The Specification of Source-To-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications

Daniel J. Quinlan[1], Markus Schordan[1],
Bobby Philip[1], and Markus Kowarschik[2]

[1] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, Livermore, CA, USA

[2] System Simulation Group, Department of Computer Science
University of Erlangen-Nuremberg, Germany

Abstract. The performance of object-oriented applications in scientific computing often suffers from the inefficient use of high-level abstractions provided by underlying libraries. Since these library abstractions are user-defined and not part of the programming language itself there is no compiler mechanism to respect their semantics and thus to perform appropriate optimizations.
In this paper we outline the design of ROSE and focus on the discussion of two approaches for specifying and processing complex source code transformations. These techniques are intended to be as easy and intuitive as possible for potential ROSE users; i.e., for designers of object-oriented scientific libraries, people most often with no compiler expertise.

1 Introduction

The future of scientific computing depends upon the development of more sophisticated application codes. The original use of Fortran represented higher-level abstractions than the assembly instructions that preceded it, but exhibited performance problems that took years to overcome. However, the abstractions represented in Fortran were *standardized* within the language; today's much higher-level object-oriented abstractions are more difficult to optimize because they are *user-defined.*

The introduction of parallelism greatly exacerbates the compile-time optimization problem. While serial languages serve well for parallel programming, they know only the semantics of the serial language. As a result a serial compiler cannot introduce scalable parallel optimizations. Significant potential for optimization of parallel applications is lost as a result.

There is a significant opportunity to capitalize upon the parallel semantics of the object-oriented framework and drive significant optimizations specific to both shared memory and distributed memory applications.

We present a preprocessor based mechanism, called *ROSE*, that optimizes parallel object-oriented scientific application codes that use high-level abstractions provided by object-oriented libraries. In contrast to compile-time optimization of basic language abstractions (loops, operators, etc.), the optimization of the *use* of library abstractions within applications has received far less attention. With ROSE, library developers define customized optimizations and build specialized preprocessors. Source-to-source transformations are then used to provide an efficient mechanism for introducing such custom optimizations into user applications. A significant advantage of our approach is that preprocessors can be built which are tailored to user-defined high-level abstractions, while vendor supplied C++ compilers know only the lower-level abstractions of the C++ language they support. So far, our research has focused on applications and libraries written in C++.

This approach permits us to leverage existing vendor C++ compilers for architecture specific back-end optimizations. Significant improvements in performance associated with source-to-source transformations have already been demonstrated in recent work, underscoring the need for further research in this direction.

Statement/GridSize	5x5	25x25	100x100
`w=1`	3.0	1.8	1.3
`w=u`	3.0	1.9	1.3
`w=u*2+v*3+u`	13.0	5.0	2.4
indirect addressing	44.0	41.0	32.5
`where` statements	23.0	5.0	3.0
9pt stencil	77.0	14.0	5.6

Table 1. Speedups associated with optimizing source-to-source transformations of abstractions within Overture applications. Results are presented for 2D array objects `u,v,w`.

Table 1 shows some of these improvements for the use of optimizing source-to-source transformations within the *OVERTURE* framework [4]. Speedups are listed for several common types of statements, the values are the ratios of execution times without and with the optimizing source-to-source transformations. In each case the optimizing transformation results in better performance. The degree of improvement depends upon the abstraction being optimized within the application code and the problem size. For example, in the case of indirect addressing the performance improvement for 100×100 size problems is 3250%, showing the rich potential for indirect addressing optimizations. We can expect that ROSE will duplicate these results through the fully automated introduction of such optimizing transformations into application codes.

Other work exists which is related to our own research. Internally within ROSE a substantially modified version of the *SAGE II* [7] AST restructuring tool is used. *Nestor* [9] 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 on *MPC++* [10, 11] has led to the development of a C++ tool similar to SAGE, but with some additional capabilities for optimization. However, it does not attempt to address the sophisticated scale of abstractions that we target or the transformations we are attempting to introduce.

Related work on *telescoping languages* [8] shares some of the same goals as our research work and we look forward to tracking its progress in the coming years. Other approaches we know of are based on the definition of library-specific *annotation languages* to guide optimizing source code transformations [12] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations [13].

Work at University of Tennessee has lead to the development of *Automatically Tuned Linear Algebra Software* (ATLAS) [5]. Within this approach numerous transformations are written 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. Our work is related to this in the sense that this is one possible mechanism for the identification of optimizing transformations that could be used within preprocessors built using ROSE to optimize application codes. Our approach to the specification of transformations in this paper is consistent with the source code generation techniques used to generate transformations within ATLAS.

The remainder of this paper is organized as follows. In section 2 we give a survey on the ROSE infrastructure; we describe the process of automatically generating library-specific preprocessors and explain their source-to-source transformation mechanisms. The main focus of this paper is on the specification of these source-to-source transformations by the developer of the library. We will thus discuss two alternative specification approaches and an AST query mechanism in section 3. In section 4 we finally summarize our work.

2 ROSE Overview

We have developed ROSE as a preprocessor mechanism because our focus is on optimizing the use of user-defined high-level abstractions and not on lower-level optimizations associated with back-end code generation for specific platforms. Our approach permits ROSE to work as a preprocessor independent of any specific C++ compiler.

In the following we will briefly describe the internal structure of a preprocessor which has been automatically generated using ROSE; particularly the recognition of high-level abstractions (section 2.1), the overall preprocessor design (section 2.2), and finally the specification of the transformations (section 3), which is the main focus of this paper.

2.1 Recognition of Abstractions

We recognize abstractions within a user's application much the same way a compiler recognizes the syntax of its base language. To recognize high-level abstractions we build a hierarchy of *high-level abstract grammars* and the corresponding *high-level ASTs* using ROSE. This hierarchy is what provides for a relationship to telescoping languages [8].

These high-level abstract grammars are very similar to the base language abstract grammar — in our case an abstract C++ grammar. They are modified forms of the base language abstract grammar with added terminals and non-terminals associated with the abstractions we want to recognize. They cannot be modified in any way to introduce new keywords or new syntax, so clearly there are some restrictions. However, we can still leverage the lower-level compiler infrastructure; the parser that builds the base language AST. New terminals and nonterminals added to the base language abstract grammar might represent specific user-defined functions, data-structures, user-defined types, etc. More detail about the recognition of high-level abstractions can be found in [3]

2.2 Preprocessor Design

Figure 1 shows how the individual ASTs are connected in a sequence of steps; automatically generated translators generate higher level ASTs from lower level ASTs. The following describes these steps:

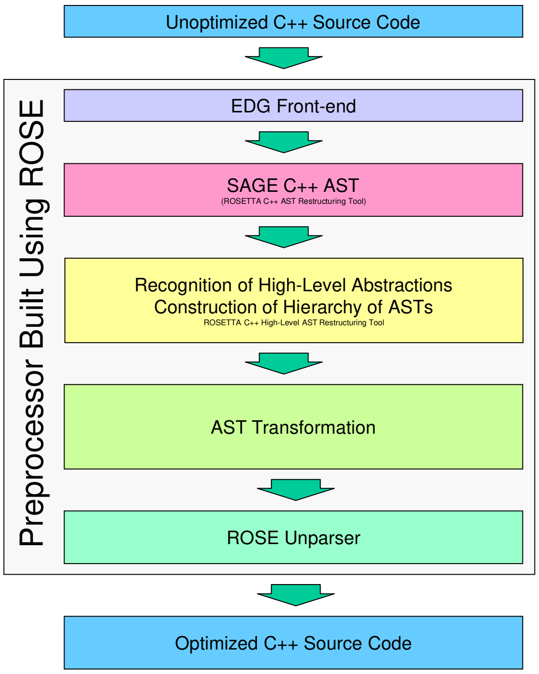

Fig. 1. Source-to-source C++ transformation with preprocessors using the ROSE infrastructure.

1. The first step generates the Edison Design Group (EDG) AST. This AST has a proprietary interface and is translated in the second step to form the abstract C++ grammar's AST.
2. The C++ AST restructuring tool is generated by ROSETTA [1] and is essentially comformant with the SAGE II implementation. This second step is representative of what SAGE II provides and presents the AST in a form where it can be modified with a non-proprietary public interface. At this second step the original EDG AST is deleted and afterwards is unavailable.
3. The third step is the most interesting since at this step the abstract C++ Grammar's AST is translated into higher level ASTs. Each parent AST (associated with a lower level abstract grammar) is translated into all of its child ASTs so that the hierarchy of abstract grammars is represented by a corresponding hierarchy of ASTs (one for each abstract grammar). Transformations can be applied at any stage of this third step and modify the parent AST recursively until the AST

associated with the original abstract C++ grammar is modified. At the end of this third step all transformations have been applied.
4. The fourth step is to traverse the C++ AST and generate optimized C++ source code (unparsing). This completes the source-to-source preprocessing.

An obvious next and final step is to compile the resulting optimized C++ source code using a vendor's C++ compiler.

3 Specification of Transformations

This paper is primarily about the specification of transformations for use within preprocessors built using ROSE. The purpose of any transformation is to locally rewrite a statement or collection of statements — the *target* — using the semantics of the high-level abstractions being optimized and the context of their use within the application.

All transformations share a common set of requirements. Internally, the application has been parsed to build the corresponding AST within the AST hierarchy, using either the abstract C++ grammar or a higher-level abstract grammar. This forms the starting point for the internal processing. The ending point is the AST which has been modified according to the specification of the transformation. Since at this point all fragments of the AST where transformations will be applied have been identified in the recognition phase, we can associate transformations with specific terminals of the high-level abstract grammar. This approach permits the transformations to be performed within a single traversal of the AST at each node corresponding to a specific terminal of the abstract grammar.

The definition of the interface for the specification of transformations is straightforward. Inputs are fragments of the application's AST representing C++ code to be optimized. Outputs are the new AST fragments representing the transformed code. The actual transformation phase is the substitution of the input AST fragment with the output AST fragment within the larger AST representing the application code.

It is the responsibility of the transformation to reproduce the semantics of the statement or collection of statements being substituted. Ultimately, it is the responsibility of the library developer to correctly specify the transformation which represents the semantics of the high-level abstraction being optimized.

Our recent research has been focusing on two fundamentally different methodologies for specifying the transformations to be applied; a first approach based on direct (manual) AST construction and a more sophisticated second approach leveraging the compiler front-end to generate the required output AST fragment. An orthogonal query mechanism allows either AST fragment construction mechanism to perform queries on the input AST fragment. This query mechanism permits the output AST fragment to be tailored to the context of the input AST fragment.

3.1 Mechanism for the Query of AST Fragments

```
list<char*> globalQueryCharStarListInitializerFunction ( void )
   {
  // This function returns a value used to initialize variables of the return type
     list<char*> returnList = 0;
     return returnList;
   }

list<char*> globalQueryGetListOperandCharStarFunction ( SgNode* astNode )
   {
  // This function returns a single element list of variable names at the astNode
     list<char*> variableNameList;

     SgVarRefExp* varRefExp = isSgVarRefExp(astNode);
     if (varRefExp != NULL)
        {
          SgVariableSymbol* variableSymbol = varRefExp->get_symbol();
          SgInitializedName* initializedName = variableSymbol->get_declaration();
          SgName variableName = initializedName->get_name();
          char* name = strdup(variableName.str());
          variableNameList.push_back (name);
        }
     return variableNameList;
   }

list<char*> globalQueryAssemblyCharStarListFunction
   ( list<char*> inputX, list<char*> inputY )
   {
  // This function adds one list to the other and returns the result
     inputX.merge(inputY);
     return inputX;
   }
```

Fig. 2. Example of functions used in the templated query interface for a query of variable names in AST fragments (e.g., expression statements) using synthesized attributes. Function pointers are used as inputs to the templated `Query` class. The templated STL `list<>` class forms an argument to the templated `Query` class.

Figures 2 and 3 show an example of the query specification mechanism using synthesized attributes. This mechanism permits the use of

```
// Build a query operator (using STL and primative types as template arguments)
   Query< int, list<char*>, int >
        localQueryOperator ( globalQueryCharStarListInitializerFunction,
                             globalQueryGetListOperandCharStarFunction,
                             globalQueryAssemblyCharStarListFunction );

// now ask the question
   list<char*> operandNameList = localQueryOperator.traverse( astNode );
```

Fig. 3. Example source code fragment specifying the query of variable names (e.g., in expression statements) using synthesized attributes.

inherited and synthesized attributes and accumulators in the development of queries upon any fragment of the AST. The mechanism is backed up by an automatically generated tree traversal mechanism generated by ROSETTA as part of the AST restructuring tool associated with each level of an abstract grammar in the hierarchy.

3.2 Direct Construction of AST Fragments

From the perspective of the compiler, at the start of the optimization phase the user's application is already parsed and represented by an AST. Any optimization must modify this representation. Evidently, the simplest approach is to modify the AST directly. Numerous specialized tools are based around techniques that directly manipulate the internal forms used within compilers. The AST and the source code are semantically equivalent in the sense that they represent the same code. However, the AST is more complex for users to manipulate as a tree, at least partly because programmers are used to manipulating source code as text.

Figure 4 shows an example of code required to construct a `for` loop within Sage++ [7] (predecessor to Sage II and our *modified* version of Sage II). Debugging the code generated from this AST fragment, requires a level of indirection which makes the specification of larger transformations particularly difficult.

figure 5 shows the code generated from the specification of the AST fragment in figure 4. Within this approach, and specifically in this example, there is a dramatic difference in the amount of code required to specify the AST fragment (figure 4) and the source code unparsed from the AST fragment (figure 5). Specific to this example there is a factor of 12 expansion in complexity as measured in the number of lines of code. It is also immediately obvious that the final code representation (figure 5) is easier to understand. The source code building the AST fragment (figure 4) additionally assumes a working knowledge of a particular AST restructuring tool (in this case Sage++).

However, conventional methods for the specification of transformations — which we have found in the literature — are characterized by the direct construction or alteration of AST fragments (e.g., declaration statement objects, `for` loop statement objects, etc.). Alternative compiler tools (Nestor [9], Sage [7], etc.) are similarly limited to such direct transformation approaches and, as a result, are most appropriate for simple transformations. These direct approaches also assume a high degree of compiler expertise which additionally limit their applicability within scientific computing.

3.3 Source-String Based Construction of AST Fragments

Since scientific library writers represent our target audience, we cannot assume any compiler expertise or familiarity with ASTs. Additionally, it is our experience that transformations for cache-based optimizations, which we are particularly interested in, are complex [14, 15]. Implementing these kinds of transformations using the approach of direct AST construction is rather tedious, if not impractical. We therefore require a more compact representation of the transformation. Clearly, from the user's perspective, the transformation would be best represented as source code in the application's programming language, even if this representation cannot immediately be substituted into the AST.

Our more sophisticated second approach is therefore based on the source code representation of the transformations and leveraging the compiler front-end in order to generate the equivalent AST fragment to be substituted into the application's AST. There are several advantages of this transformation mechanism:

- The source code represents the most compact representation of the equivalent AST and is familiar to the programmer.
- The source code representing the transformation can be most easily examined for correctness by the user.
- Since the source code can be extracted from files, transformations can be built from working versions of the code representing the transformations. This approach thus allows test codes representing the transformations to be built separately and introduced as optimizing transformations into applications. We expect this approach will permit an interface to optimization tools such as ATLAS.

– The transformation source code can be parsed directly by the internal compiler infrastructure to generate the AST fragment required. Thus the process of generating the AST fragment for insertion into the AST at compile-time can be automated.

With sufficient exercise of the query mechanism the source-string can be tailored (programmed) to build most source code transformations. Figure 6 shows the source code and function call required to generate the identical AST fragment as in figure 4.

We consider the manipulation of strings, as an alternative way to specify the AST transformation at compile time, to be an added approach especially useful for larger transformations. This approach is direct from the user's point of view, since the source-to-source transformation is specified using source code. But our approach should be considered indirect from the compiler's point of view, since the AST fragment is subsequently generated from source-strings and *it* (the AST fragment) is what is needed at compile-time.

The optimization of object-oriented array class libraries can form an interesting example problem. The array statements elegantly represent mathematical expressions because of the operator overloading made possible within the C++ language. We consider "`A(I)= (B(I-1)+B(I+1))*0.5;`" as a sample array statement from the A++/P++ array class library [16, 17]. This library permits the specification of serial and parallel array objects and their manipulation using overloaded operators. The library permits the evaluation of expressions using pair-wise operator or expression template mechanisms. Both of these approaches have performance problems. The pair-wise evaluation of expressions within a statement is not cache friendly and results in a loss of performance (factor of 1-6) [17, 14]. While the expression templates have long compile times and limits on their application [14].

Figure 8 shows the semantically equivalent transformation generated from the above A++/P++ target (figure 7). In this case the optimizing transformation removes all array class overhead and provides the same performance as C or Fortran 77, since the data is accessed through `restrict` pointers. More sophisticated transformations could provide fusion between statements to provide improved temporal locality of array statement expressions (providing larger internal loops).

4 Conclusions

ROSE is a library to simplify the construction of optimizing preprocessors. The specification of the transformation is done within the program that is compiled to be the preprocessor. This program leverages both the ROSE library for internal infrastructure and the source code generated by ROSETTA (part of ROSE). Source code generated by ROSETTA implements AST restructuring tools corresponding to abstract grammars and higher-level abstractions, this source code is compiled to build the preprocessor. Infrastructure within ROSE permits the specification of transformations, either directly modifying the AST or indirectly through the specification of source-strings which are processed to form AST fragments which are used to modify the AST.

We have presented the ROSE infrastructure to automatically generate library-specific source-to-source compilers (preprocessors). These preprocessors can be used to optimize the use of high-level abstractions in parallel object-oriented applications.

We have presented two basic approaches for specifying transformations. While our first approach of direct AST construction turned out to be tedious (especially for complex cache-based transformations), our second approach, which leverages the compiler front-end instead, provides an elegant and comfortable alternative.

References

1. Quinlan, D., Philip, B., "ROSETTA: The Compile-Time Recognition Of Object-Oriented Library Abstractions And Their Use Within Applications", Proceedings of the PDPTA'2001 Conference, Las Vegas, Nevada, June 24-27 2001
2. Quinlan, D., "ROSE: Compiler Support for Object-Oriented Frameworks", Parallel Processing Letters, Vol. 10, also Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France, January 2000.
3. Quinlan, D. Schordan, M. Philip, B. Kowarschik, M. "Parallel Object-Oriented Framework Optimization", (submitted to) Special Issue of Concurrency: Practice and Experience, also in Proceedings of Conference on Parallel Compilers (CPC2001), Edinburgh, Scotland, June 2001.
4. Brown, D., Henshaw, W., Quinlan, D., "OVERTURE: A Framework for Complex Geometries", Proceedings of the ISCOPE'99 Conference, San Francisco, CA, Dec 7-10 1999.
5. ATLAS homepage, `http://www.netlib.org/atlas`.
6. Edison Design Group, `http://www.edg.com`.
7. 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.

8. 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.
9. Silber, G.-A., `http://www.ens-lyon.fr/~gsilber/nestor`.
10. Ishikawa, Y., et. al., "Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —", Proceedings of Reflection'96 Conference, April 1996, more info available at: `http://pdswww.rwcp.or.jp/mpc++/mpc++.html`.
11. Chiba, S., "Macro Processing in Object-Oriented Languages", Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98), Australia, November, IEEE Press, 1998, more info available at: `http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html`.
12. Guyer, S.Z., Lin, C., "An Annotation Language for Optimizing Software Libraries", Proceedings of the Second Conference on Domain-Specific Languages, October 1999.
13. Menon, V., Pingali, K., "High-Level Semantic Optimization of Numerical Codes", Proceedings of the ACM/IEEE Supercomputing 1999 Conference (SC99), Portland, OR, 1999.
14. Bassetti, F., Davis, K., Quinlan, D., "Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures" Proceedings of the ISCOPE'98 Conference, Santa Fe, NM, 1998.
15. Weiß, C., Karl, W., Kowarschik, M., Rüde, U., "Memory Characteristics of Iterative Methods", Proceedings of the ACM/IEEE Supercomputing 1999 Conference (SC99), Portland, OR, 1999.
16. Lemke, M., Quinlan, D., "P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications", published as part of CONPAR/VAPP V, September 1992, Lyon, France; also published in Lecture Notes in Computer Science, Springer Verlag, September 1992.
17. Parsons, R., Quinlan, D., "A++/P++ Array Classes for Architecture Independent Finite Difference Computations", Proceedings of the Second Annual Object-Oriented Numerics Conference, pages 408-418, Sunriver, OR, April 1994.

```
SgExpression *Expression = CExpressionStatement->expr()->lhs();
SgSymbol *Argument = (Expression->lhs()->symbol() == NULL) ?
                      Expression->lhs()->lhs()->symbol() : Expression->lhs()->symbol();
SgExpression dimen_call(RECORD_REF);
dimen_call.setLhs(SgVarRefExp(*TemporaryArrayPtr));
SgSymbol *FieldSymbol = FindFieldWName("redim", TemporaryArrayPtr );
SgFunctionCallExp dimen_func (*FieldSymbol);
dimen_func.addArg(SgVarRefExp(*Argument));
dimen_call.setRhs(dimen_func);
SgCExpStmt RedimMemberFunction (dimen_call);

SgExpression *Expression = getRootExpression ( Statement );
SgVariableSymb *TemporaryArrayPtr = new SgVariableSymb ("xxx_dA_T");
TemporaryArrayPtr->declareTheSymbol( *( StatementPtr->controlParent() ) );
SgExpression *le = Expression->lhs();
SgDerivedType *dtp = NULL;
SgSymbol *vsb = le->symbol();
TemporaryArrayPtr->setType(vsb->type());

SgExpression *Expression = getRootExpression ( Statement );
SgVariableSymb *TemporaryArrayPtr = new SgVariableSymb ("xxx_dB");
TemporaryArrayPtr->declareTheSymbol( *( StatementPtr->controlParent() ) );
SgExpression *le = Expression->lhs();
SgDerivedType *dtp = NULL;
SgSymbol *vsb = le->symbol();
TemporaryArrayPtr->setType(vsb->type());

SgVariableSymb *LoopInductionVariable = new SgVariableSymb ("i_loopxx");
LoopInductionVariable->setType( SgTypeInt() );
LoopInductionVariable->declareTheSymbol( *( StatementPtr->controlParent() ) );
SgCExpStmt *AssignmentExpression =
     new SgCExpStmt ( SgAssignOp( *LhsExpression , SgVarRefExp(*TemporaryArrayPtr) ) );
SgBasicBlock* LoopBody = new SgBasicBlock ();
LoopBody.insert(AssignmentExpression);
int upperBound = 100;
SgForStmt *ForStatementPtr =
     new SgForStmt ( SgAssignOp(SgVarRefExp(*LoopInductionVariable),SgValueExp(0)),
                    SgVarRefExp(*LoopInductionVariable) < SgValueExp(upperBound),
                    SgUnaryExp(PLUSPLUS_OP,1,SgVarRefExp(*LoopInductionVariable)),
                    SgCExpStmt(*LoopBody));
```

Fig. 4. Code required to build an AST fragment for the `for` loop shown in figure 5.

```
A.redim(size);
for (i_loopxx = 0; i_loopxx < 100; i_loopxx++)
   {
     xxx_dA_T[i_loopxx] = xxx_dB[i_loopxx];
   }
```

Fig. 5. Unparsed source code from the AST formed in figure 4.

```
buildAST_Fragment (
     "A.redim(size); \n for (i_loopxx = 0; i_loopxx < 100; i_loopxx++) \n \
      { \n xxx_dA_T[i_loopxx] = xxx_dB[i_loopxx]; }");
```

Fig. 6. Function call using a source-string to create an AST representing the source code in figure 5.

```
// A and B are declared as array objects (not shown)
// and used in an array statement
   A(I) = ( B(I-1) + B(I+1) ) * 0.5;
```

Fig. 7. Target of optimizing transformation (transformation shown in figure 8).

```
// Transformation Target: A(I) = ( B(I-1) + B(I+1) ) * 0.5;
   int rose_index [8];
   int rose_stride[8];
   int rose_base  [8];
   int rose_bound [8];
   double restrict* B_rose_pointer = B.getDataPointer();
   double restrict* A_rose_pointer = A.getDataPointer();
   rose_base[0]   = (B.getBase)(0);
   rose_bound[0]  = (B.getBound)(0);
   rose_stride[0] = (B.getStride)(0);
   for (int i = rose_base[0]; i <= rose_bound[0]; i += rose_stride[0])
      {
        A_rose_pointer[i] = (B_rose_pointer[i-1] + B_rose_pointer[i+1] ) * 0.5;
      }
```

Fig. 8. Unparsed source code represented by an AST of the transformed target code (figure 7). The specification uses the internal ROSE infrastructure (not shown). A source-string is processed to generate an AST fragment and then unparsed to form the text.