LLNL-CONF-422873

A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries

C. Liao, D. Quinlan, T. Panas

January 26, 2010

The International Workshop on OpenMP
Tsukuba, Japan
June 14, 2010 through June 16, 2010

A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries [⋆]

Chunhua Liao , Dan Quinlan and Thomas Panas

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

Abstract. OpenMP is a popular and evolving programming model for shared-memory platforms. It relies on compilers for optimal performance and to target modern hardware architectures. 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 of the OpenMP 3.0 constructs and facilitate their manipulation. Since OpenMP research is often complicated by the tight coupling of the compiler translations 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 and present comparative performance results against other OpenMP compilers.

1 Introduction

OpenMP 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 become available, OpenMP has become a rapidly evolving programming model; numerous improvements are being proposed to broaden the range of hardware architectures it can

[⋆] This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. We thank Bronis de Supinski for his suggestion of emphasizing supporting multiple runtime libraries.

accommodate. A variety of robust and extensible compiler implementations are the key to OpenMP's sustainable success in the future. The reason is that it is an OpenMP compiler's responsibility to delivery portable performance. Open source OpenMP compilers permit active research for this rapidly evolving programming model.

Developed at Lawrence Livermore National Laboratory, the ROSE compiler [1] 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 faithfully represent the latest OpenMP 3.0 constructs and facilitate their manipulation; allowing the construction of custom OpenMP analysis tools.

More generally, OpenMP research is often complicated by the tight coupling of the compiler translations and the runtime system upon which they are dependent. It is often a major effort to change the existing compiler translations to utilize a new runtime library. Conversely, it can be difficult to change the runtime system where new features require support from compiler translations. As a result, this tight coupling impedes research work on the OpenMP programing model. We seek to use ROSE as a testbed to decouple the compiler translations from dependence upon the OpenMP runtime libraries. A common runtime library interface and a set of corresponding compiler translations have been designed and developed 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 both GCC 4.4 and Omni 1.6's OpenMP runtime libraries.

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 runtime library, and translation support. Section 4 presents a preliminary evaluation of ROSE's OpenMP support. Related work is discussed in Section 5. Section 6 concludes this paper and discusses future work.

2 The ROSE Compiler

ROSE [1] 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 (IR) with a set of analysis and transformation interfaces allowing users to quickly build translators, analyzers, optimizers, and specialized tools. The intended users of ROSE could be either experienced compiler researchers or library and tool developers who may have minimal compiler experience.

A representative translator built using ROSE works as follows. The EDG [2] front-end is used to parse C and C++ applications. Language support for Fortran 2003 (and earlier versions) is based on the open source Open Fortran Parser (OFP) [3]. 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. Example AST analysis interface supports analysis for call graph, control flow, data flow(live variables, def-use chain, reaching definition, alias analysis, etc.), class hierarchy, data dependence and system dependence. Representative program optimization and translation interfaces cover partial redundancy elimination, constant folding, inlining, outlining [4], and loop transformations. Through a mechanism called persistent attribute, the ROSE AST also allows user-defined data to be attached to any node as a way to extend the IR to store any 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, a vendor compiler is optionally called to continue the compilation of the generated (transformed) source code; generating a final executable.

3 OpenMP Support in ROSE

ROSE supports parsing OpenMP constructs, 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 runtime libraries, with the help from XOMP, a common OpenMP runtime library that abstracts the details of any specific runtime libraries.

3.1 Parsing and Representing OpenMP

Neither EDG (as of version 4.0 and earlier versions) nor OFP recognizes 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 only minimum changes are needed to existing ROSE AST. In fact, this light-weight representation for OpenMP is also used as the output of ROSE's automatic parallelization module [5]. 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.

After that, a conversion phase is used to convert 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, most existing AST traversal, query, scope comparison, and other manipulation interfaces developed with ROSE can be directly reused 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

A major task of an OpenMP implementation is to translate OpenMP applications into multi-threaded 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. It is up to an implementation to decide on what work to be put into a runtime library and the way compiler translation interacts with a 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. For an OpenMP research compiler for OpenMP, it would be especially desirable to support multiple OpenMP runtime libraries.

Fortunately, although the interface varies from one library to another, there are many similar or overlapped runtime library functions. For example, most portable OpenMP runtime libraries rely on the Pthreads API to create and manipulate threads. Such a library usually provides a function which accepts a function pointer and a parameter to start multiple threads. The same is true for loop scheduling. Many loop scheduling policies have well defined behaviors and the runtime support for them significantly overlap.

We have introduced a common OpenMP runtime library, XOMP, so that minimal changes are needed in ROSE to support multiple OpenMP runtime libraries (RTLs) . Depending on the similarity among runtime libraries, three rules are used 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 minor differences) before calling different target RTLs internally. By doing this, one translation targeting XOMP's functions can be reused across multiple RTLs.

- **Rule 2**. A target RTL (referred as libA) has an extra function (referred as funcA()) compared to others.
  1. This may be caused by some need to explicitly call funcA() to work with libA while other libraries do not have similar need or meet the need transparently. We define an interface function in XOMP for this function. The XOMP function's implementation is conditional based on target runtime libraries, 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 which is indeed suitable to be put into a runtime library. 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 which is absent in another RTLs. Compiler translation targets the XOMP function.
  3. funcA() implements some functionality which is indeed suitable to be directly implemented by compiler translation. We develop 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**. There may be a situation that none of the above options 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 runtime libraries. These tasks include generating an outlined function to be passed to each thread, variable handling for shared and private data, replacing directives with a function call, and so on. We have developed a set of AST transformation functions to support these common tasks. For example, the ROSE outliner [4] 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 using OpenMP:

1. Use a top-down AST traversal to make implicit data-sharing attribute 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 performance 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. Details have been reported in other papers [6].

As we can see, our algorithm handles variables with OpenMP data-sharing attributes in a separated phase before the rest translation. The motivation is to 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 specially handle OpenMP data-sharing variables during outlining, which is undesirable.

3.4 Examples

We take the GCC 4.4.1's GOMP [7] library and Omni Compiler [8] (v1.6)'s runtime library 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 runtime libraries within a single compiler is a good indication of extensibility of a research compiler.

Fig. 1 and Fig. 3 give an example OpenMP program using tasks and ROSE's OpenMP translation result for it targeting 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 explicitly initialize and terminate runtime support 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 to 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

```
int main ()
{
#pragma omp parallel
  {
#pragma omp single
    {
      int i;
#pragma omp task untied
      {
        for (i = 0; i < 5000; i++)
        {
#pragma omp task if(1)
          process (item[i]);
        }
      }
    }
  }
  return 0;
}
```

Fig. 1. An example using tasks

```
class A
{
  private:
    int i;
  public:
    void pararun()
    {
#pragma omp parallel
      {
#pragma omp critical
        cout<<"i= "<< i <<endl;
      }
    }
};
```

Fig. 2. A C++ example

declared as **threadprivate**. On the other hand, Omni uses heap to manage threadprivate variables and relies on more complex translation and runtime support to initialize, access the right place of heap as a private storage for each thread. These two implementations represent two main methods to support **threadprivate**. Each of them has well-known advantages and disadvantages. As a result, we decided to support both methods and conditionally use different translation and/or runtime support depending on the choice of the final target runtime library. XOMP_task is an exception 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. 4 shows the translation result of an example C++ program shown in Fig. 2. The ROSE outliner supports generating an outlined function with C-bindings at global scope from a code segment within a C++ member function. This is helpful since most OpenMP runtime library's thread handling functions 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 legally access all class members.

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 which can detect a common mistake of using OpenMP locks. As shown in Fig. 5, a lock variable (at line 3) is

```
#include "libxomp.h"
struct OUT__1__1527___data {int i;};
struct OUT__2__1527___data {int i;};

static void OUT__1__1527__(void *__out_argv)
{
  int i = (int )(((struct OUT__1__1527___data *)__out_argv) -> i);
  int _p_i = i;
  process((item[_p_i]));
}

static void OUT__2__1527__(void *__out_argv)
{
  int i = (int )(((struct OUT__2__1527___data *)__out_argv) -> i);
  int _p_i = i;
  for (_p_i = 0; _p_i < 5000; _p_i++) {
    struct OUT__1__1527___data __out_argv1__1527__;
    __out_argv1__1527__.i = _p_i;
    /* void XOMP_task (
     * void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
     * long arg_size, long arg_align, bool if_clause, bool untied )*/
    XOMP_task(OUT__1__1527__,&__out_argv1__1527__,0,4,4,1,0);
  }
}

static void OUT__3__1527__(void *__out_argv)
{
  if (XOMP_single()) {
    int i;
    struct OUT__2__1527___data __out_argv2__1527__;
    __out_argv2__1527__.i = i;
    XOMP_task(OUT__2__1527__,&__out_argv2__1527__,0,4,4,1,1);
  }
  XOMP_barrier();
}

int main(int argc,int argv)
{
  int status = 0;
  XOMP_init(argc,argv);
  /* void XOMP_parallel_start (
   * void (*func) (void *), void *data, unsigned num_threads )*/
  XOMP_parallel_start(OUT__3__1527__,0,0);
  XOMP_parallel_end();
  XOMP_terminate(status);
  return 0;
}
```

Fig. 3. Translated example using tasks

declared within a parallel region and then used within the same parallel region. This won't work since a lock has to be **shared** to be effective. A locally declared lock is **private** to each thread.

Fig. 6 shows the ROSE AST analysis code (slightly simplified) needed to find a mistaken use of locks mentioned above. Programmers only need to create a class(OmpPrivateLock) by inheriting a builtin AST traverse class in ROSE and provide a visitor function implementation. All AST nodes are visited during a traversal 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

```
#include "libxomp.h"
struct OUT__1__1527___data {  void *this__ptr___p; };
static void OUT__1__1527__(void *__out_argv);
static void *xomp_critical_user_;

class A
{
  private:
    int i;
  public:
    friend void ::OUT__1__1527__(void *__out_argv);
    void pararun()
    {
      class A *this__ptr__ = this;
      struct OUT__1__1527___data __out_argv1__1527__;
      __out_argv1__1527__.this__ptr___p = (void *)this__ptr__;
      XOMP_parallel_start(OUT__1__1527__,&__out_argv1__1527__,0);
      XOMP_parallel_end();
    }
};

static void OUT__1__1527__(void *__out_argv)
{
  class A *this__ptr__ =
       (class A *)(((struct OUT__1__1527___data *)__out_argv) -> this__ptr___p);
  XOMP_critical_start(&xomp_critical_user_);
  std::cout<<"i=␣"<<( *this__ptr__).i<<std::endl;
  XOMP_critical_end(&xomp_critical_user_);
}
```

Fig. 4. Translated C++ example

```
#pragma omp parallel
  {
    omp_lock_t lck;
    omp_set_lock(&lck);
    printf("Thread␣=␣%d\n", omp_get_thread_num());
    omp_unset_lock(&lck);
  }
```

Fig. 5. Using a private lock

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 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)).

As demonstrated by the example, writing analysis tools using ROSE is straightforward since OpenMP constructs are represented as nodes which are inherently part of the ROSE AST.

4.2 OpenMP Translation

A set of OpenMP benchmarks, including the NAS Parallel Benchmarks(NPB)[9] and the Barcelona OpenMP Task Suite (BOTS) [10], have been used to evaluate ROSE's OpenMP translations and the corresponding XOMP interface. Those

```
void OmpPrivateLock::visit(SgNode* node)
{
  //1. Find an OpenMP lock routine
  SgFunctionCallExp * func_call = isSgFunctionCallExp(node);
  if (!func_call) return;
  std::string f_name = func_call->get_name();
  if (f_name != "omp_unset_lock" && f_name != "omp_set_lock"
      && f_name != "omp_test_lock")      return;

  //2. Grab the only routine parameter as the use of a lock
  std::vector<SgVarRefExp*> exp_vec =
       SageInterface::querySubTree<SgVarRefExp>(func_call, V_SgVarRefExp);
  ROSE_ASSERT(exp_vec.size() ==1);

  //3. If the lock's use is inside a parallel region
  SgOmpParallelStatement* lock_region =
    SageInterface::getEnclosingNode<SgOmpParallelStatement>(exp_vec[0]);
  if (lock_region)
  {
  //4. Check if the lock declaration is also inside the same region
      SgVariableDeclaration* lock_decl = exp_vec[0]->get_declaration();
    if (SageInterface::isAncestor(lock_region, lock_decl))
      cerr<<"Found␣a␣private␣lock␣within␣a␣parallel␣region"<<endl;
  }
}
```

Fig. 6. A ROSE-based tool to find private locks

benchmarks have builtin correctness verification so they also test the correctness of compiler implementations. All experiments were run 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. In addition to ROSE, several other OpenMP compilers were also used. They include GCC 4.4.1, Intel Compilers 11.1.059, and the Mercurium 1.3.3 compiler with Nanos 4.1.4 runtime. GCC 4.4.1 was used as the backend compiler for all source-to-source implementations. Compiler option *-O3* was used whenever possible.

Fig. 7 shows the speedup of a subset of NPB (V 2.3 C version [11]) 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 have any significant performance disadvantages compared to other compilers.

5 Related Work

Some other OpenMP research compilers exist. Representative examples include Omni [8], OdinMP [12] and OpenUH [6]. Most research compilers adopt the 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 [13] is another source-to-source compiler

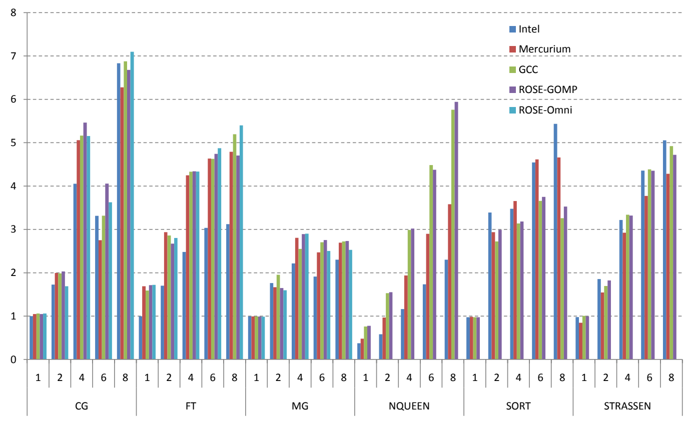

Fig. 7. Speedup of some NPB 2.3 and BOTS 1.0 benchmarks

aimed at fast prototyping for OpenMP. It was among the first to support the 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. [14] presented the OpenMP 3.0 implementation in OpenUH [6] 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 similar goal. Finally, ROSE's XOMP easy translation interface enables ROSE to quickly implement translation targeting different runtime libraries as demonstrated in this paper. Other compilers usually target only a single runtime library.

6 Conclusion

In this paper, we have presented ROSE as an OpenMP 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 which can target multiple OpenMP runtime libraries. Our AST representation for OpenMP is inherently part of the ROSE AST so most existing AST manipulating, 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 runtime libraries 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 [1]).

In the future, we plan to add the OpenMP Fortran support and 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) and so on. 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. Quinlan, D.J., et al.: ROSE compiler project. `http://www.rosecompiler.org/`
2. Edison Design Group: C++ Front End. `http://www.edg.com`
3. Rasmussen, C., et al.: Open Fortran Parser. `http://fortran-parser.sourceforge.net/`
4. 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)
5. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: IWOMP '09: Proceedings of the 5th International Workshop on OpenMP, Berlin, Heidelberg, Springer-Verlag (2009) 28–41
6. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: an optimizing, portable OpenMP compiler. Concurrency and Computation: Practice and Experience **19**(18) (2007) 2317–2332
7. : GOMP - an OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp` (2005)
8. 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) 32–39
9. 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)
10. : Barcelona OpenMP task suite. `http://nanos.ac.upc.edu/content/barcelona-openmp-task-suite`
11. : C version NPB 2.3 in OpenMP. `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html`
12. Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience **12**(12) (2000) 1193–1203
13. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. (2008) 63–77
14. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. In: Open64 Workshop at CGO 2009. (2009)