DOI: 10.3745/KIPSTA.2011.18A.3.109

컴파일러에 의한 C레벨 에러 체크

정 지 문[†] · 윤 종 희[††] · 이 종 원[†††] · 백 윤 흥[††††]

요 약

IR(Intermediate Representation) 최적화 과정은 컴파일러 back-end의 중요한 부분으로서 sub-expression elimination, dead code elimination 등 최적화 기법들을 사용한다. 하지만 IR 최적화 단계에서 생기는 에러들을 검출하고 디버깅하는데 많은 어려움이 있다. 그 첫 번째 이유로는 컴파일 된 어셈블리 코드를 해독하여 에러를 체크하기 어렵고 두 번째로는 IR 최적화 단계에서 에러가 생겼는지 결정 짓기 어렵기 때문이다. 이런 이유들로 인하여, 우리는 C 레벨에서 IR 코드변환 무결점 여부를 체크하기 위한 기법들에 관한 연구를 진행하여 왔다. 우리는 MeCC(Memory Comparison-based Clone) 탐색기를 기반으로 하여, 최적화하기 전 IR코드와 최적화 한 후의 IR코드를 각각 C코드로 다시 변환한 뒤, 이 두 개의 C코드를 MeCC의 입력으로 주고, 결과의 일치 여부를 확인하는 방법을 사용한다. 하지만 MeCC가 완벽한 결과를 알려주지 않기 때문에, 우리는 각 IR 최적화 기법마다의 특징에 대한 정보를 사전에 처리해서 그 결과의 정확도를 높였다. 이 논문에서는 dead code elimination, instruction scheduling 및 common sub-expression elimination 등 최적화 기법들을 이용한 변환 코드들을 예시로 실험하여 최종적으로 MeCC에서의 C 레벨 코드의 정확한 에러 체크 동작여부를 보여준다.

키워드 : 컴파일러 최적화, 정확성 증명

Compiler triggered C level error check

Zhiwen Zheng[†] · Jonghee M. Youn[††] · Jongwon Lee[†††] · Yunheung Paek[††††]

ABSTRACT

We describe a technique for automatically proving compiler optimizations sound, meaning that their transformations are always semantics-preserving. As is well known, IR (Intermediate Representation) optimization is an important step in a compiler backend. But unfortunately, it is difficult to detect and debug the IR optimization errors for compiler developers. So, we introduce a C level error check system for detecting the correctness of these IR transformation techniques. In our system, we first create an IR-to-C converter to translate IR to C code before and after each compiler optimization phase, respectively, since our technique is based on the Memory Comparison-based Clone(MeCC) detector which is a tool of detecting semantic equivalency in C level. MeCC accepts only C codes as its input and it uses a path-sensitive semantic-based static analyzer to estimate the memory states at exit point of each procedure, and compares memory states to determine whether the procedures are equal or not. But MeCC cannot guarantee two semantic-equivalency codes always have 100% similarity or two codes with different semantics does not get the result of 100% similarity. To increase the reliability of the results, we describe a technique which comprises how to generate C codes in IR-to-C transformation phase and how to send the optimization information to MeCC to avoid the occurrence of these unexpected problems. Our methodology is illustrated by three familiar optimizations, dead code elimination, instruction scheduling and common sub-expression elimination and our experimental results show that the C level error check system is highly reliable.

Keywords : Compiler Optimization, Correctness Proofs

※ This work was supported by the Korea Science and Engineering Foundation(KOSEF) NRL Program grant funded by the Korea government(MEST) (No. 2010-0018465), the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF) (Grant 2010-0001724), the Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (Grant 2011-0012522).

† 준 회 원 : 서울대학교 전기컴퓨터공학부 석사
†† 정 회 원 : 강릉원주대학교 컴퓨터공학과 강의전담교수(교신저자)
††† 준 회 원 : 서울대학교 전기컴퓨터공학부 박사과정
†††† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
논문접수 : 2011년 2월 16일
수 정 일 : 1차 2011년 4월 6일
심사완료 : 2011년 4월 6

1. Introduction

Compiler is an important part of the software development infrastructure relied upon by programmers. If a compiler is faulty, then all programs compiled with it would have some errors. Unfortunately, it is hard to detect and debug compiler errors by programmers. There are two main reasons why it is difficult. First, it is not easy to inspect the output of the compiler. Problems of the output are often found only by running a compiled program. Second, it is difficult to determine whether errors come from the compiler or the source program that was compiled when problems are detected.

For these reasons, it is very useful to develop tools and techniques that give compiler developers and programmers confidence in their compilers. One way to gain confidence in the correctness of a compiler is to run it on various benchmark programs and check that the optimized version of each program produces correct results on various inputs. While this method can check the correctness of the compiler, it cannot provide any useful information when the error appears, and it is difficult to know which phase of the compiler causes the errors.

To ensure that a compiler works correctly, it should be proven to be sound, which means each compilation phase does not change the semantics of the source program. Optimizations, but sometimes even in complete compilers, have been proven sound by hand [1, 2, 3, 4, 5, 6, 7, 8]. However, it takes a long time and requires a lot of effort to manually prove the soundness of a compiler.

In this paper, we present a new technique for proving the soundness of compiler optimizations in C level. We first create an IR-to-C converter to translate IR(intermediate representation) to C code before and after each compiler optimization phase, respectively, since our technique is based on the Memory Comparison-based Clone(MeCC) detector[9] which is a tool of detecting semantic equivalency in C level. Then we input the two C codes generated by IR-to-C converter into MeCC to detect whether or not they keep semantic equivalence. In this way, we can prove each compiler phase respectively, if there exist some problems in a specific phase, we just need to modify the part of the code.

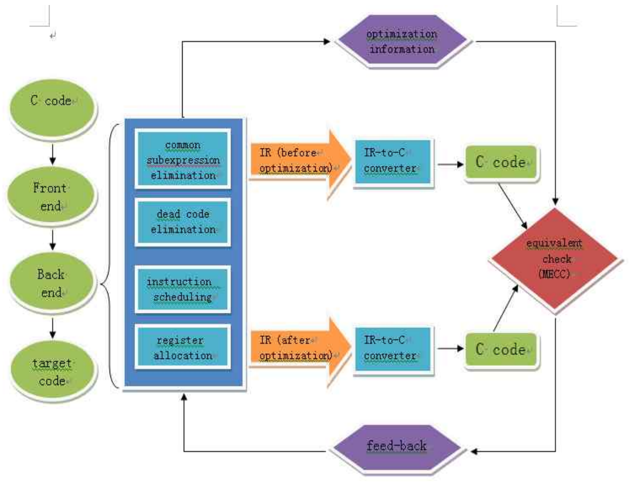
This paper organized as follows. Section 2 presents the architecture of IR optimization error check system. Section 3 presents the technique of how to increase the whole system's reliability. Experimental result about code size and the accuracy of the whole system are presented in section 4, and Section 5 offers our Conclusions.

2. Architecture of the IR Optimization Error Check System

We describe a technique for automatically proving compiler optimizations sound, meaning that their transformations are always semantics-preserving. As is well known, IR (Intermediate Representation) optimization is an important step in a compiler backend. In this step, a compiler can serve several optimization techniques such as common sub-expression elimination, dead code elimination and so on. But unfortunately, the IR optimization errors can be difficult for compiler developers to detect and debug, as briefly mentioned in Section 1.We create a C level error check system to ensure the correctness of these IR transformation techniques. In our system as shown in (Figure 1), we first create an IR-to-C converter to translate IR to C code before and after each compiler optimization phase, respectively, since our technique is based on the Memory Comparison-based Clone(MeCC) detector which is a tool of detecting semantic equivalency in C level. Our IR-to-C converter is based on a compiler named SoarGen[10] which is a re-targetable compiler platform we have invented. Then we input the two C codes generated by IR-to-C converter to MeCC to detect whether they keep semantic equivalence or not.

MeCC accepts only C codes as its input and it uses a path-sensitive semantic-based static analyzer to estimate the memory states at exit point of each procedure, and compares memory states to determine whether the procedures are semantically equivalent or not. Although it can effectively detect semantic clones, sometimes the result of MeCC cannot get 100% similarity. To increase the credibility of the results, we describe a technique which comprises how to generate C codes in IR-to-C transformation phase and how to send the optimization information to MeCC to avoid the occurrence of these unexpected problems.

And with the optimization information, MeCC can get 100% similarity when the two C codes are semantically equivalent. In this way, we avoid some unexpected problems and increase the similarity rate of the result. So when the result is 100%, it means the C codes are semantically equivalent, and also means the IR codes before and after optimizations are semantically equivalent. When the result is not 100%, we can say that the two C codes are not semantically equivalent, and the IR optimization phase runs incorrectly. So in this way, we can find which phase generates errors.



(Figure 1) Infrastructure for IR optimization error check system

3. How to Increase the Reliability

The MeCC uses a path-sensitive semantic-based static analyzer to estimate the memory states at exit point of each procedure, and compares memory states to determine whether the procedures are semantically equivalent or not. Since the abstract memory states have a collection of the memory effects along the execution paths within procedures, it can effectively detect semantic clones. However, It is not perfect since the disadvantage of MeCC is that it may cause some unexpected problems. For instance, MeCC cannot guarantee two semantic-equivalency codes always have 100% similarity or two codes with different semantics cannot get the result of 100% similarity. MeCC only compares the memory state at exit point of procedure, it does not care whether the intermediate phases keep the same semantic or not, so it may still get 100% similarity ratio even if the two codes are not semantically equivalent. At the exit point of the procedure, if the number of memory entries is different from each other, it will not get 100% similarity ratio even if the two codes are semantically equivalent. We will illustrate these problems by three familiar optimizations such as dead code elimination, instruction scheduling and common sub-expression elimination.

3.1 Dead Code Elimination

There are 3 different cases after dead code elimination.

The first case: definition – definition

As shown in (Figure 2), the variable VASM_R[140] is defined at the first line, and then defined again at the second line without any use. In this case, it does not affect the result to eliminate the previous definition of VASM_R[140], and the memory state of the variable is still equal at the exit point of the procedure. Therefore the optimized version will not cause different abstract memory state.

```
VASM_R[140] = VASM_R[139] + 1;
VASM_R[140] = VASM_R[139] + VASM_R[121];
VASM_R[141] = VASM_R[140] + 4;
```
Original C code

```
VASM_R[140] = VASM_R[139] + VASM_R[121];
VASM_R[141] = VASM_R[140] + 4;
```
Optimized C code

(Figure 2) Dead code elimination (1)

The second case : definition – use – definition

As shown in (Figure 3), if the compiler eliminates a dead code incorrectly, it will affect the other variables, and the value of some other variables will be changed, which will cause some of the memory states not to keep equal at the exit of the procedure. Therefore MeCC will not get 100% similarity, and it shows that the dead code elimination algorithm is performed incorrectly.

```
VASM_R[141] = VASM_R[140] + VASM_R[122];
VASM_R[140] = VASM_R[139] + VASM_R[121];
VASM_R[142] = VASM_R[140] + 4;
VASM_R[140] = VASM_R[139] + 1;
```
Original C code

```
VASM_R[141] = VASM_R[140] + VASM_R[122];
VASM_R[142] = VASM_R[140] + 4;
VASM_R[140] = VASM_R[139] + 1;
```
Optimized C code

(Figure 3) Dead code elimination (2)

The third case: definition – no use

```
VASM_R[141] = VASM_R[140] + VASM_R[122];
// VASM_R[141] will not use until the end of the procedure
VASM_R[142] = VASM_R[140] + 4;
VASM_R[143] = VASM_R[142] + 1;
```
Original C code

```
VASM_R[142] = VASM_R[140] + 4;
VASM_R[143] = VASM_R[142] + 1;
```
Optimized C code

(Figure 4) Dead code elimination (3)

As shown in (Figure 4), the variable VASM_R[141] is defined at the first line, but it will not be used until the end of the procedure, so the first line is a dead code. But when the two C codes are input to MeCC, the result will not get 100% similarity since the memory state of the variable VASM_R[141] does not keep equality.

To avoid this case, we introduce a new variable, which is assigned to the variable whose later definition will be eliminated, and insert an assignment instruction just before the eliminated code as shown in (Figure 5). Finally, the information of the eliminated variable is sent to MeCC in order to ignore the difference of the memory state of the variables during the memory comparison algorithm and to get higher similarity.

```
temp = VASM_R[141];
VASM_R[141] = VASM_R[140] + VASM_R[122];
// VASM_R[141] will not use until the end of the procedure
VASM_R[142] = VASM_R[140] + 4;
VASM_R[143] = VASM_R[142] + 1;
```

Original C code

```
temp = VASM_R[141];
VASM_R[142] = VASM_R[140] + 4;
VASM_R[143] = VASM_R[142] + 1;
```

Optimized C code

(Figure 5) Dead code elimination (4)

3.2 Common Sub-expression Elimination

As shown in (Figure 6), the code is optimized correctly but the optimized code may have some more temporary variables than original code, so in this case, it causes some unexpected problems in MeCC phase. Since the abstract memory states have a collection of all the memory effects along the execution paths within procedures, the abstract memory state will certainly collect the temporary variable generated by optimization phases. So the optimized code will have some more temporary variable states in the abstract memory state. These abstract memory states of the optimized code cannot match with the one of the original code, so the result will not get 100% similarity ratio even if the two codes are semantically equivalent.

```
VASM_R[139] = VASM_M[VASM_FPR + (-24)];
VASM_R[140] = VASM_R[139] + VASM_R[138] + 1;
VASM_R[141] = VASM_R[139] + VASM_R[138] + VASM_R[121];
```

Original C code

```
VASM_R[139] = VASM_M[VASM_FPR + (-24)];
TEMP = VASM_R[139] + VASM_R[138];
VASM_R[140] = TEMP + 1;
VASM_R[141] = TEMP + VASM_R[121];
```

Optimized C code

(Figure 6) Common sub-expression elimination

To avoid this case, we need to send the variable information to MeCC so that it can ignore the difference of the memory state of the variable during the memory comparison algorithm and get higher similarity. Certainly the temporary variables generated by optimization phase keep signatures so that we can guarantee that the name of the temporary variable is unique in the whole procedure.

3.3 Instruction Scheduling

After instruction scheduling, if an optimized phase does not break the define-use (DU) chain of the procedure, it means the optimization keep semantically equivalent, also the memory state of each variables at the exit point of the procedure keep the same, and the result will get 100% similarity. If an optimized phase breaks the DU chain of the procedure, it will change some variable states, and the result of MeCC will not get 100% similarity. But in some special case, although the code motion is incorrect, MeCC will still get 100% similarity.

(Figure 7) show an example of an incorrect optimization. The variable VASM_R[140] is used in line 2, and its definition is in line 1. After instruction scheduling, the DU chain of variable VASM[140] is broken, but the memory state at the exit point of the procedure is not changed since the instruction “printf” is not memory effect instruction. Therefore it causes the result of 100% similarity even if the code motion is incorrect.

```
VASM_R[140] = VASM_R[139] + 1;
printf("%d",VASM_R[140]);
VASM_R[140] = VASM_R[139] + VASM_R[121];
```

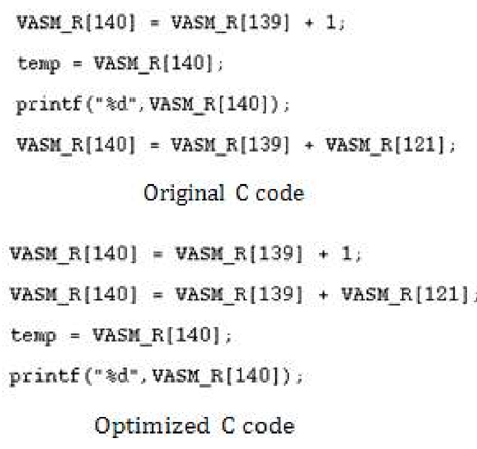
Original C code

```
VASM_R[140] = VASM_R[139] + 1;
VASM_R[140] = VASM_R[139] + VASM_R[121];
printf("%d",VASM_R[140]);
```

Optimized C code

(Figure 7) Instruction scheduling (1)

To avoid this case, we introduce new variables, which are assigned to all the function parameters. Certainly we first need to guarantee the new variables are unique in the whole procedure as shown in (Figure 8).



```
VASM_R[140] = VASM_R[139] + 1;
temp = VASM_R[140];
printf("%d",VASM_R[140]);
VASM_R[140] = VASM_R[139] + VASM_R[121];
```

Original C code

```
VASM_R[140] = VASM_R[139] + 1;
VASM_R[140] = VASM_R[139] + VASM_R[121];
temp = VASM_R[140];
printf("%d",VASM_R[140]);
```

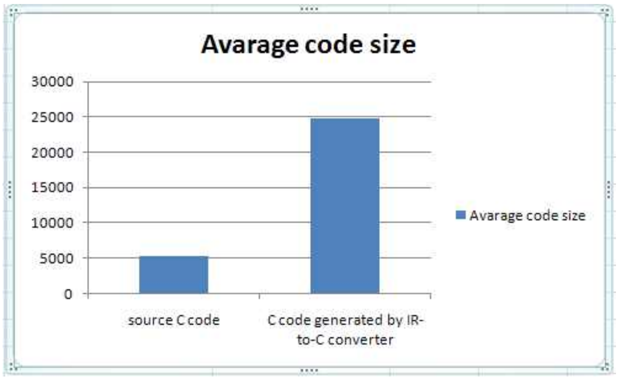
Optimized C code

(Figure 8) Instruction scheduling (2)

We use these new variables to check whether the function calls are semantically equivalent or not. If the memory states of the variables are equal, it means all the function parameters keep the same, it also means the semantic of these function calls are equivalent. In these ways, we avoid the occurrence of these unexpected problems to increase the reliability of whole system.

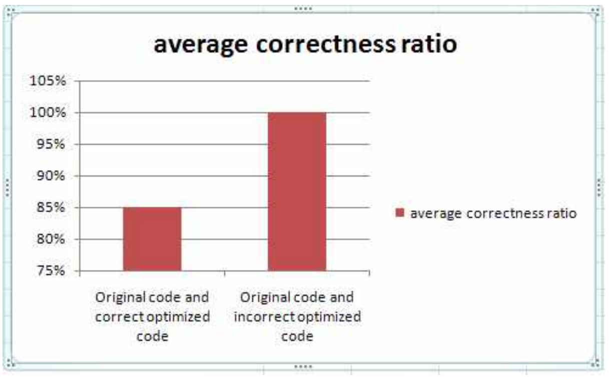
4. Experimental Results

In order to test the accuracy of our IR optimization error check system, our IR-to-C translator coverts IR to C code at each optimization phases in the compiler, then compares the two generated C codes before and after IR optimization phase. To guarantee the correctness of the IR optimization, we optimized it manually. We test the accuracy ratio of IR optimization error check system in 20 small-scale open source projects, and in IR optimization phase, we manually optimize them correctly and incorrectly. Then we compare the original code with the incorrect optimized code and correct optimized code respectively. As shown in (Figure 9), the average code size of the 20 small-scale benchmarks is 5226 lines, but the average code size of the C code generated by IR-to-C converter is 24785 lines, it is almost 5 times of the source code. The generated C code contains the lowered C code than the original one because IR code does not keep the original code structures. But it does not affect the quality of final output code (assembly code) because the converted C code is only used to evaluate the correctness of each optimization techniques in the compiler.



(Figure 9) The average code size of 20 small-scale projects

In (Figure 10), all the 20 comparison results of original codes and incorrect optimized codes show that the average accuracy ratio is 100%. But the comparison result of the correct optimized code is not 100% such that three of them didn't get 100% similarity, since MeCC produced unknown value memory state in these three cases. The average correctness ratio is 85%. This result means our approach can detect the incorrect optimized code accurately, but ours reports inaccurate result for the correct optimized code in a few case. We will take care of it in our future work.



(Figure 10) Average correctness ratio of the IR optimization error check system

5. Conclusion

In this paper, we describe a technique for automatically proving compiler optimizations sound, meaning that their transformations are always semantics-preserving. We create an IR-to-C converter to translate IR to C code before and after each compiler optimization phase, respectively, then input the two C codes generated by IR-to-C converter to MeCC to detect whether they keep semantic equivalence or not. Finally, for increasing the reliability of the whole system, we present some techniques which comprise how to generate C codes in IR-to-C transformation phase and how to send the optimization information to MeCC to avoid the occurrence of these unexpected problems. Through experiment results, the accuracy of the whole system is about 92.5%.

Although we have increased some accuracy of the system, it is still insufficient because of the occurrence of the unknown values. In the future, we will modify MeCC to reduce the occurrence of the unknown value memory states and improve the accuracy of the static analysis.

Reference

[1] A. Pnueli, M. Siegel, and E. Singerman, Translation validation, In Tools and Algorithms for Construction and Analysis of Systems, TACAS '98, volume 1384 of Lecture Notes in Computer Science, pages151 - 166, 1998.

[2] George C. Necula, Translation validation for an optimizing compiler, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 83 - 95, Vancouver, Canada, June, 2000.

[3] Martin Rinard, Credible compilation, Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, March, 1999.

[4] J. Guttman, J. Ramsdell, and M. Wand, VLISP: a verified implementation of Scheme, Lisp and Symbolic Compucation, 8(1-2):33 - 110, 1995.

[5] F. Lockwood Morris, Advice on structuring compilers and proving them correct, In Conference Record of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston MA, January, 1973.

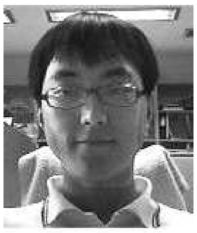
[6] Jens Knoop, Oliver R¨uthing, and Bernhard Steffen, Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Systems, 16(4):1117 - 1155, July, 1994.

[7] M. Kauffmann and R.S. Boyer, The Boyer-Moore theorem prover and its interactive enhancement, Computers and Mathematics with Applications, 29(2):27 - 62, 1995.

[8] Bernhard Steffen, Generating dataflow analysis algorithms for model specifications, Science of Computer Programming, 21(2):115 - 139, 1993.

[9] Heejung Kim, Yungbum Jung, Sunghun Kim, Kwangkeun Yi, MeCC: Memory Comparison-based Clone Detector, ICSE 2011: The 33rd International Conference on Software Engineering, Waikiki, Honolulu, Hawaii, May 21 ~ 28, 2011.

[10] M. Ahn, SoarGen: A user retargetable compiler in the design of embedded systems, Ph.D thesis, Seoul National University, 2009.



정 지 문

e-mail : jmjung@optimizer.snu.ac.kr
2007년 중국 Jilin University 소프트웨어 공학과(학사)
2011년 서울대학교 전기컴퓨터공학부 (석사)
관심분야 : Embedded systems, optimizing compiler, software optimizations and computer architecture.

윤 종 희

e-mail : jhyoun@gwnu.ac.kr
2003년 경북대학교 전자전기공학부(학사)
2011년 서울대학교 전기컴퓨터공학부 (박사)
2011년 ~ 현 재 강릉원주대학교 컴퓨터 공학과 강의전담교수
관심분야 : Embedded systems, Optimizing compiler, software optimizations and computer architecture, MPSoC.

이 종 원

e-mail : jwlee@optimizer.snu.ac.kr
2007년 서울대학교 전기공학부(학사)
2007년 ~ 현 재 서울대학교 전기컴퓨터 공학부 박사과정
관심분야 : embedded systems, optimizing compiler, software optimizations and computer architecture.

백 윤 흥

e-mail : ypaek@snu.ac.kr
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산과(박사)
1997년 ~ 1999년 NJIT 조교수
1999년 ~ 2003년 KAIST 전자전산학과 부교수
2003년 ~ 현 재 서울대학교 전기컴퓨터공학부 교수
관심분야 : 임베디드 소프트웨어, 임베디스 시스템 개발도구, 컴파일러, MPSoC