논문 22-2-04

컨텍스트를 구분하지 않는 분석이 오히려 느린 이유와 속도를 높이는 방법 [1]

Why context-insensitivity is costly and how to reduce its cost

오학주 · 이광근
서울대학교 컴퓨터공학부 프로그래밍 연구실
{pronto; kwang}@ropas.snu.ac.kr

요 약

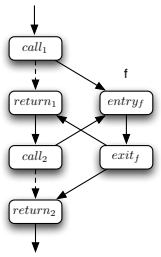
이 논문은 컨텍스트를 구분하지 않는 분석이 낮은 정확도에도 불구하고 실제로는 빠르지 않은 이유를 설명하고 그 비용을 줄이는 방법을 제시한다. 우리는 컨텍스트를 구분하지 않는 분석에서 발생하는 거짓실행경로들로 인해서 전체 프로그램에 걸쳐서 커다란 고리가 형성되고, 이로 인해 분석비용의 대부분이 큰 고리의 고정점을 계산하느라 낭비되고 있음을 관찰하였다. 이에 대한 해결책으로 어느함수가 분석되기 시작할 때, 분석을 마치고 리턴할 지점을 따로 기억함으로써 거짓고리로 인해 발생하는 불필요한 계산을 줄이는 알고리즘을 소개하고 오픈소스 프로그램에 대한 성능향상 실험결과를 보고한다. 상용 프로그램 분석기인 Sparrow에 적용하여 실험한 결과 알고리즘을 적용하기 전에 비해서 평균 다섯배의 분석속도 향상이 있었다.

1 도입

정확도가 낮은 분석이 오히려 계산량이 더 많을 수 있음은 알려져 있지만 [4, 11] 그에대한 정확한 이유나 해결책은 알려져 있지 않다. 예를 들어, Martin [4]은 컨텍스트를 구분하지 않는(context-insensitive) 분석이 구분하는(context-sensitive) 분석보다 항상 빠르지는 않음을 보고하였고, Rival [11] 은 실행경로(trace)를 구분하지 않을 경우, 실행경로를 구분하는 경우보다 계산량이 더 많을 수 있음을 보고하였다. 우리도 Sparrow[12, 13, 14, 15] 를 통해서 실제프로그램들을 분석하면서 비슷한 경험을 하였는데, 컨텍스트를 구분하지 않는 분석을 적용하더라도 글로벌 분석은 프로그램의 크기가 커짐에 따라 비용이 급속히 증가하였다. 일반적으로 이러한 현상은 분석의 정확도가 낮을수록 실제 프로그램의 실행과는 관계없는 거짓계산(spurious computation)이 많아지기 때문이라 할수있다. 하지만 이와 관련한 구체적인 설명이나 이를 해결하여 성능을 올릴수 있는지 여부등은 알려져 있지 않다.

이 논문은 함수호출을 부정확하게 분석할 경우에 많은 거짓계산이 왜 발생하는지를 설명하고, 워크리스트 기반의 고정점 계산 알고리즘을 수정하여 대부분의 거짓계산을 제거함으로써 분석의 성능을 획기적으로 개선할 수 있음을 보인다.

[1]본 연구는 교육과학기술부/한국과학재단 우수연구센터 육성사업의 지원(과제번호:R11-2008-007-01002-0)과 교육인적자원부 두뇌한국21사업의 지원으로 수행하였다.



$$
\begin{aligned}
x_{entry_f} &= \mathcal{F}_{call}(x_{call_1}) \sqcup \mathcal{F}_{call}(x_{call_2}) && \cdots (1)\\
x_{exit_f} &= \mathcal{F}_{body_f}(x_{entry_f}) && \cdots (2)\\
x_{return_1} &= \mathcal{F}_{return}(x_{exit_f}, x_{call_1}) && \cdots (3)\\
x_{call_2} &= x_{return_1} && \cdots (4)\\
x_{return_2} &= \mathcal{F}_{return}(x_{exit_f}, x_{call_2}) && \cdots (5)
\end{aligned}
$$

[그림 1] 함수호출을 부정확하게 요약하는 경우 $entry_f \rightarrow exit_f \rightarrow return_1 \rightarrow call_2 \rightarrow entry_f \rightarrow \cdots$ 와 같은 거짓고리가 생겨나게 된다.

1.1 문제: 컨텍스트를 구분하지 않는 분석이 비싼 이유

함수호출을 부정확하게 분석하면 거짓고리들이 생겨난다. 그림 1은 연속된 두 함수호출을 컨텍스트를 무시한채 요약한 연립방정식의 예이다. 방정식 (1)은 함수 f를 호출할 때 두 함수호출지점의 입력을 모음을 의미하며, 방정식 (3)과 (5)는 두 리턴지점에 같은 값(x_{exit})이 흘러감을 의미한다. 때문에 연립방정식을 푸는 과정이 고리 (4) → (1) → (2) → (3) → (4) → (1) → ⋯ 을 따라 진행되며, 이의 고정점을 구해야 한다. 하지만 이 고리는 프로그램의 실제 실행과는 상관없기 때문에 실행의미를 계산하는데는 불필요하다.

우리는 이러한 고리들이 분석비용을 크게 증가시킴을 발견했다. 그 이유는 이러한 고리들이 프로그램 각 부분에서 발생하며 이들이 서로 합쳐져서 커다란 하나의 고리를 형성하기 때문이다. 이런 이유로, 오픈소스프로그램들을 이용하여 실험한 결과 전체 프로그램의 80-90%가 하나의 고리에 포함되었다. 그 결과, 이들 프로그램을 분석하는 것은 큰 거짓고리를 분석하는 것과 거의 같아지며, 이 과정에서 반복적인 계산이 일어나게 된다.

1.2 해결책: 거짓계산을 줄이는 알고리즘

도출된 분석방정식을 변경하지 않으면서, 고정점 계산 알고리즘을 수정함으로써 이러한 거짓계산들을 상당부분 제거할 수 있다. 그림 1의 방정식을 푸는 과정중 일부인 ⋯ → (4) → (1) → (2) → ⋯을 생각하자. 다음의 할 일은 방정식 (3)과 (5)를 계산하는 것인데, 이 때 (3)을 계산하는 것은 거짓계산이다. 왜냐하면 방정식 (4)는 함수호출지점 $call_2$을 의미하고 프로그램의 실제 실행은 $return_1$로 가지 않기 때문이다. 따라서 우리는 이 상황에서 (5)만을 분석하고 (3)으로의 계산은 생략할 수 있다.

고정점 계산중에 거짓계산을 탐지하는 방법은 함수호출분석을 시작할때마다 그 지점에 대응하는 리턴지점을 각 함수마다 기억하는 것이다. 그 후 함수가 리턴할 때 기억된 리턴지점으로만 리턴하고, 나머지 지점으로의 계산은 생략한다. 각 함수마다 리턴정보를 올바르게 기억하기 위해서 각 함수의 분석은 독점적으로(mutually exclusive) 이루어지도록 조절한다. 즉, 하나의 함수가 분석중인 경우에 다른 함수의 분석을 시작하지 않는다. 분석의 안전성(soundness)을 위해서 재귀함수가 아닌 경우에만 이 방법을 적용한다.

제안된 알고리즘의 결과는 방정식의 고정점은 아니지만 프로그램의 실행의미에 대해서는 안전한 요약결과이다. 방정식의 고정점을 계산하는 과정에서 일부 계산을 생략하기 때문에 계산결과가 고정점이 아닐 수 있다. 하지만 우리가 제거하는 계산과정들은 모두 프로그램의 실제 실행과는 관련없는, 요약으로 인해 부차적으로 생겨나는 실행과 관련있는 것들이기 때문에 이들과 관련된 계산을 생략한다 하더라도 실제 실행의미를 포섭하는 결과를 얻게된다.

상용화된 프로그램 분석기인 Sparrow[12, 13, 14, 15] 에 적용하여 오픈소스프로그램들을

프로그램	#고리에속한함수/#전체함수	#고리에속한지점/#전체프로그램지점
spell-1.0	24/31(77%)	751/782(95%)
gzip-1.2.4a	100/135(74%)	5,988/6,271(95%)
sed-4.0.8	230/294(78%)	14,559/14,976(97%)
tar-1.13	205/222(92%)	10,194/10,800(94%)
bison-1.875	410/832(49%)	12,558/18,110(69%)
proftpd-1.3.1	940/1,096(85%)	35,386/41,062(86%)
apache-2.2.2	1,364/2,075(66%)	71,719/95,179(75%)

[표 I]거짓고리가 포함하는 프로그램 지점들과 함수개수의 비율. 프로그램의 대부분의 저점들과 함수들이 하나의 고리에 포함됨을 보여준다.

대상으로 실험한 결과 평균 84.17%의 속도향상을 얻었다. 우리는 컨텍스트를 완전히 구분하지 않는(context-insensitive) 분석의 성능향상에 대해서 실험하였지만, 제안된 알고리즘은 일반적으로 컨텍스트를 어느정도 구분하는(partially context-sensitive) 분석의 경우에도 적용가능하다. 또한 기존 분석의 정확도를 떨어뜨리지 않는다.

1.3 관련연구

프로그램 분석에서 잘못된 함수호출경로 (invalid paths)를 피하는 연구는 많지만 [1, 3, 7], 이들은 모두 더 정교한 분석방정식을 세우는 방법에 대한 것이지 방정식을 푸는 과정에서 발생하는 거짓계산을 줄여주지는 않는다. 컨텍스트를 구분하는(context-sensitive) 분석은 구분하지 않는(context-insensitive) 분석에 비해서 더 정교한, 즉 요약을 덜 한, 방정식을 세우고 푸는 방법으로써, 분석의 정확도를 높이는데 목적이 있다. 우리가 제안하는 방법은 방정식을 풀 때 발생하는 의미없는 거짓계산들을 제거하여 분석의 효율을 높이는데 목적이 있다.

기존의 프로그램 분석기들의 경우에 거짓고리 때문에 발생하는 분석효율의 저하를 구체적으로 설명한 사례가 없다 [1, 8, 10, 9, 6]. 우리 알고리즘은 수퍼그래프(supergraph) [3]로 표현된 프로그램에 대해서 고정점계산을 하는 분석기들 [6, 10, 8, 9]에 직접 적용가능하다.

2 거짓고리로 인한 성능저하 문제

거짓고리가 분석 효율을 떨어뜨리는 이유는 1) 실제 프로그램에 나타나는 고리의 크기가 매우 크며(전체프로그램 크기의 80-90%) 2) 고리가 클수록 고정점을 계산하기 위해서는 더욱 많은 반복적인 계산이 필요하기 때문이다.

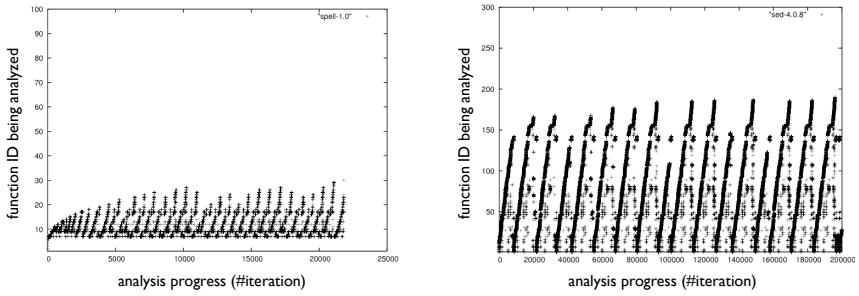
2.1 실제 C 프로그램에 나타나는 거짓고리의 크기

거짓고리들은 크게 다음과 같은 두 가지 이유로 서로 합쳐져서 크기가 커지게 된다.

(1) 함수호출체인의 길이가 길수록 고리의 크기가 커진다. 다음과 같은 함수호출체인을 생각하자.

$$F_0 \rightarrow_{t_0} F_1 \rightarrow_{t_1} \cdots \rightarrow_{t_{n-1}} F_n$$

$F_i \rightarrow_{t_i} F_{i+1}$ 은 함수 F_i 가 F_{i+1} 를 t_i 개의 서로다른 지점에서 호출함을 의미한다. $j(< n)$ 를 $F_j \rightarrow_{t_j} F_{j+1}$ and $t_j \geq 2$ 를 만족하는 가장작은 인덱스라고 하면 고리는 j번째 함수호출에서 생긴다. 그리고 이 고리는 함수 $F_{j+1}, \cdots, F_n$ 를 모두 포함하게 된다. 때문에 고리가 함수호출체인의 윗단계에서 생성될수록 고리의 크기가 커지는 것이다. C에서는 같은 함수를 여러번 호출하는 것이 흔하기 때문에 체인의 윗단계에서부터 고



spell-1.0 (2,213 LOC, > 30 repetitions)

sed-4.0.8 (26,807 LOC, > 150 repetitions)

[그림 2] 분석의 자취. 분석이 진행될수록 비슷한 패턴의 작업이 계속 반복되고 있다. 이는 함수호출경로를 구분하지 않는데서 발생한 거짓고리에 의한 계산량 증가에 기인한다.

리가 생길 가능성이 크다. 이렇게 생성된 고리는 콜스트링을 이용하여 컨텍스트를 구분하는 분석을 이용해도 제거하기 어렵다. 왜냐하면, 콜스트링의 길이가 1인 경우를 생각하면, 컨텍스트의 구분이 체인의 아랫단계부터 이루어지기 때문이다. 따라서 가장 윗단계에서 생성된 고리를 콜스트링을 이용하여 제거하려면 길이 n의 콜스트링을 이용하는 분석이 필요한데, 대부분의 분석에서 이는 너무 무겁다.

(2) 같은 함수를 여러곳에서 부를수록 크기가 커진다. $c_1, c_2, \ldots, c_n$을 같은 실행경로위에서 같은 함수를 호출하는 여러 지점들을 위상순서로 정렬한 것이라고하자. 이들로부터 생성되는 거짓고리들은 c_1과 c_n사이의 모든 프로그램 지점들을 포함하게 된다. 만약 이들사이에 다른 함수들을 호출하는 지점들이 존재한다면 호출되는 모든 함수들도 이 고리에 포함될 것이다.

이러한 이유로 인해서 실제 C 프로그램에 나타나는 거짓고리는 전체 프로그램의 80-90%를 포함할 정도로 커지게 된다. 표 1.3 은 오픈소스 프로그램들에 대해서 전체 프로그램의 함수와 지점의 수에대해 고리가 포함하는 함수와 프로그램 지점의 개수의 비율을 보여준다. 대부분의 프로그램 지점 및 함수들이 하나의 고리에 포함됨을 알 수 있다. 따라서 이들 프로그램을 분석하기 위해서는 프로그램의 대부분을 포함하는 커다란 고리의 고정점을 계산해야 한다.

2.2 고리의 크기에 따른 분석비용의 증가

크기가 큰 고리일수록 그것을 분석하려면 더 많은 반복된 계산이 필요하기 때문에 프로그램의 크기가 커질수록 거짓고리로 인한 성능저하가 심해진다. 그 이유는 거짓고리의 크기가 클수록 그 안의 값들사이의 의존성(dependence)이 강해지기 때문이다. 변수들간의 컨트롤에 따른 의존성(control-dependence)를 생각해보자. 만약 어떤 고리내에 길이 l인 의존성 체인 $v_1 \succeq v_2 \succeq \cdots \succeq v_n$이 존재한다면, 일반적인 넓히기 연산 [13, 2] 과 가지치기 [13] 을 이용하여 그 고리의 고정점을 구하기 위해서는 고리을 따라 $n+2$회이상 반복적으로 돌면서 분석해야 한다. 고리를 처음 돌때에는 변수 v_1의 값이 변하고 나머지 변수들의 값은 변하지 않기 때문이다. 즉, 변수 v_i는 고리을 i번 반복적으로 분석한 경우에 값이 변하게 된다.

프로그램의 크기가 더 크고, 거짓고리내에 더 긴 의존성 체인이 존재하게 되면, 더욱 여러번 반복된 계산이 필요하게 되고, 분석의 효율이 떨어질 것이다. 그림 3은 오픈소스 프로그램을 분석할 때 이러한 현상이 실제로 일어나고 있음을 보여준다. 각 그래프의 가로축은 분석의 진행과정(iteration)을 세로축은 해당 시간에 프로그램의 어느 함수가 분석되고 있는지를 보여준다. 왼쪽 그래프는 spell-1.0 오른쪽은 sed-4.0.8을 분석하는 경우이다.

sed-4.0.8의 경우에는 전체 분석과정이 아닌 중간 200,000 iteration만 그린 결과이다. 두 경우 모두 비슷한 패턴이 계속 반복되는 양상을 보인다. 그리고 한 패턴은 프로그램의 대부분의 함수를 포함하고 있다. 이것은 분석과정이 하나의 큰 고리의 고정점을 구하기 위해서 그 고리를 여러차례 반복적으로 분석하고 있다는 증거가 된다. spell-1.0과 sed-4.0.8의 결과를 서로 비교해보면 더욱 흥미로운 결과를 알 수 있는데, 패턴이 반복되는 횟수가 sed-4.0.8의 경우에(150여회) spell-1.0보다(30여회) 훨씬 많다는 것이다. 이는 sed-4.0.8이 spell-1.0보다 프로그램 크기가 더 크며, 따라서 거짓고리의 크기도 더 크기 때문에 고리의 고정점을 계산하는데 더 많은 반복이 필요하기 때문이다.

3 리턴지점에 민감한 워크리스트 알고리즘

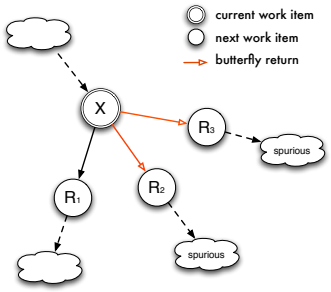
3.1 프로그램의 표현

프로그램은 다음과 같은 노드들로 구성된 수퍼그래프 [3] 로 표현된다고 하자.

$$n \in Node = entry_f \mid exit_f \mid call_f^{g,r} \mid rtn_f^c \mid cmd_f$$

첨자 f는 각 노드가 속한 함수를 의미한다. $entry_f, exit_f$ 는 함수마다 하나씩 가지고 있는 시작지점, 끝지점이며 $call_f^{g,r}$는 함수 g를 호출하는 노드인데 대응하는 리턴노드가 r 임을 의미한다. rtn_f^c는 리턴노드로써 대응하는 함수호출노드가 c이다. cmd_f는 일반적인 명령문이다.

3.2 아이디어



분석이 거짓고리를 따라 반복계산하는 것을 방지하기 위한 전체적인 아이디어는 위의 그림과 같다. 현재 X를 분석하고 있으며 다음 할일이 R_1, R_2, R_3라고 하자. 이 때, $X \to R_2, X \to R_3$는 거짓고리를 만드는 계산과정임을 알 수 있다면 이들 계산을 생략하고 $X \to R_1$로만 진행하는 것이다. 그렇다면 거짓고리를 만드는 계산과정을 어떻게 찾아내는지 살펴보자. 워크리스트 알고리즘의 수행과정에서 다음의 세가지 단계를 통해서 거짓고리를 찾아낸다.

(1) 함수 g를 호출하는 함수호출지점 c를 분석하는 경우, g를 분석한 후 r로 리턴함을 기억한다. 이 때, r은 c에 대응하는 리턴지점이다. g를 모두 분석한 후 알고리즘은 리턴지점으로 기억하고있는 r로만 리턴한다. 즉, 다른 리턴들은 모두 거짓고리를 만들 가능성이 있다고 판단한다.

(2) 각 함수마다 리턴지점 r을 올바르게 기억하기 위해서, 워크리스트 알고리즘은 하나의 함수f를 분석하고 있을때에는 f를 호출하는 함수호출지점을 다시 분석하는 경우가 없

Algorithm 1 리턴지점을 기억하는 워크리스트 알고리즘

1: $\mathcal{W} \in Worklist = 2^{Node}$
2: $\mathcal{T} \in Table = Node \to State$
3: $\mathcal{R} \in ReturnMap = ProcName \to 2^{Node}$
4: rss_basic : $Worklist \times Table \times ReturnMap \to Worklist \times Table \times ReturnMap$
5: $= \lambda(\mathcal{W}, \mathcal{T}, \mathcal{R})$.
6: **repeat**
7: $F := \mathsf{first}(\mathcal{W})$
8: $n := \mathsf{choose}(F)$
9: $\mathcal{W} := \mathcal{W} - \{n\}$
10: $(\mathcal{W}', \mathcal{T}', \mathcal{R}') := \mathsf{process\_one}(n, \mathcal{W}, \mathcal{T}, \mathcal{R})$
11: $(\mathcal{W}, \mathcal{T}, \mathcal{R}) := (\mathcal{W}', \mathcal{T}', \mathcal{R}')$
12: **until** $\mathcal{W} = \emptyset$
13: **return** $(\mathcal{W}, \mathcal{T}, \mathcal{R})$
14: process_one : $Node \times Worklist \times Table \times ReturnMap \to Worklist \times Table \times ReturnMap$
15: $= \lambda(n, \mathcal{W}, \mathcal{T}, \mathcal{R})$.
16: $m := \mathcal{F}\ \mathcal{T}\ n$
17: **if** $n = call_f^{g,r}$ && $\mathsf{isnotrecursive}(g)$ **then**
18: $\mathcal{R}(g) := \{r\}$
19: **end if**
20: **if** $m \not\sqsubseteq \mathcal{T}(n)$ **then**
21: **if** $n = exit_g \wedge \mathsf{isnotrecursive}(g)$ **then**
22: $\mathcal{W} := \mathcal{W} \cup \mathcal{R}(g)$
23: **else**
24: $\mathcal{W} := \mathcal{W} \cup \mathsf{succof(n)}$
25: **end if**
26: $\mathcal{T}(n) := \mathcal{T}(n) \sqcup m$
27: **end if**
28: **return** $(\mathcal{W}, \mathcal{T}, \mathcal{R})$

도록 보장해주어야 한다. 우리는 함수가 리턴하기 전에 다시 분석되는 일이 없도록 하는 워크리스트 오더를 정의하여 이 성질을 보장한다.

(3) 재귀호출의 경우에는 위의 알고리즘을 적용하지 않는다. 재귀호출에 대해서는 하나의 리턴지점을 올바르게 기억할 수 없기 때문이다.

3.3 함수호출그래프의 역위상순서로 워크리스트를 조작

우리는 올바른 함수분석순서를 위해서 워크리스트 오더 $\geq_{r.t.o}$를 정의한 후 워크리스트 알고리즘이 이 순서대로 할일들을 처리하도록 한다. 워크리스트내의 두 노드 n, m가 있다고 할 때 두 노드가 속하는 함수를 각각 f, g라고 하자. $n \geq_{r.t.o} m$ 는 f가 g보다 함수호출그래프에서의 역위상순서로 앞서거나, 이 순서가 같다면, 함수내의 실행흐름그래프상에서 약한위상순서로 [5] 앞설때로 정의한다.

재귀호출을 제외하면 $\geq_{r.t.o}$는 함수마다 기억하고 있는 리턴지점 정보의 올바름을 보장해준다. 왜냐하면 이 순서는 호출된 함수가 호출한 함수보다 항상 먼저 분석됨을 의미하기 때문이다. 예를 들어서 함수 f, g를 생각하고 f가 g를 호출한다고 하자. g의 노드들이 f의

노드들보다 함수호출그래프상의 역위상순서가 더 높기 때문에 두함수의 노드들이 워크리스트에 동시에 존재한다고 해도 항상 g의 노드들이 먼저 분석된다. 이는 f내에 g 호출하는 지점들이 여럿이라고 해도 g가 리턴할 지점은 항상 하나로 올바르게 기억됨을 의미한다.

3.4 재귀함수 처리

위의 방식은 재귀함수가 있을경우에 작동하지 않는다. g가 재귀함수일경우 g를 분석하는 도중에 자신을 호출할 수 있기 때문이다. 이 경우 각 함수마다 하나의 리턴지점을 기억하면, 이전에 기억하고 있던 리턴정보를 잃어버리게 된다.

따라서 재귀함수가 아닌 함수를 호출하는 경우에만 리턴지점을 저장하고, 재귀함수는 모든 리턴지점으로 돌아가도록 한다. 이것은 우리 알고리즘이 재귀함수 호출로 생기는 거짓 고리는 제거하지 않음을 의미한다. 하지만 실제 C프로그램에서 대부분의 함수는 재귀함수가 아니기 때문에 알고리즘은 대부분의 성능저하문제를 해결할 것으로 예상한다.

3.5 알고리즘

Algorithm 1은 위에서 설명한 알고리즘이다. 전체적인 구조는 기본적인 워크리스트 알고리즘과 유사하다. 먼저 함수 first가 워크리스트부터 $\geq_{r.t.o}$로 노드들을 꺼내오는 역할을 하며, 다음과 같이 정의된다.

$$\mathsf{first}(\mathcal{W}) = \{n \in \mathcal{W} \mid \forall m \in \mathcal{W}.\ m \not\trianglerighteq_{r.t.o} n, m \neq n\}$$

워크리스트로부터 꺼내온 노드의 계산은 process_one이 담당한다. 의마함수 $\mathcal{F}$를 이용하여 출력메모리를 새로 계산한 후, 현재 계산중인 노드가 함수호출노드이며 재귀함수를 호출하지 않을 경우 리턴지점을 저장한다. $\mathcal{R}$은 각 함수마다 분석을 끝내고 돌아가야 할 리턴지점을 저장한다. 워크리스트에 새로 계산할 노드들을 추가할때에는 할때에는 함수의 끝지점인지를 확인한 후 재귀함수가 아니라면 $\mathcal{R}$이 저장하고 있는 리턴지점만 추가한다.

4 실험

우리는 기존의 Sparrow로부터 위의 알고리즘을 적용한 RSS를 만들었다. 따라서 Sparrow와 RSS는 워크리스트 알고리즘과 워크리스트 오더에서 차이가 날뿐 나머지 부분은 똑같다. Sparrow의 경우에는 워크리스트 오더로 함수호출그래프에서의 위상순서, 실행흐름그래프에서의 약한위상순서로 정의되는 오더를 사용하였다. 왜냐하면 우리의 실험들에서 이 오더로 분석할때 평균적으로 가장 계산량이 적었기 때문이다. RSS는 3장에서 설명한 워크리스트 오더를 사용한다.

4.1 세팅

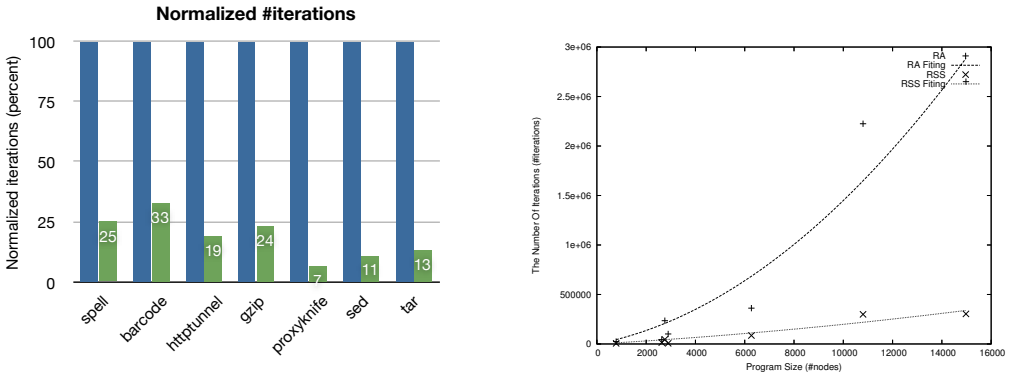
Sparrow와 RSS로 7개의 오픈소스프로그램을 분석하여 성능을 비교하였다. 표 II은 실험에 사용된 소프트웨어들의 정보와 분석결과를 보여준다. **LOC**는 코드의 라인수, **#procs**는 함수의 개수, **#nodes**는 프로그램 지점의 개수를 의미한다. 이들을 분석할때는 모두 글로벌분석, 즉 `main`부터 시작해서 전체프로그램을 분석하였다. 실험은 모두 인텔 펜티엄 4 3.2GHz 프로세서와 4GB메모리를 갖춘 리눅스 2.6 시스템에서 수행하였다.

분석결과의 비교에는 다음 두가지를 이용하였다.

- **#iters** 분석을 종료하기 위해서 워크리스트 알고리즘의 메인 루프가 수행된 횟수이다. 분석에 필요한 직접적인 계산량을 의미한다.
- **time** 분석에 소요된 초단위 시간이다.

Software	Properties			Sparrow		RSS	
	LOC	#procs	#nodes	#iters	time(s)	#iters	time(s)
spell-1.0	2,213	31	782	24,442	33.21	6,163	12.76
barcode-0.96	4,460	57	2,634	43,573	181.92	14,462	52.46
httptunnel-3.3	6,174	110	2,757	235,494	816.37	45,849	747.45
gzip-1.2.4a	7,327	135	6,271	361,875	1838.86	85,840	672.19
proxyknife-1.7	7,744	113	2,893	101,339	364.80	6,958	25.11
sed-4.0.8	26,807	294	14,976	2,650,218	30037.06	303,970	4118.47
tar-1.13	28,333	222	10,800	2,225,255	26391.10	299,076	18357.36

[표 II]실험대상 소프트웨어와 분석결과



(a)계산량 비교 (b)복잡도 비교

[그림 3] 실험결과. 제시한 알고리즘을 적용하였을때 분석의 계산량과 복잡도 감소를 보여준다. 계산량은 Sparrow의 계산량을 100으로 할 때 RSS의 계산량의 비율을 보여준다. 복잡도 그래프는 프로그램 크기가 증가함에 따라 계산량이 증가하는 양상을 보여준다.

4.2 결과

그림 3(a)는 Sparrow와 RSS이 소모한 계산량을 비교하여 보여준다. 평균적으로 RSS는 Sparrow의 계산량을 81.15%만큼 줄였다. 이에따라 분석시간도 평균 60%가 줄어들었다. 예를 들어서 gzip을 분석하는데 있어서 Sparrow대비 RSS의 계산량은 24%이며, 분석시간은 37%이다.

그림 3(b)는 RSS의 시간복잡도가 Sparrow에 비해서 향상되었음을 보여준다. Sparrow의 경우에는 프로그램크기가 커질수록 계산량이 급한 곡선으로 올라가는데 비해서 RSS의 경우에는 작은 기울기의 직선으로 올라가는 모습을 볼 수 있다. 이 결과는 Sparrow의 분석에서는, 2.2 장에서 설명했듯이, 프로그램의 크기가 클 수록 거짓고리의 크기도 커짐에 따라 분석의 효율이 점점 더 나빠지고 있음을 증거한다. RSS에서는 이 효율저하 원인을 제거하였기 때문에 분석의 계산복잡도가 향상되었다.

5 결론

이 논문은 정확도가 낮은 분석, 특히 함수호출경로를 구분하지 않을경우에 발생하는 분석의 효율저하의 원인을 설명하고 해결책을 제시하였다. 이 문제는 간단한 프로그램을 분석할때는 눈에 잘 띄지 않지만, 오픈소스등의 실제 프로그램을 분석할때에는 분석기의 성능

에 큰 영향을 주게된다. 오픈소스 프로그램들을 대상으로 실험한 결과 제시한 알고리즘을 적용할 경우 기존 알고리즘에 비해서 평균 81.15% 향상된 성능을 얻을 수 있었다.

참고문헌

[1] M.Sharir, A.Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications* In Muchnick, S.S. and Jones, N.D., editors, Prentice-Hall: 1981Wiley: New York, 1995; pages17-31

[2] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL* ; pp238-252, 1997

[3] Thomas Reps, Susan Horwitz and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pp49-61, 1995

[4] Florian Martin. Experimental Comparison of call string and functional Approaches to Interprocedural Analysis. In *Computational Complexity*, pp63-75, 1999

[5] Francois Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pp128-141, 1993

[6] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, pp5-23, 2004

[7] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *PLDI*. pp235-248, 1992

[8] S. Guyer and C. Lin. Client-driven pointer analysis. In *SAS* pages 214–236, June 2003

[9] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *CC*, pp17-31, Springer, 2006.

[10] Erik Ruf Context-insensitive alias analysis reconsidered. In *PLDI*, 1995

[11] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. In *ACM TOPLAS*, 29(5):73-123, 2007

[12] Sparrow: static source code analyzer. `http://www.spa-arrow.com`

[13] Youngbum Jung and Jaehwang Kim and Jaeho Shin and Kwangkeun Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis. In *SAS*, pp203-217, 2005

[14] Yongin Jhee, Minsik Jin, Yungbum Jung, Deokhwan Kim, Soonho Kong, Heejong Lee, Hakjoo Oh, Daejun Park and Kwangkeun Yi. Abstract Interpretation + Impure Catalysts: Our Sparrow Experience. Presentation at the *Workshop of the 30 Years of Abstract Interpretation*, San Francisco, `ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf`, 2008

[15] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *ISMM*, 2008

오 학 주

– 2001-2005 한국과학기술원 학사
– 2005-2007 서울대학교 석사
– 2007-현재 서울대학교 박사과정

<관심분야> 프로그램 분석 및 검증

이 광 근

– 1983-1987 서울대학교 학사
– 1988-1990 Univ. of Illinois at Urbana-Champaign 석사
– 1990-1993 Univ. of Illinois at Urbana-Champaign 박사
– 1993-1995 Bell Labs., Murray Hill 연구원
– 1995-2003 한국과학기술원 교수
– 2003-현재 서울대학교 교수

<관심분야> 프로그램 분석 및 검증, 프로그래밍 언어