A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries*

Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{liao6,dquinlan,panas2,desupinski1}@llnl.gov

Abstract. OpenMP is a popular and evolving programming model for shared-memory platforms. It relies on compilers to target modern hardware architectures for optimal performance. A variety of extensible and robust research compilers are key to OpenMP's sustainable success in the future. In this paper, we present our efforts to build an OpenMP 3.0 research compiler for C, C++, and Fortran using the ROSE source-to-source compiler framework. Our goal is to support OpenMP research for ourselves and others. We have extended ROSE's internal representation to handle all OpenMP 3.0 constructs, thus facilitating experimenting with them. Since OpenMP research is often complicated by the tight coupling of the compiler translation and the runtime system, we present a set of rules to define a common OpenMP runtime library (XOMP) on top of multiple runtime libraries. These rules additionally define how to build a set of translations targeting XOMP. Our work demonstrates how to reuse OpenMP translations across different runtime libraries. This work simplifies OpenMP research by decoupling the problematic dependence between the compiler translations and the runtime libraries. We present an evaluation of our work by demonstrating an analysis tool for OpenMP correctness. We also show how XOMP can be defined using both GOMP and Omni. Our comparative performance results against other OpenMP compilers demonstrate that our flexible runtime support does not incur additional overhead.

1 Introduction

OpenMP [1] is a popular parallel programming model for shared memory platforms. By providing a set of compiler directives, user level runtime routines and environment variables, it allows programmers to express parallelization opportunities and strategies on top of existing programming languages like C/C++ and Fortran. As a proliferation of new hardware architectures becomes available, OpenMP has become a rapidly evolving programming model; numerous improvements are being proposed to broaden the range of hardware architectures that it

* This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 15–28, 2010.

can accommodate. A variety of robust and extensible compiler implementations are the key to OpenMP's sustainable success in the future since an OpenMP compiler should deliver portable performance. Open source OpenMP compilers permit active research for this rapidly evolving programming model.

Developed at Lawrence Livermore National Laboratory, the ROSE compiler [2] is an open source compiler infrastructure to build source-to-source program translation and analysis tools for large-scale C/C++ and Fortran applications. Given its stable support for multiple languages and user-friendly interface to build arbitrary translations, ROSE is particularly well suited to build reference implementations for parallel programming languages and extensions. It also enables average users to create customized analysis and transformation tools for parallel applications. In this paper, we present our efforts to build an OpenMP research compiler using ROSE. Our goal is to support OpenMP research for ourselves and others. For example, we have extended ROSE's internal representation to represent the latest OpenMP 3.0 constructs faithfully and to facilitate their manipulation; allowing the construction of custom OpenMP analysis tools.

More generally, the tight coupling of the compiler translations and the runtime system upon which they depend often complicate OpenMP research. Changing the existing compiler translations to utilize a new runtime library (RTL) usually requires significant effort. Conversely, changing the RTL when new features require support from compiler translations can be difficult. This tight coupling impedes research work on the OpenMP programing model. We seek to use ROSE as a testbed to decouple compiler translations from the OpenMP runtime libraries. We have designed and developed a common RTL interface and a set of corresponding compiler translations within ROSE. As a preliminary evaluation, we demonstrate an OpenMP analysis tool built using ROSE and the initial performance results of ROSE's OpenMP implementation targeting the OpenMP RTLs of both GCC 4.4 and Omni [3] 1.6.

The remainder of this paper is organized as follows. In the next section, we introduce the design goal of ROSE and its major features as a source-to-source compiler framework. Section 3 describes the OpenMP support within ROSE, including internal representation, a common RTL, and translation support. Section 4 presents a preliminary evaluation of ROSE's OpenMP support. We discuss related work in Section 5 while we present our conclusions and discuss future work in Section 6.

2 The ROSE Compiler

ROSE [4,2] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C/C++ and Fortran applications. It also has increasing support for parallel applications using OpenMP, UPC and MPI. Similar to other source-to-source compilers, ROSE consists of frontends, a midend, and a backend, along with a set of analyses and

optimizations. Essentially, it provides an object-oriented intermediate representation (IR) with a set of analysis and transformation interfaces allowing users to build translators, analyzers, optimizers, and specialized tools quickly. The intended users of ROSE are experienced compiler researchers as well as library and tool developers who may have minimal compiler experience.

A representative translator built using ROSE works as follows (shown in Fig. 1). ROSE uses the EDG [5] front-end to parse C (also UPC) and C++ applications. Language support for Fortran 2003 (and earlier versions) is based on the open source Open Fortran Parser (OFP) [6]. ROSE converts the intermediate representations (IRs) produced by the front-ends into an intuitive, object-oriented abstract syntax tree (AST). The AST exposes interface functions to support transformations, optimizations, and analyses via simple function calls. Our object oriented AST includes analysis support for call graphs, control flow, data flow (e.g., live variables, def-use chain, reaching definition, and alias analysis), class hierarchies, data dependence and system dependence. Representative program optimization and translation interfaces cover partial redundancy elimination, constant folding, inlining, outlining [7], and loop transformations [8]. The ROSE AST also allows user-defined data to be attached to any node through a mechanism called persistent attributes as a way to extend the IR to store additional information. The ROSE backend generates source code in the original source language from the transformed AST, with all original comments and C preprocessor control structures preserved. Finally, ROSE can call a vendor compiler to continue the compilation of the generated (transformed) source code; generating a final executable. ROSE is released under a BSD-style license and is portable to Linux and Mac OS X on IA-32 and x86-64 platforms.

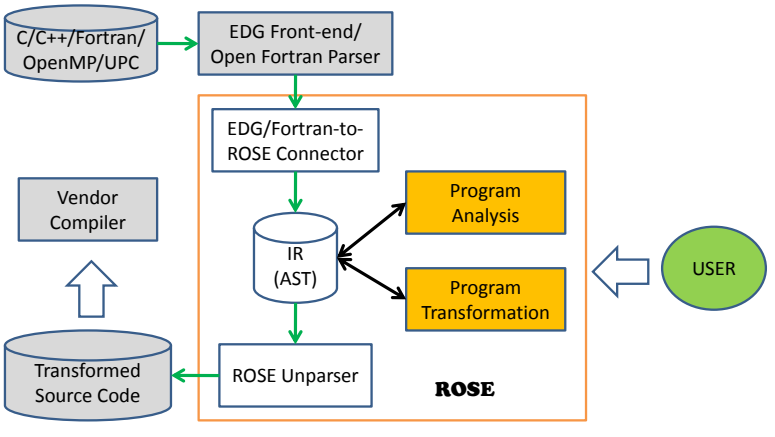

Fig. 1. A source-to-source translator built using ROSE

3 OpenMP Support in ROSE

As Fig. 2 shows, ROSE supports parsing OpenMP 3.0 constructs for C/C++ and Fortran[1], creating their internal representation as part of the AST, and regenerating source code from the AST. Additional support includes a set of translations targeting multiple OpenMP 2.5/3.0 RTLs based on XOMP, our common OpenMP RTL that abstracts the details of any specific RTL (such as GCC's OpenMP RTL GOMP [9] and the Omni [3] compiler's RTL). An automatic parallelization module is also available in ROSE [10].

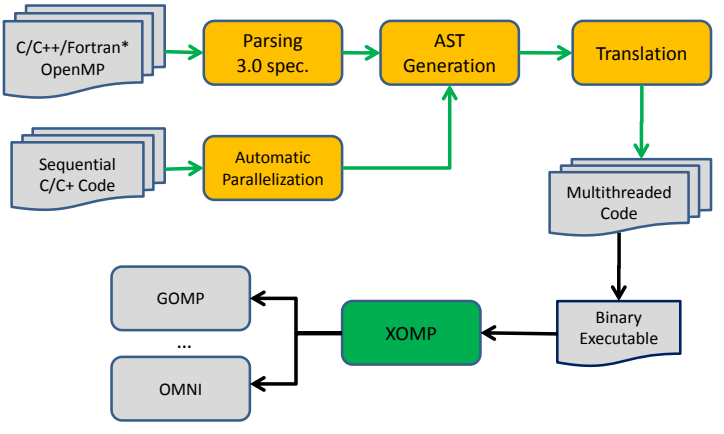

Fig. 2. OpenMP support in ROSE

3.1 Parsing and Representing OpenMP

Neither EDG (version 4.0 or earlier) nor OFP recognize OpenMP constructs. The raw directive strings exist in the ROSE AST as pragma strings for C/C++ and source comments for Fortran. Thus, we had to develop two OpenMP 3.0 directive parsers within ROSE, one for C/C++ and the other for Fortran. This, however, has significant advantages for users since they can easily change our parsers to test new OpenMP extensions without dealing with EDG or OFP.

ROSE's OpenMP parsers process OpenMP directive strings and generate a set of data structures representing OpenMP constructs. These data structures are attached to relevant AST nodes as persistent AST attributes. Using persistent AST attributes as the output of the parsers simplifies the work for parsing since we only make minimal changes to the existing ROSE AST. In fact, this light-weight representation for OpenMP is also used as the output of ROSE's automatic parallelization module [10]. As a result, the remaining OpenMP-related processing can work on the same input generated either from user-defined OpenMP programs or automatically generated OpenMP codes.

[1] Translation of Fortran OpenMP applications is still ongoing work.

After that, a conversion phase converts the ROSE AST with persistent attributes for OpenMP into an AST with OpenMP-specific AST nodes, which include statement-style nodes for OpenMP directives and supporting nodes (with file location information) for OpenMP clauses. Compared to the auxiliary persistent attributes attached to AST nodes, the newly-introduced AST nodes for OpenMP directives and clauses are inherently part of the ROSE AST. Thus, we can directly resuse most existing AST traversal, query, scope comparison, and other manipulation interfaces developed within ROSE to manipulate OpenMP nodes. For instance, a regular AST traversal is able to access all variables used within the AST node for an OpenMP clause with a variable list. This significantly simplifies the analysis and translation of OpenMP programs.

3.2 OpenMP Translation and Runtime Support

An OpenMP implementation must translate OpenMP applications into multithreaded code with calls to a supporting runtime library. To offer maximal freedom and optimization opportunities to OpenMP implementations, the OpenMP specification does not mandate the interface between a compiler and a runtime library. The implementation must decide what work to defer to the runtime library and how the compiler translation interacts with the library. Therefore, an OpenMP compiler's translation is traditionally tightly coupled with a given runtime library's interface. It is often a major effort to change the existing compiler translation to utilize a new runtime library. However, different runtime library choices and changes in the interactions between the compiler can significantly impact OpenMP performance. Thus, it would be especially desirable for an OpenMP research compiler to support multiple OpenMP runtime libraries.

Fortunately, although OpenMP runtime library interfaces vary, they usually include many similar or overlapped runtime library functions. For example, most portable OpenMP runtime libraries rely on the Pthreads API to create and to manipulate threads. Such a library usually provides a function that accepts a function pointer and a parameter to start multiple threads. Similalry, the OpenMP specification prescriptively defines some aspects of loop scheduling policies so runtime support for them often significantly overlaps.

We have introduced a common OpenMP RTL, XOMP, so that ROSE requires minimal changes to support multiple OpenMP RTLs. Depending on the similarity among RTLs, we use three rules in order to define XOMP and the corresponding compiler transformations.

- **Rule 1.** Target RTLs have some functions with similar functionalities. Those functions often differ by names and/or parameter lists. For each of the functions, we define a common function name and a union set of parameters in XOMP. The implementation of the common function will handle possible type conversion, parameter dispatch, inclusions/exclusions of functionality (to compensate for minor differences) before calling different the target RTL internally. By doing this, we can use one translation targeting XOMP's functions across multiple RTLs.

- **Rule 2.** Compared to other RTLs, a target RTL, libA, has an extra function, funcA().
  1. libA needs to call funcA() explicitly while other libraries do not have a similar need or meet the need transparently. We define an interface function in XOMP for funcA(). The implementation of the XOMP function is conditional based on the target RTL, either calling funcA() for libA or doing nothing for all others. Compiler translation targets the same XOMP interface as if all RTLs had the explicit need.
  2. funcA() implements some common functionality that is indeed suitable to be put into an RTL. Other libraries lack the similar support and rely on compiler translation too much. We define an XOMP function for the common functionality. The XOMP function either calls funcA() for libA or implements the functionality that is absent in other RTLs. Compiler translation targets the XOMP function.
  3. funcA() implements some functionality that is better suited to direct implementation by compiler translation. We develop the compiler translation to generate statements to implement the functionality without leveraging any runtime support. Still, the compiler translation can work with all RTLs.
- **Rule 3.** Occasionally, none of the above options may apply nicely. For example, the translation methods and the corresponding runtime support for an OpenMP construct can be dramatically different. In this case, we expose all the runtime functions in XOMP and have different translations for different XOMP support depending on the choice of implementation.

Finally, OpenMP translations share many similar tasks regardless of their target RTLs. These tasks include generating an outlined function to be passed to each thread, variable handling for shared and private data, and replacing directives with a function call. We have developed a set of AST transformations to support these common tasks. For example, the ROSE outliner [7] is a general-purpose tool to extract code portions from both C and C++ to create functions. It automatically handles variable passing according to variable scope and use information.

3.3 Translation Algorithm

We use the following translation algorithm for each input source file that uses OpenMP:

1. Use a top-down AST traversal to make implicit data-sharing attributes explicit, including implicit **private** loop index variables for loop constructs and implicit **firstprivate** variables for task constructs.
2. Use a bottom-up AST traversal to locate OpenMP nodes and to perform necessary translations.
   (a) Handle variables if they are listed within any of **private**, **firstprivate**, **lastprivate** and **reduction** clauses of a node.

(b) For (**omp parallel**) and (**omp task**) constructs, generate outlined functions as tasks and replace the original code block with XOMP runtime calls.
(c) For loop constructs, normalize target loops and generate code to calculate iteration chunks for each thread, with the help from XOMP loop scheduling functions.
(d) Translation for other constructs, such as **barrier**, **single**, and **critical**, are relatively straightforward [11].

Our algorithm handles variables with OpenMP data-sharing attributes in a separate phase before other translation activities. Thus, we eliminate OpenMP semantics from a code segment as much as possible so the general-purpose ROSE outliner can easily handle the code segment. Combined OpenMP variable handling and outlining would otherwise force us to tweak the outliner to handle OpenMP data-sharing variables specially during outlining.

3.4 Examples

We take the GCC 4.4.1's GOMP [9] and Omni Compiler [3] (v1.6) RTLs as two examples to demonstrate the definition of XOMP and the corresponding reusable compiler translations. GOMP is a widely available OpenMP runtime library and has recently added support for the task features of OpenMP 3.0. The Omni compiler is a classic reference research compiler for OpenMP 2.0/2.5 features. Supporting these two representative RTLs within a single compiler is a good indication of extensibility of a research compiler.

Fig. 3 and Fig. 4 give an example OpenMP program that uses tasks and ROSE's OpenMP translation that targets XOMP. ROSE uses a bottom-up traversal to find OpenMP parallel and task nodes and generates three outlined functions with the help from the outliner. These outlined functions are passed to either XOMP_parallel_start() or XOMP_task() to start multithreaded execution.

Some XOMP functions, such as XOMP_parallel_start(), XOMP_barrier() and XOMP_single(), are defined based on Rule 1 as common interfaces on top of both GOMP and Omni's interfaces. Rule 2.1 applies to XOMP_init() and XOMP_terminate(), which are introduced by Omni to initialize and to terminate runtime support explicitly while GOMP does not need them. In another case, GOMP does not provide runtime support for some simple static scheduling while Omni does. We decided to use Rule 2.3, letting the translation generate statements calculating loop chunks for each thread and totally ignore any runtime support. Rule 3 applies to the implementation for **threadprivate**. GCC uses Thread-Local Storage (TLS) to implement **threadprivate** variables. The corresponding translation is simple: mostly by adding the keyword __thread in front of the original declaration for a variable declared as **threadprivate**. On the other hand, Omni uses heap storage to manage threadprivate variables and relies on more complex translation and runtime support to initialize and to access the right heap location as a private storage for each thread. These two implementations represent two common methods to support **threadprivate** that each has well-known advantages and disadvantages. As a result, we decided to support

```
1  int main ()
2  {
3  #pragma omp parallel
4    {
5  #pragma omp single
6      {
7        int i;
8  #pragma omp task untied
9        {
10         for (i = 0; i < 5000; i++)
11         {
12 #pragma omp task if(1)
13           process (item[i]);
14         }
15       }
16     }
17   }
18   return 0;
19 }
```

Fig. 3. An example using tasks

both methods and to use different translation and/or runtime support conditionally depending on the choice of the final target RTL. XOMP_task is an exceptional case since Omni does not have corresponding support and we defined it based on GOMP's interface. In summary, less than 20% of the XOMP functions are defined using Rule 3. This means that more than 80% of the OpenMP translation can be reused across multiple RTLs.

Leveraging ROSE's robust C++ support, we are also able to implement OpenMP translation for C++ applications. Fig. 6 shows the translation result of an example C++ program shown in Fig. 5. The ROSE outliner supports generating an outlined function with C-bindings at global scope from a code segment within a C++ member function. This binding choice is helpful since the thread handling functions of most OpenMP RTLs expect a pointer to a C function, not a C++ one. The outlined function at line 22 is also declared as a friend (at line 11) in the host class to access all class members legally.

4 Evaluation

We evaluate ROSE's support for both OpenMP analysis and translation.

4.1 OpenMP Analysis

We have used ROSE to build a simple analysis tool that can detect a common mistake of using OpenMP locks. As Fig. 7 shows, a lock variable (at line 3) is declared within a parallel region and then used within that same parallel region, which is incorrect since a lock must be **shared** to be effective. A locally declared lock is **private** to each thread.

Fig. 8 shows the ROSE AST analysis code (slightly simplified) that can find this error in using locks. Programmers only need to create a class(OmpPrivateLock) by inheriting a builtin AST traverse class in ROSE and to provide a visitor

```
1  #include "libxomp.h"
2  struct OUT__1__1527___data {int i;};
3  struct OUT__2__1527___data {int i;};
4
5  static void OUT__1__1527__(void *__out_argv)
6  {
7    int i = (int )(((struct OUT__1__1527___data *)__out_argv) -> i);
8    int _p_i = i;
9    process((item[_p_i]));
10 }
11
12 static void OUT__2__1527__(void *__out_argv)
13 {
14   int i = (int )(((struct OUT__2__1527___data *)__out_argv) -> i);
15   int _p_i = i;
16   for (_p_i = 0; _p_i < 5000; _p_i++) {
17     struct OUT__1__1527___data __out_argv1__1527__;
18     __out_argv1__1527__.i = _p_i;
19     /* void XOMP_task (
20      * void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
21      * long arg_size, long arg_align, bool if_clause, bool untied )*/
22     XOMP_task(OUT__1__1527__,&__out_argv1__1527__,0,4,4,1,0);
23   }
24 }
25
26 static void OUT__3__1527__(void *__out_argv)
27 {
28   if (XOMP_single()) {
29     int i;
30     struct OUT__2__1527___data __out_argv2__1527__;
31     __out_argv2__1527__.i = i;
32     XOMP_task(OUT__2__1527__,&__out_argv2__1527__,0,4,4,1,1);
33   }
34   XOMP_barrier();
35 }
36
37 int main(int argc,int argv)
38 {
39   int status = 0;
40   XOMP_init(argc,argv);
41   /* void XOMP_parallel_start (
42    * void (*func) (void *), void *data, unsigned num_threads )*/
43   XOMP_parallel_start(OUT__3__1527__,0,0);
44   XOMP_parallel_end();
45   XOMP_terminate(status);
46   return 0;
47 }
```

Fig. 4. Translated example using tasks

function implementation. The traversal visits all AST nodes to find a use of an OpenMP lock within any of OpenMP lock routines (line 4-13). The code then detects if the use of the lock is lexically enclosed inside a parallel region (line 16-18) and if the declaration of the lock is also inside the same parallel region (line 21-22). The statement style OpenMP node(SgOmpParallelStatement) for a parallel region enables users to directly reuse AST interface functions, such as the function to find a lexically enclosing node of a given type (SageInterface::getEnclosingNode<ParentType>(node)) and another function to tell if a node is another node's ancestor (SageInterface::isAncestor(a_node, c_node)). This

```
1  class A
2  {
3    private:
4      int i;
5    public:
6      void pararun()
7      {
8  #pragma omp parallel
9        {
10 #pragma omp critical
11         cout<<"i= "<< i <<endl;
12       }
13     }
14 };
```

Fig. 5. A C++ example

```
1  #include "libxomp.h"
2  struct OUT__1__1527___data {  void *this__ptr___p; };
3  static void OUT__1__1527__(void *__out_argv);
4  static void *xomp_critical_user_;
5
6  class A
7  {
8    private:
9      int i;
10   public:
11     friend void ::OUT__1__1527__(void *__out_argv);
12     void pararun()
13     {
14       class A *this__ptr__ = this;
15       struct OUT__1__1527___data __out_argv1__1527__;
16       __out_argv1__1527__.this__ptr___p = (void *)this__ptr__;
17       XOMP_parallel_start(OUT__1__1527__,&__out_argv1__1527__,0);
18       XOMP_parallel_end();
19     }
20 };
21
22 static void OUT__1__1527__(void *__out_argv)
23 {
24   class A *this__ptr__ =
25        (class A *)(((struct OUT__1__1527___data *)__out_argv) -> this__ptr___p);
26   XOMP_critical_start(&xomp_critical_user_);
27   std::cout<<"i= "<<( *this__ptr__).i<<std::endl;
28   XOMP_critical_end(&xomp_critical_user_);
29 }
```

Fig. 6. Translated C++ example

```
1  #pragma omp parallel
2    {
3      omp_lock_t lck;
4      omp_set_lock(&lck);
5      printf("Thread = %d\n", omp_get_thread_num());
6      omp_unset_lock(&lck);
7    }
```

Fig. 7. Using a private lock

example demonstrates that writing analysis tools using ROSE is straightforward since OpenMP constructs are represented as nodes that are inherently part of the ROSE AST.

```
1  void OmpPrivateLock::visit(SgNode* node)
2  {
3    //1. Find an OpenMP lock routine
4    SgFunctionCallExp * func_call = isSgFunctionCallExp(node);
5    if (!func_call) return;
6    std::string f_name = func_call->get_name();
7    if (f_name != "omp_unset_lock" && f_name != "omp_set_lock"
8        && f_name != "omp_test_lock")     return;
9
10   //2. Grab the only routine parameter as the use of a lock
11   std::vector<SgVarRefExp*> exp_vec =
12       SageInterface::querySubTree<SgVarRefExp>(func_call, V_SgVarRefExp);
13   ROSE_ASSERT(exp_vec.size() ==1);
14
15   //3. If the lock's use is inside a parallel region
16   SgOmpParallelStatement* lock_region =
17     SageInterface::getEnclosingNode<SgOmpParallelStatement>(exp_vec[0]);
18   if (lock_region)
19   {
20   //4. Check if the lock declaration is also inside the same region
21      SgVariableDeclaration* lock_decl = exp_vec[0]->get_declaration();
22     if (SageInterface::isAncestor(lock_region, lock_decl))
23       cerr<<"Found a private lock within a parallel region"<<endl;
24   }
25 }
```

Fig. 8. A ROSE-based tool to find private locks

4.2 OpenMP Translation

We have evaluated ROSE's OpenMP translations and the corresponding XOMP interface through a set of OpenMP benchmarks, including the NAS Parallel Benchmarks(NPB)[12] and the Barcelona OpenMP Task Suite (BOTS) [13], Those benchmarks have builtin correctness verification so they also test the correctness of our compiler implementations. We ran all experiments on a Dell T5400 workstation with dual processors and 8 GB of memory. Each of the processors is a 3.16 GHz quad-core Intel Xeon X5460 processor. We used several other OpenMP compilers in addition to ROSE, including GCC 4.4.1, Intel Compilers 11.1.059, and the Mercurium 1.3.3 compiler with the Nanos 4.1.4 runtime. We used GCC 4.4.1 as the backend compiler for all source-to-source implementations. We used compiler option *-O3* whenever possible.

Fig. 9 shows the speedup of a subset of NPB (V 2.3 C version [14]) and BOTS V 1.0 using up to 8 threads by different compiler/runtime configurations. Results for the remaining benchmarks had similar patterns and are not shown for brevity. ROSE-Omni's speedup for the BOTS benchmarks (NQUEEN, SORT, and STRASSEN) is not available since the Omni runtime library does not support OpenMP tasking. In general, all implementations had comparable performance. ROSE's source-to-source translation and extra layer of runtime support do not incur any significant performance overhead compared to other compilers.

5 Related Work

Some other OpenMP research compilers exist. Representative examples include Omni [3], OdinMP [15] and OpenUH [11]. Most research compilers adopt the

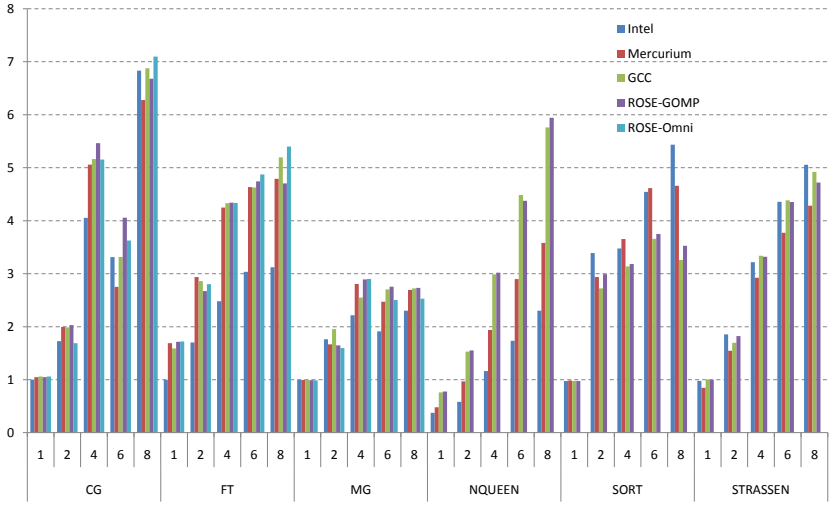

Fig. 9. Speedup of some NPB 2.3 and BOTS 1.0 benchmarks

source-to-source translation approach. Based on Open64, OpenUH supports both source-to-source translation and generating the final binary code by itself. The Nanos Mercurium compiler [16] is another source-to-source compiler aimed at fast prototyping for OpenMP. It was among the first to support OpenMP 3.0's task feature and was used to evaluate the expressiveness and flexibility of OpenMP task directives compared to using nested parallelism and Intel's taskqueues. More recently, Addison et. al. [17] presented the OpenMP 3.0 implementation in OpenUH [11] with an extended runtime system supporting tasking. However, the corresponding compiler translation was done manually, as reported in their paper. Leveraging GCC 4.4's runtime library, ROSE is one of the few OpenMP compilers supporting OpenMP 3.0. It might be the only OpenMP research compiler with stable C++ source-to-source support, although both OpenUH and Mercurium have a similar goal. Finally, ROSE's XOMP translation interface enables ROSE to implement translations targeting different RTLs quickly as demonstrated in this paper. Other compilers usually target only a single RTL.

6 Conclusion

In this paper, we have presented ROSE as an OpenMP 3.0 research compiler for C/C++ and Fortran. ROSE's OpenMP support includes extensions to ROSE's AST to represent OpenMP constructs, a common runtime support interface (XOMP), and a set of reusable translations that can target multiple OpenMP runtime libraries. An automatic parallelization module is also available in ROSE. Our AST representation for OpenMP is inherently part of the ROSE AST so

most existing AST manipulation, analysis, and transformation interface functions can be easily reused to handle OpenMP applications. Preliminary evaluation demonstrates that it is straightforward to write static analysis tools for OpenMP. Also, ROSE's OpenMP translation targeting two mainstream OpenMP RTLs has competitive performance compared to other OpenMP implementations. The latest ROSE OpenMP support has been released as part of the ROSE distribution (downloadable from our website [2]).

In the future, we plan to add the OpenMP Fortran support and to complete the OpenMP 3.0 implementation, such as loop collapse. We will build more static analysis tools to help users write correct OpenMP applications. With ROSE's unique C++ support, we are interested in exploring more C++-related issues within OpenMP. The introduction of explicit tasks in OpenMP 3.0 gives implementations and users more choices to optimize parameters related to tasks, such as the cut-off depth of tasks, tied or untied tasks, or task scheduling policies (including task aggregation granularity). We expect that empirical tuning can play an important role in finding the best OpenMP compilation and execution parameters for a given application on a particular platform. Finally, we especially welcome external collaborations using ROSE for research specific to the requirements of the OpenMP research community.

References

1. OpenMP Architecture Review Board: OpenMP application program interface, version 3.0 (2008), `http://www.openmp.org/mp-documents/spec30.pdf`
2. Quinlan, D.J., et al.: ROSE compiler project, `http://www.rosecompiler.org/`
3. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP compiler for an SMP cluster. In: The 1st European Workshop on OpenMP (EWOMP'99), September 1999, pp. 32–39 (1999)
4. Quinlan, D.: ROSE: Compiler support for object-oriented frameworks. In: Proceedings of Conference on Parallel Compilers, CPC (2000)
5. Edison Design Group: C++ Front End, `http://www.edg.com`
6. Rasmussen, C., et al.: Open Fortran Parser, `http://fortran-parser.sourceforge.net/`
7. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: 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)
8. Yi, Q., Quinlan, D.: Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 253–267. Springer, Heidelberg (2005)
9. GOMP - an OpenMP implementation for GCC (2005), `http://gcc.gnu.org/projects/gomp`
10. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 28–41. Springer, Heidelberg (2009)

11. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: an optimizing, portable OpenMP compiler. Concurrency and Computation: Practice and Experience 19(18), 2317–2332 (2007)
12. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center (1999)
13. Barcelona OpenMP task suite, `http://nanos.ac.upc.edu/content/barcelona-openmp-task-suite`
14. C version NPB 2.3 in OpenMP, `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html`
15. Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience 12(12), 1193–1203 (2000)
16. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model, pp. 63–77 (2008)
17. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. In: Open64 Workshop at CGO 2009 (2009)