컴파일러 최적화 코드 분석 연구 조사

한상훈[1], 강정환[2], 권동현[3*]
[1]부산대학교 정보융합공학과 석사과정
[2]부산대학교 정보융합공학과 석박사통합과정
[3]부산대학교 정보컴퓨터공학부 교수
sanghun@pusan.ac.kr, jeonghwan@pusan.ac.kr, kwondh@pusan.ac.kr

A Study on Research in Analysis of Code Generated by Compiler Optimization

Sang-Hun Han[1], Jeong-Hwan Kang[2], Dong-Hyun Kwon[3]
[1]Dept. of Information Convergence Engineering, Pusan National University
[2]Dept. of Information Convergence Engineering, Pusan National University
[3]Dept. of Computer Science and Engineering, Pusan National University

요 약

컴파일러는 사용자가 작성한 소스 코드로부터 타겟 머신에서 동작하는 코드로 변환하기 위해 사용되는 프로그램으로 컴파일러의 구현은 몇 가지 단계로 구성된다. 그 단계 중 하나에 속하는 최적화 단계는 사용자에 의해 작성된 코드를 실행 시간, 메모리 사용량, 에너지 소모 등을 줄이기 위해 코드를 효율적으로 사용하고자 적용되는 단계이다. 그렇지만 이러한 컴파일러 최적화 기법은 사용자가 의도하지 않은 결과를 만들어 프로그램의 보안성을 낮추기도 한다. 이에 본 논문에서는 관련 연구 내용을 살펴보고 정리하고자 한다.

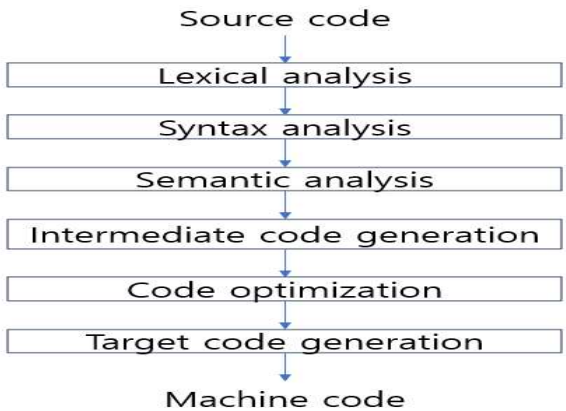
1. 서론

1952년 세계 최초의 컴파일러가 개발되고, GCC와 LLVM 등 많은 컴파일러 기술이 개발되었다. 컴파일러는 그 목적에 따라 어휘 분석, 구문 분석, 의미 분석, 중간 코드 생성, 코드 최적화, 타겟 코드 생성의 단계로 구성된다. [1] 그 중 코드 최적화 단계는 실행되는 코드를 최적화하는 단계로 그 예로 핍홀 최적화, 지역 최적화 등의 기법들이 있다. [5] 이렇듯 컴파일러 최적화 기법은 사용자가 작성한 코드의 효율성을 높여 실행 시간, 메모리 사용량, 에너지 소모 등을 줄이기 위한 목표로 컴파일러 개발 단계에 포함되어진다. 실제로 컴파일러와 관련해 많은 최적화 기법이 개발되었으며 GCC의 경우 200가지, LLVM의 경우 100가지 이상의 최적화 기법이 존재한다고 한다. [4] 그렇지만 몇 가지 연구에 의하면 이러한 최적화 기법들 중 일부는 최적화를 통과한 후 사용자가 의도하지 않은 코드가 생성되어 프로그램의 보안성을 낮추기도 한다. [2][3] 이에 본 논문은 관련된 연구 내용을 살펴보고 정리하고자 한다.

* Corresponding Author

2. 컴파일러

컴파일러는 사용자가 작성한 소스 코드로부터 타겟 머신에서 동작하는 코드로 변환하기 위해 사용되는 프로그램이다. 이러한 컴파일러는 그림1과 같이 각 단계별로 특정 기능을 수행하는 모듈들로 구성되어져있다. [1]



(그림1) 컴파일러 단계

각 모듈이 수행하는 기능은 다음과 같다. Lexical analysis은 사용자가 입력한 소스 코드를 토큰으로 분류하는 단계이다. Syntax analysis은 전 단계로부

터의 토큰 스트림으로부터 파스 트리를 생성하는 단계이다. Semantic analysis은 타입 검사 등을 수행하는 단계로 컴파일러 구현 목적에 따라 생략되기도 한다. Intermediate code generation은 전 단계로부터의 파스 트리로 부터 목적 코드를 생성하기 전에 중간 코드를 생성하는 단계이다. Code optimization은 코드에 대해 최적화를 수행하는 단계이다. Target code generation은 최적화를 거친 이후 머신 코드를 생성하는 단계이다.

3. 컴파일러 최적화 기법

컴파일러 최적화 기법에는 여러 기법이 있지만 4와 관련해 3.1, 3.2, 3.3, 3.4의 최적화 기법에 대해 살펴본다.

3.1 공통 하위 표현식 제거

명령어의 공통된 부분이 여러 번 나타났을 때 공통된 부분의 연산을 한 번만 수행하고 그 결과를 활용하는 기법이다.

3.2 강도 감소

기존 명령어의 연산 수행에 필요한 클럭 수보다 작은 명령어의 연산 수행으로 대체될 수 있다면 대체하는 기법이다.

3.3 Dead Store Elimination

변수가 할당된 후 다음 명령문들에서 해당 변수가 읽혀지지 않는다면 해당 변수를 인자로 받거나 관련된 연산을 수행하지 않는 기법이다.

3.4 Function Call Inlining

어떤 함수 A(Callee)와 이에 대한 Caller 함수가 있을 때 Callee에 대한 스택 프레임을 Caller 함수의 스택 프레임에 둠으로써 Callee 함수의 프롤로그와 에필로그 실행을 하지 않도록 하는 기법이다. 이에 대해 Callee 함수가 안전하고, Caller 함수의 스택 프레임이 불안전하다고 가정할 때 Callee의 스택 프레임이 Caller 함수의 스택 프레임에 존재하게 된다는 점이 있다. [2]

Example	Before	After
Common subexpression elimination	A = a+b+3 B = a+b+x C = a+b+y	T = a+b A = T+3 B = T+x C = T+y
Strength reduction	A = a * 2	A = a + a
Dead Store Elimination	1. 패스워드를 읽고 해시 값을 계산 2. 패스워드 메모리 초기화 연산을 수행 3. 패스워드에 대한 연산 수행이 없음	1. 패스워드를 읽고 해시 값을 계산 2. 패스워드 메모리 초기화 연산 수행(remove) 3. 패스워드에 대한 연산 수행이 없음

(표1) 컴파일러 최적화 예제

4. Compiler-Introduced Security Bug

compiler-introduced security bug, CISB는 컴파일러 최적화에 의해 만들어진 코드가 의미적으로는 작성한 소스 코드의 의도대로 만들어지나 보안과 관련된 행동을 하는 코드에서는 그렇지 않은 버그를 말한다. [2]

CISB의 예로 Dead Store Elimination, DSE이 있다. 표1의 DSE 예제에서 사용자가 패스워드를 입력받아 해시값을 계산하여 패스워드 점검을 수행하는 상황을 고려해보자. 이에 패스워드를 입력받고 해시값을 계산한 후 패스워드를 초기화하고자 한다. 그렇지만 이후 패스워드와 관련된 연산이 존재하지 않기에 컴파일러는 패스워드를 초기화하는 연산을 제거하고 패스워드는 메모리 상에 존재하게 된다.

이에 프로그램의 동작은 사용자가 작성한 소스 코드에서 수행 결과가 의미적으로는 동등하지만 패스워드를 초기화하고자 하는 연산은 수행되지 않게 되어 패스워드에 대한 정보 누출이 발생할 수 있다.

또 다른 예로 강도 감소를 살펴보자. 강도 감소를 통해 기존 연산은 비용이 낮은 연산으로 대체될 수 있다. 사용자가 입력받는 키 값에 따라 특정 연산을 수행하자고 하자. 연산은 동일하고 피연산자가 다르며 타이밍 공격을 고려하여 소스 코드를 작성했다고 했을 때 강도 감소로 인해 특정 부분의 연산이 보다 낮은 비용의 연산으로 대체되어지면 타이밍 공격이 발생할 가능성이 있게 된다. [3]

공통 하위 표현식 제거 기법 역시 강도 감소와 마찬가지의 경우가 발생할 가능성이 있다.

이처럼 컴파일러 최적화를 통해 나온 코드는 실제 의도와는 다르게 동작할 수 있다. 이러한 컴파일러 최적화에 의해 만들어지는 코드는 기존에 사용자가

고려했던 보안 요소에 대한 코드를 만들지 못하게 된다.

이를 해결하기 위해 컴파일러 최적화를 수행하지 않을 수도 있지만 실행 오버헤드가 커서 비효율적이다는 측면이 있다.[5]

4. 결론

이처럼 사용자가 작성한 소스 코드는 컴파일러 최적화를 통해 의도하지 않았던 코드로 바뀔 수 있다. 이에 사용자는 컴파일러 최적화를 활용할 때 이러한 버그를 고려하여 코드를 작성해야 함을 말해준다.

이러한 버그가 수정될 때까지 걸리는 시간은 평균적으로 GCC의 경우 11.16개월, LLVM의 경우 13.55개월로 많은 시간이 소요된다고 한다. [4]

이러한 문제는 CISB에 대한 연구가 추가적으로 필요하고, 컴파일러 최적화 기법이 보완되어야 할 점이 있음을 시사한다.

Acknowledgement

"본 연구는 과학기술정보통신부 및 정보통신기획평가원의 대학ICT연구센터사업의 연구결과로 수행되었음" (IITP-2023-RS-2023-00259967)

참고문헌

[1] Enyindah P., Okon E. Uko, The New Trends in Compiler Analysis and Optimizations, International Journal of Computer Trends and Technology (IJCTT), 46, 2017, 95-99

[2] Jianhao Xu, Kangjie Ju, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, Bing Mao, Silent Bugs Matter:
A Study of Compiler-Introduced Security Bugs, USENIX, Anaheim, CA, USA, 2023, p. 3655-3672

[3] Vijay D'Silva, Mathias Payer, Dawn Song, The Correctness-Security Gap in Compiler Optimization, IEEE CS Security and Privacy Workshops, San Jose, California, 2015, p. 73-87.

[4] Zhide Zhou, Zhilei Ren, Guojun Gao, He Jiang, An empirical study of optimization bugs in GCC and LLVM, Journal of Systems and Software, 174, 110884, 2021

[5] 박두순, 컴파일러의 이해(내공 있는 프로그래머로 길러주는), 서울, 한빛아카데미, 2020