The ROSE Source-to-Source Compiler Infrastructure

Dan Quinlan Chunhua Liao
Lawrence Livermore National Laboratory
{dquinlan, liao6}@llnl.gov

Abstract

The development of future software for proposed new computer architectures is expected to make significant demands on compiler technologies. Significant rewriting of applications will be likely to support the use of new hardware features and preserve the current software investment. Existing compilers provide little support for the broad range of research efforts addressing exascale challenges, such as parallelism, locality, resiliency, power efficiency, etc. The economics of how new machines will change an existing code base of software, that is too expensive to manually rewrite, may well drive automated mechanisms to transform existing software to take advantage of future machine features. This approach will lessen the cost and delay of the move to new, and possibly radically different, future architectures.

Source-to-source compilers provides a pragmatic vehicle to support research, development, and deployment of novel compiler technologies by compiler experts or even advanced application developers. Within a source-to-source approach the input source code is read by the compiler, an internal representation (IR) is constructed, the IR is the basis of analysis that is used to guide transformations, the transformations occur on the IR, the IR is used to regenerate new source code, which is then compiled by a backend compiler. Our source-to-source compiler, ROSE, is a project to support the requirements of DOE. Work on ROSE has focused on the development of a community based project to define source-to-source compilation for a broad range of languages especially targeted at DOE applications (addressing robustness and large scale codes as required for DOE applications).

Novel research areas are most easily supported when they can leverage significant tool chains that interact and use source code while allowing the hardware vendor's own compiler for low level optimizations. In fact, high level optimization are rarely feasible for existing low level compilers for common languages such as C, C++, and Fortran. ROSE addresses the economics of how compiler research can be moved closer to the audience with significant technical performance problems and for whom the hardware is likely to be changing significantly in the next decade. Within ROSE it is less the goal to solve all problems than to permit domain experts to better solve their own problems. This talk will focus on the design and motivation for ROSE as an open community source-to-source compiler infrastructure to support performance optimization, tools for analysis, verification and software assurance, and general custom analysis and transformations needs directly on software using the languages common within DOE High Performance Computing.

1. Introduction

ROSE is designed to support software analysis and optimization for both source code and binary software. As a source-to-source compiler infrastructure it supports the automated analysis and transformation of large scale applications as part of ongoing research to support compiler research, future computer architectures, and software analysis and verification. ROSE supports a broad range of languages: C, C++, Fortran (Fortran 4 - Fortran 2003), Python, UPC, OpenMP, PHP, and Java. ROSE also supports the analysis of both Windows and Linux binary executables, and multiple instruction set families: x86, Power PC, and ARM. ROSE include most common forms of program analysis and many prepackaged forms of transformations. The overview of the design of ROSE is presented in figure 1, which shows how users can interact with the ROSE IR and predefined program analysis and transformations to define new tools to operate on both source code and binary executables.

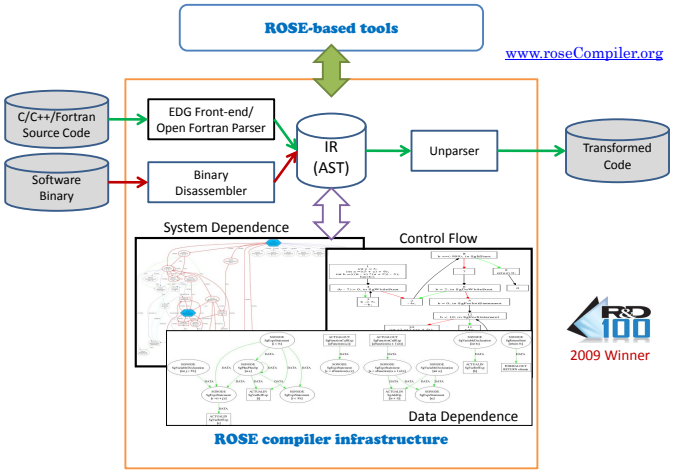

Figure 1. ROSE compiler infrastructure

2. Background

ROSE is a source-to-source compiler, developed to support the automated rewriting of large scale DOE application to support new computer architectures and especially novel future architectures. We refer to the IR in ROSE as SageIII, to acknowledge the Sage++ and SageII projects by Dennis Gannon and Karl Kesselman in the early 90's. SageIII is sometimes referred to as the ROSE IR; our IR is automatically generated using tools that we wrote and distribute as part of ROSE. It shares a design with the Sage IR as an object

oriented IR that closely follows the specification of the grammar for the different languages that are supported. Modifications to the IR are typically done through modifications of the inputs to ROSETTA; the tool we wrote to generate the IR. A significant emphasis in ROSE has been on automation and code generation.

ROSE supports the generation of the source code for the IR so that it can support a large number of IR nodes required for multilanguage source to source and binary analysis across modern instruction sets. Automated code generation of the IR source code permits embedding sophisticated features into the design of the IR: attribute grammar based traversals, memory handling for IR node pools, internal analysis support, AST copy support, AST merge, AST serialization, etc.

A property of the design of the IR is that it saves all possible information about the input software (source code or binary). An explicit goal is to unify the handling of source code and binary analysis and support software analysis generally. Most work goes into the representation of the software within ROSE, the output of software to generate code is generally straight forward and requires little or no analysis. A significant focus is on supporting the transformation of the internal Abstract Syntax Tree (AST). Unlike most compilers which target a sophisticated compiler audience, ROSE defines a broader audience of tool developers dominantly. Though clearly the more sophisticated tools and ambitions can require significant compiler background.

A large number of IR nodes are supported within ROSE, about half to support source code and the other half for binary analysis. ROSE supports a wide range of languages and contains relatively small numbers of IR nodes that are language specific.

A few general details about ROSE:

- Support for six serial languages (source code) and three instruction set families (binary analysis)
- Support for two parallel languages (UPC and OpenMP)
- IR support for Dwarf debug format
- Supports over three thousand types of machine instructions (x86, Power PC, and ARM)
- 700 IR nodes in ROSE: 350 IR nodes for source code and about 350 for binary analysis (significant space savings from rich type system of the IR).
- ROSE is about 1.5 million lines of code
  - 700K of automatically generated code
  - 300K of EDG code (distributed as a binary)
  - 500K of hand written code (not counting ROSETTA and inputs to ROSETTA)
- ROSE is a DOE funded project for the last 8 years
- Every commit from a user branch to the main trunk is tested for about 50 machine hours (run in parallel, this occurs more quickly).
- Release tests take another 20 machine hours (run in parallel this occurs more quickly).
- Personnel over the history of the project include:
  - three research staff
  - two professional programmers
  - 5 post-docs
  - 70 students have been a part of of the project over the last 8 years.
- Hundreds of pages of on-line documentation
- Multiple mailing lists: public, developers, core staff
- Basis for numerous compiler and tools projects worldwide
- About 80% of source code IR nodes are shared between all languages
- About 10% of IR nodes are devoted to specialized corner cases that are relatively rare in the AST. Common IR nodes in the AST are used by most of the supported languages.
- The AST represents more structure of the software than semantics
- Two external software repositories (GIT and SVN)
- Approximately 6000 downloads to date
- Released publicly on the web for 3 years
- BSD license
- 2009 R&D Award
- Largest AST held in memory for analysis: 10 million lines of code (over a hundred million IR nodes, serialized to a single 15 Gig file) AST traversal time was about 20 seconds
- Source code position information for every aspect of code (unoptimized this consumes 25% of the space for the AST)

3. Analysis

We have developed a set of analyzes on top of the ROSE AST. They are designed to be utilized by users via simple function calls to interfaces. The program analysis available include control flow analysis, data flow analysis (live variables, def-use chain, reaching definition, side effect, alias analysis, etc.), call graph analysis, class hierarchy analysis, data dependence and system dependence analysis, and MPI communication pattern analysis.

In particular two control flow graphs are provided in ROSE: virtual control flow graph and static control flow graph. The virtual control flow graph (VCFG) is generated on the fly to accommodate frequent AST transformations. No explicit graph is ever created: only the particular CFG nodes and edges used in a given program ever exist. CFG nodes and edges are value classes (they are copied around by value, reducing the need for explicit memory management). The static control flow graph is generated from VCFG and stored explicitly. It is designed to avoid overhead of dynamically generation of control flow for AST with no or rare changes. In both graphs, control flow is expressed at statement and expression levels. They reflect short-circuited logical and conditional operators properly. It assumes operands of expressions are computed in left-to-right order, unlike the actual language semantics, however.

Our compiler research also includes forms of analysis not common to performance optimization and stretching to formal methods; such as SMT solvers, and symbolic execution, and abstract interpretation. These forms of analysis have a rich opportunity to be used to support proof based techniques to establish properties that may prove important to verification of software for correct use of novel hardware features in the future. This work expands the range of analysis avaiable in ROSE and explores its usefulness to supporting HPC on evolving machine architectures. Such work also opens the opportunity to support automated low level verification of software against complex future constraints for power optimization, correct use of complex programming models, and transformations to support increased resiliancy for future HPC.

4. Transformation and Optimization

ROSE provides different levels of interfaces to support building code transformation via manipulating AST. High level interfaces are recommended to use whenever possible for their simplicity.

Low level interfaces can give users the maximum freedom to manipulate some details in AST trees. We also create a wide range of transformation and optimization functions to demonstrate the use of the APIs. Users can further reuse these existence functions to build more complex or customized transformations and optimizations.

The low level AST interface exposes member functions of different types of objects in AST, including the tree nodes, symbol tables, attributes, preprocessing information, file location information, and so on. This level of interface provides users the maximum freedom to change the AST. But using such a low level interface is a tedious and error-prone process since users have to manually create the edges between nodes and maintain the consistence between the tree and the corresponding symbol tables. An AST consistency test function is provided to check the transformed AST is complete and consistent.

The high level AST interface contains a set of functions to build AST subtrees and insert them into existing AST. A lot of helper functions are also provided to walk the tree, query nodes, replace, delete, or even copy subtrees. Using the high level interface, users only need to provide essential source code information as parameters to these functions. All details of edges and symbol tables are automatically and transparently handled by the interface. The high level interface also support different order of AST constructions. Users can either build parent nodes first or build children nodes first. Or statements using some variables can be even build before the declaration statements are created. The high level interface is able to automatically patch up AST and symbol tables once enough information is available.

We have developed a wide range of transformations and optimizations in ROSE, including partial redundancy elimination, constant folding, inlining, outlining (separating out a portion of code as a function), OpenMP 3.0 implementation [2], automatic parallelization [3] and loop transformations, such as fusion, fission, interchange, unrolling, and blocking. For example, the ROSE outliner [1] can extract code portions of C, C++ and Fortran into functions. It supports the classic way of generating one function parameter for each variable to be passed in or out of the generated function. It also can optionally wrap multiple variables into an array and use the array as a single function parameter. More importantly, the ROSE outliner differentiates the usage of parameters into use-by-value and use-by-address. So the classic outlining algorithm's excessive pointer dereferences (used to access written parameters) can be largely eliminated if the accesses to the written parameters are substituted by their value clones. Other analyses, such as liveness analysis and scope analysis are also used to further improve the quality of the outlining so the generated function preserves the original semantics and performance characteristics.

5. Enabling Exascale Research

ROSE is being used by several DOE projects to address the challenges of exascale computing. One of the projects, Thrify, is using ROSE to explore novel compiler analysis and optimization to take advantage of an exascale architecture with many-core chips specially designed to improve performance per watt. In another project, CODEX, an alternative simulator is used to support the analysis of kernels from DOE applications using other proposed exascale node architectures and possible future network connection topologies and hardware. In still another project, ROSE has been used to analyze example applications and extract the MPI usage as an input code to support evaluation of message passing to support the network communication simulation. This work is part of a project to automate how co-design can be applied to support the evaluation of DOE applications.

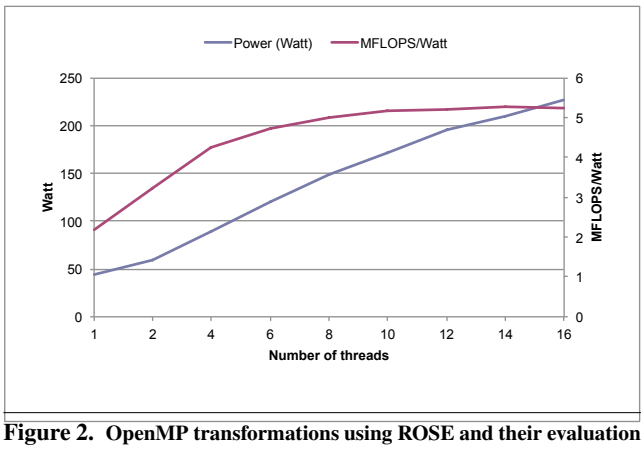

Figure 2. OpenMP transformations using ROSE and their evaluation using a simulator on a numerical kernel (Jacobi Iteration) for both power and MFLOP/Watt.

Figure 2 shows the evaluation of both power and MFLOP per Watt using a node simulator and for different numbers of threads. All transformations were done using ROSE. The simulator represents a proposed future node architecture.

6. Results

Numerous tools are available in ROSE, such as

1. Auto parallelization tools for HPC,
2. OpenMP compiler,
3. Exascale Co-design skeleton generator,
4. ROSE-CIRM (Runtime error detection support),
5. Tools for the analysis of MPI,
6. Tools to enforce data integrity for Exascale architectures.
7. Binary emulation to support general analysis,
8. Compass static analysis tool,
9. Data structure visualization (to support debugging) and
10. Program Visualization.

Separate from the tools provided in ROSE, and the tools that external groups have built using ROSE, ROSE is itself a tool for building software analysis and optimization tools. Is is used as a basis for both graduate courses in a wide range of topics from compiler construction and analysis to software assurance. ROSE is released under a BSD license to support as low a barrier to adoption as possible for industrial use.

References

[1] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *The 22th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Newark, Delaware, USA, 2009.

[2] Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias S. Müller, Barbara M. Chapman, and Bronis R. de Supinski, editors, *IWOMP*, volume 6132 of *LNCS*, pages 15–28. Springer, 2010. ISBN 978-3-642-13216-2. doi: 10.1007/978-3-642-13217-9_2.

[3] Chunhua Liao, Daniel J. Quinlan, Jeremiah Willcock, and Thomas Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010.