Automated test generation for optimizing compilers with OpenMP support

Svyatoslav Pankratov[1],*

[1]A.P. Ershov Institute of Informatics Systems of the SB RAS, 630090, Pr. Akademika Lavrentyeva, 6, Novosibirsk, Russia

Abstract. The correctness of the compiler is a necessary requirement for the correct operation of the software compiled by it. Therefore, the most important stage in the development of the compiler is verification. Recent widespread of multi-core processors and graphics core integrated to CPU emphasized the problem of the transition from single-threaded to multi-threaded computing and re-usage of graphics core for general purpose heterogeneous parallel computations in particular. In this paper, we are presenting an approach to automate test creation for the verification of the compiler with OpenMP support, based on a generator that uses grammars to generate syntactically correct executable tests.

1 Introduction

High-level programming languages are the main development tool for software. The task of translating the source code of programs into a representation executable on a computer system is solved by the compiler. Compilers have high quality requirements since correctness of compiled programs significantly depends on compiler correctness. And software defects caused by errors in the compiler are difficult to identify, and impossible to correct without interference in the compiler itself. The correctness of the compiler is a necessary requirement for the correct operation of the software compiled by it. Therefore, the most important stage in the development of the compiler is verification.

Due to complexity of programs as compiler input data and their transformations, the task of compiler verification is very tedious and difficult. And in the case of using an optimizing compiler, it is also algorithmically unsolvable. But we can consider the behaviour of the compiler on some limited class of programs. We can not compare the result of the transformation with some reference, we can only consider some properties of the translation produced. For example, if the compilation of the same program without optimizations succeeds, and with optimizations leads fails in compiler, then we can talk about the presence of errors in the optimization code.

Recent widespread of multi-core processors and graphics core integrated to CPU emphasized the problem of the transition from single-threaded to multi-threaded computing and re-usage of graphics core for general purpose heterogeneous parallel computations in particular. To support these trends from the software side there are open standards for parallel computing: OpenMP, OpenCL, Cilk Plus, and others that are supported by major mainstream compilers. Taking all these facts into account, testing of compiler's implementation of parallel C extension is particularly important.

Automatic test generation is an important part of the supplemental testing, because tests written by hand usually can't effectively cover all possible combinations of language constructions, as well as all situations of application optimizations. The QA command of the Intel compiler chose an approach using a parametric context-free grammar, described by a special meta-language for test generator. Parametric context-free grammars have proven themselves as a good formalism for constructing generators of semantically correct and compiler-specific tests with deterministic behaviour. They were distinguished by the clarity of the description of generated test programs, as well as the flexibility and convenience in working with their context. However, this approach was used only to generate single-threaded tests executed on the central processor. Therefore it is an open question about the possibility of using an already existing generator of single-threaded tests based on parametric context-free grammars, for generating tests for multi-threaded extensions of the C language, like OpenMP.

2 The problem of compiler testing

The programming language whose strings can be provided as a compiler input is described by specifications of its syntax, static semantics and dynamic semantics. Test generation can be based on any of these language specifications. These specifications define the set of nested subsets of all possible generated tests, so that program with correct dynamic semantic should has correct static semantic and program with correct static semantic should have correct syntax.

The syntax specification is defined by the formal grammar [10]. The grammar consists of the following components:

*e-mail: aquaxpi@gmail.com

- finite set of nonterminal symbols;
- finite set of terminal symbols;
- finite set of production rules;
- distinguished symbol, that is the start symbol.

Sequences of terminal symbols that can be derived from the start symbol of the grammar are called syntactically correct programs. The set of syntactically correct programs is a subset of the set of all sequences of terminal symbols. Unfortunately, these programs can be used only for compiler's parser verification since they cannot be always compiled.

Static semantics is defined only for syntactically correct programs; it defines rules for computing program properties that can be determined without program execution. These properties include, in particular, types of variables and expressions. Rules for checking static program correctness (context conditions) impose constraints on possible combinations of values of static program properties. Programs that meet these static context conditions can be compiled and used to test the static analyzer of the compiler.

Dynamic semantics of a programming language defines the meaning of the execution of statically correct programs in given language. To test dynamic semantics implementation in the compiler, we compile statically correct programs and execute them to match their observable behavior with the reference behavior determined by the reference program implementation. Reference program could be obtained by using other compiler that is considered as error free or the same compiler without optimizations. Programs with correct dynamic semantic should produce the same output as reference implementation unless there is an error in a compiler implementation. In this article, we'll make an assumption that *programs with correct dynamic semantic are deterministic programs*. Of course it's not true in common case, but OK for testing purposes.

In general, for mainstream programming languages, such as C/C++, automatic determinism evaluation of any program is very complex and algorithmically unsolvable problem for static analysis [2].

Syntactically valid test generation is not a difficult task and usually it's done on the basis of context-free grammar that is used for a target language specification. However, the generation of the compilable (statically semantically correct) program is much more difficult task, because it requires compliance with all the contextual constraints of the target language which is usually done with help of context-sensitive grammars. Generation of deterministic programs (dynamic semantically correct) is even more complex issue.

Generators of deterministic programs are usually a monolithic programs written on a high-level language. These generators are usually hard to extend and they can be used only for the generation of programs of a certain class. There is also an approach to generate executable programs by using a certain set of predefined patterns with a given set of variations, but this method also isn't flexible and has complexity almost equal to the manual test creation.

To avoid writing another monolithic test generator Intel Compiler QA team decided to use parametric context-free grammar described by a special meta-language [1]. Parametric context-free grammars were proved to be a good formalism to construct generators of semantically correct tests with deterministic behaviour. They provide simple formalism to specify program syntax structure like context-free grammars, but allow generation of a broader class of context-sensitive languages. However, this approach was used only for the generation of single-threaded programs that run on the CPU. Therefore it is an open question if it is possible to use it for an efficient generation of parallel heterogeneous programs or there is a more suitable solution? In this paper we will try to answer this question.

3 Related works

At the moment, there are many works about automatic test generation, we will describe only majors.

In earlier studies of test generation for compiler, K.V. Hanford [6] and P. Pardom [9] presented methods to generate syntactically correct programs for procedural language compilers, without regard to the rules of static semantics.

Hanford's work [6] was published in 1970. He proposed a method for generating test data for PL/1 compiler based on the dynamic grammar. This method produces syntactically corrected programs, but part of this are semantically incorrect.

P. Pardom's work [9] had became fundamental in the field of test generation for compilers and served as a starting point for further researches.

A.G. Duncan and J.S. Hutchison [5] presented a method for generating test cases that can be used throughout the entire life cycle of a program. This method uses attributed grammars as input for generator. If it possible due contextual constraints, all non-terminal symbols will be consistently disclosed during generation process. Thus, as result we have a set of syntactically and semantically correct tests, that covers all production rules of grammar and satisfy to all contextual conditions. This approach allows only conduct analysis of performed contextual conditions. This aspect leads to a large number of pointless runs of the generator, because we need to terminate generation process if we have unmet contextual conditions.

Bazichi and Spadafora [3] presented a new method, when generator is driven by a tabular description of source language. This description is in a formalism which nicely extends context-free grammars in a context-dependent direction, but still retains the structure and readability of BNF. Unfortunately, authors suggested to use non-terminal symbols for recognising the left part of rule, and it may take a huge time if chain of symbols is long. Also paper doesn't solves a problem of finding short grammar's rules.

Work of A.Stasenko [1] can be reviewed as a successor of Bazzichi and Spadafora's work. It uses parametric context-free grammar for generating syntactically and semantically correct test cases for C/C++/Fortran compilers.

As we decided to use the same formalism, we will explain it later.

4 Approach to the problem

4.1 Main idea

As part of this work we've extended existing parametric context-free grammar (generates serial C tests) for test generation with OpenMP extension. We focused on generation of parallel extensions of standard C loops. It's a trick, but it will help to rump up support of other major parallelization C extensions, such us Cilk, CilkPlus, UPC, OpenHMPP and GFX-offload.

Focus on generating various parallel loops allows to improve compiler vectorization (data parallelism) optimization validation which also play important role in providing performance in modern computing architectures. OpenMP implementation have some restrictions about a parallel loops such as follows:

– no transition (return, break or goto) from the loop is allowed;

– restrictions on type, range and increment statement of control loop variable;

– restrictions on mixing construction of various parallel language extensions (for example, OpenMP block cannot use _Cilk_spawn and _Cilk_sync primitives).

4.2 A brief formal description of the parametric context-free grammar

To describe the grammar we use a special language - mix of BNF-notation and functional language. The grammar is specified in a single text file that consists of strings that specify list of rules and comments. Rule consists of left and right parts. Left part consists of identifier, context parameters, context recognizers and context conditions. Right part consists of set of rewrite alternatives or multiline string. Identifier is a sequence of letters that does not start with a digit. Context parameters can be specified by simple name or in a list like manner. Context recognizers are used to give names to different parts of the context parameters and to form other named objects to be used later in the rule. The context condition is an expression that evaluates to a boolean type. And if rule contains several conditions, they are joined by logical operation "AND". Also there are several built-in functions to ease work with lists, conditions and numbers.

During generation the choice between rule alternatives is made randomly. Of course that does not guarantee reachability of all specified rules, however, as shown by other studies in this area, such problem is typical for other testing approaches [8]. Also the problem of infinite generation is solved automatically this way: after a specified number of generated symbols generator starts to select the shortest alternative if it exists. Generation approach used in this paper was described in the article by Stasenko A.P. [2].

4.3 Generator infrastructure

Grammar and generator are not enough to organize a serious testing process. We also need a system that will validate the generated tests as it was created by the authors of the generator. It's was named as Test Generator Harness (hereinafter - TGH) and allows to automatically test the compiler via generated tests. Block scheme of the system is shown on Figure 1.

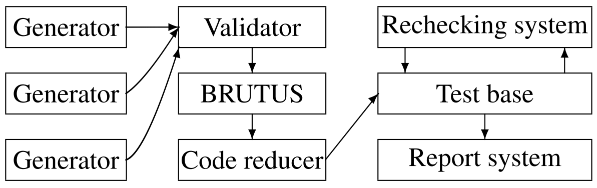

Figure 1. Generator infrastructure

It's written on Python language and could be used in virtualized environments, such Docker or KVM.

Let's take a closer look to a whole system process flow. Each generated test is compiled two times with and without optimizations. If the test with additional pointer and type checks is successfully compiled without compiler optimizations and executed without crashes we consider it to be a statically and dynamically correct program and its results are used later as reference for test compiled with compiler optimizations. The successful reference test execution assumes the lack of runtime reports about problems with pointers and types, zero exit code and execution without exceeding the specified timeout. This is the first stage of testing. The next step is to check the compilation optimizations. If the test at this point caused the compiler problem it is stored for further processing. If the compilation is successful, we check execution results and compare it with reference results. If the program fails on runtime then it is also stored. In the other case test is removed and TGH starts the process of new test generation.

Also Intel compiler provides special functionality that allows TGH to selectively disable optimizations and it may help to find guilty compiler component (optimization). Name of this internal compiler's functionality is BRUTUS. So we try to find guilty optimization and store information about it with failed test.

Failed tests are passed to the tool (Reduce) that iteratively removes parts of the test program that are not important for compiler problem reproduction. Reduced failing tests are stored in the testbase with notes about platform (combination of hardware and software components), guilty optimization and computer system where problem was found.

To optimize usage of storage space, we store only the smallest 10 test for each different optimization. The system periodically rechecks all the tests from the testbase to remove tests that start to pass on all platforms. In addition to checking for the latest version of the compiler, system checks test on product branch of compiler to avoid getting these errors in the final product. Every day, TGH sends reports to QA engineer with list of compiler optimizations that have failing tests that are not marked by compiler defect in a bug tracking system. As a next step QA engineer submits new defect to the bug tracking system and marks the test by new defect identifier. The state of all defects that mark tests are also tracked by TGH. If a defect is in the closed state, but the test continues to fall, then the QA engineer will be informed about that.

4.4 OpenMP grammar rules

As a basis for OpenMP grammar we took existing grammar to generate simple C++ tests with basic rules to generate loops and conditions. As it was mentioned above we decided to modify loop generation to use "omp parallel for" pragma statements. To achieve that we made the following changes:

- modified grammar context data to detect if we are inside such parallel loop to be able to prevent generation of loop exit constructions and avoid nested parallelization;
- added required context conditions to all rules that generate constructions that are prohibited under parallel loops [7];
- added rules, that generate loops with "omp parallel for" pragma and form a new context new-ctx for all production rules in parallel block.

We found that usage of local or global variables for array access can cause data races, so for array accesses it was decided to use only variables declared in the parallel loop as well as variables from outside loops. Given example below, it fails due to the loop-carried output dependence on the variable global_var. The global_var is shared among all threads based on OpenMP default shared rule, so there is a data-race condition on the global_var while one thread is reading global_var, another thread might be writing to it.

Listing 1. Data Race example
```
#pragma omp parallel for
for (k = 0; k < 100, k++){
    l = array[k + global_var];
    array[k + global_var] = do_work(l);
}
```

All these means ensure that generated programs for the most part will have correct dynamic semantics without data races in this particular case. We do not require guaranteed absence of data races, because all generated tests will be validated on further phases that will be discussed later.

After adding new generation rules it was also required to add the support of parallel directives into Reduce tool to avoid situations when reduce will accidentally make serial program from parallel.

Listing 2. OpenMP rules
```
omp-for-clause (():_)
  ::= "; /* no lvals variables case */"

omp-for-clause ctx @
  (lv:rv:as:ivs:fs:ret:isl:_) = ctx ? eq(isl, 1)
  ::= "; /* OpenMP cycle skipped due to no nesting */"

omp-for-clause ctx @
  (lv:rv:as:ivs:fs:ret:isl:_) = ctx,
  i = "i", new-lv = (), new-ivs = (i,),
  new-ctx = (new-lv:rv:as:new-ivs:fs:0:1:())
  ? and(le(len(new-ivs), 3), eq(isl, 0))
  ::= *10 { "_Pragma(\"omp parallel for"
           omp-private(new-lv)"\")"
    "for (int " i "=" low-lim(ivs) "; "
    i " <= " big-lim(ivs) "; " step-pos(i)  ") "
    block(new-ctx)}
```

Since we can now allow test generation with race conditions it is a important to explain how we can select tests with correct dynamic semantic among them. For that task we planned to use third-party software for static and dynamic analysis. Unfortunately, we were not able to find a suitable solution based on static analysis. All existent static analyzers have a huge count of false positive messages, high price and require additional changes in generated program. For dynamic analysis we experimented with Helgrind/DRD [4] and Intel Inspector XE [11]. Our experiments with DRD identified that it has major issue with false positive messages that cannot be easily filtered out since exception mechanism that this tool provides is not flexible enough.

Experiments with Intel Inspector XE that also allows dynamic analysis of multi-threaded programs showed an interesting thing: Intel compiler pointer checker mechanism that we utilize for additional program validation cause false positive messages about data races. It was found to be caused by Intel compiler instrumentation for pointer checker that do insecure stack accesses, that was confirmed by simple program with empty omp loop compiled with Intel pointer checker functionality turned on, where Inspector still shows messages about possible data races in empty loop. To avoid this false positive diagnostic, in the TGH we modified validation process for tests from OpenMP generator to do validation with pointer checker and Inspector separately.

5 Conclusion

Extended grammar for checking parallelization and vectorization loop transformations allowed us to find a several new bugs in the parallel and offload features of modern compiler. Formalism of parametric context-free grammars proved that it can be a convenient tool for deterministic test generation even in case of parallel constructs although we had to use additional tools to ensure dynamic correctness of generated code.

In the future we plan to expand a set of parallel constructions that can be generated and increase percentage of correctly generated tests. There are also plans to increase the amount of compiler optimization covered by generated tests by tuning generation weights in rules. We are also experimenting with average size of generated tests to make them more complex and thus more productive.

This research has been supported by the Russian Science Foundation (RSF grant No 18-11-00118).

References

[1] A.P. Stasenko, Parallel programs construction and optimization, **16**, 301-313 (2008)

[2] A.S. Kossatchev, M.A. Posypkin, Programming and Computing Software, **31** No. 1, 10-19 (2005)

[3] F. Bazzichi, I. Spadafora, IEEE transactions on Software Engineering, **SE-8**, 343-353 (1982)

[4] A. Muehlenfeld, F. Wotawa, Informal Proceedings of the International Workshop on Multithreading in Hardware and Software, **06**, (2006)

[5] A.G. Duncan, J.S. Hutchison, In Proc. of the 5th international conference on Software engineering, 170-178 (1981)

[6] K.V. Hanford, IBM Systems Journal, **9**, 242-257 (1970)

[7] B. P. Miller, ACM Letters on Programming Languages and Systems, **1** No.1, 74-88 (1992)

[8] A.S. Kossatchev, M.A. Posypkin, Programming and Computing Software, **31** No. 1, 10-19 (2005)

[9] P.A. Purdom, BIT, **2**, 336-375 (1972)

[10] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: principles, techniques, and tools* (Addison-Wesley Longman Publishing Co., Inc., Boston, 1986) 796

[11] S. Blair-Chappell, A. Stokes, *Parallel Programming with Intel Parallel Studio XE* (John Wiley & Sons, 2012) 217-250