIONS

To recognize where transformations can be automatically introduced it is required that the use of the object-oriented abstractions be identified and classified into specific language/grammatical elements (e.g. expressions, statements, types, etc.). With this level of detail, the AST is refined and abstraction dependent optimizations can be performed. Our work shows that when using ROSE to leverage additional semantics, more information is fundamentally available at compile time. How much information is available depends upon the abstraction. These additional semantics have permitted optimizations resulting in high-level abstractions with performance as good as low-level C and HPF. Significantly better performance should be possible using this additional semantic information in conjunction with cache-based transformations.

ACKNOWLEDGEMENT

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

REFERENCES

1. Lemke M, Quinlan D. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured Grid applications. *Proceedings of the CONPAR/VAPP V*, Lyon, France, September 1992. (*Lecture Notes in Computer Science*). Springer: Berlin, 1992.
2. Edison Design Group. http://www.edg.com.
3. Bodin F *et al*. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. *Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994.
4. Silber G-A, Darte A. *The Nestor Library: A Tool for Implementing Fortran Source to Source Transformations* (*Lecture Notes in Computer Science*, vol. 1593). Springer: Berlin, 1999.
5. Chiba S. Macro processing in object-oriented languages. *Proceedings Conference on the Technology of Object-Oriented Languages and Systems (TOOLS Pacific'98)*, Australia, November, 1998. IEEE Press, 1998. More info available at http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html.
6. Broom B, Cooper K, Dongarra J, Fowler R, Gannon D, Johnsson L, Kennedy K, Mellor-Crummey J, Torczon L. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing* 2000.
7. Guyer SZ, Lin C. An annotation language for optimizing software libraries. *Proceedings of the Second Conference on Domain-Specific Languages*, October 1999. USENIX Association: Berkeley, CA, 1999; 39–52.
8. Menon V, Pingali K. High-level semantic optimization of numerical codes. *Proceedings of the ACM/IEEE Supercomputing 1999 Conference (SC99)*, Portland, OR, 1999. ACM Press, 1999; 434–443.
9. ATLAS homepage. http://www.netlib.org/atlas.
10. Veldhuizen T. Arrays in Blitz++. *Proceedings of the Second International Symposium, ISCOPE 98*, Santa Fe, NM, December 1998 (*Lecture Notes in Computer Science*, vol. 1505). Springer: Berlin, 1998; 223–230.
11. Bassetti F, Davis K, Quinlan D. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, Marina del Rey, CA, December 1997 (*Lecture Notes in Computer Science*, vol. 1343). Springer: Berlin, 1997.
12. Carmesin S, Crotinger J, Cummings J, Haney S. *Array Design and Expression Evaluation in POOMA II* (*Lecture Notes in Computer Science*, vol. 1505). Springer: Berlin, 1998; 231–240.
13. Bassetti F, Davis K, Quinlan D. Toward FORTRAN 77 performance from object-oriented C++ scientific frameworks. *High-Performance Computing '98, Grand Challenges in Computer Simulation*. The Society for Computer Simulation International, 1998; 168–173.