공학석사학위논문

정적 분석에서의 함수 호출 사이클 제거를 통한 C 프로그램의 분석 속도 개선

Improving the Speed of Global Static Analysis by Removing Static Function Call Cycles in C Programs

2014 년 2 월

서울대학교 대학원

전기·컴퓨터 공학부

허 진 영

공학석사학위논문

정적 분석에서의 함수 호출 사이클 제거를 통한 C 프로그램의 분석 속도 개선

Improving the Speed of Global Static Analysis by Removing Static Function Call Cycles in C Programs

2014 년 2 월

서울대학교 대학원

전기·컴퓨터 공학부

허 진 영

정적 분석에서의 함수 호출 사이클 제거를 통한 C 프로그램의 분석 속도 개선

Improving the Speed of Global Static Analysis by Removing Static Function Call Cycles in C Programs

지도교수 이 광 근

이 논문을 공학석사학위논문으로 제출함

2013 년 12 월

서울대학교 대학원

전기·컴퓨터 공학부

허 진 영

허진영의 석사학위논문을 인준함

2014 년 1 월

위 원 장	이 상 구	(인)
부위원장	이 광 근	(인)
위 원	김 선	(인)

요약

이 논문은 정적 분석에서 나타날 수 있는 함수 호출 사이클 제거를 통해 C 프로그램의 분석 속도를 개선하는 방법을 제시한다. 흐름을 고려한 정적 분석에서 나타나는 복잡한 함수 호출 사이클은 분석 속도에 커다란 영향을 준다. 이런 사이클을 제거하고, 흐름을 고려하지 않은 분석 결과를 이용해 분석 속도 개선하는 방법을 제시한다. 이론적으로 이 방식이 안전함을 증명하고, 실용적인 분석기 구현을 위해 일부 안전성과 정확도를 포기하는 대신 분석 속도를 평균 1.5배 정도 개선함을 실험을 통해 보인다.

주요어: 정적분석, 프로그램분석, 요약해석, 함수호출사이클

학번: 2012-20885

목차

그림 목차

표 목차

제 1 장 서론

정적 분석 시 거대한 함수 호출 사이클이 나타나는 C 프로그램들이 있다. 이러한 프로그램을 정적 분석하는 비용은 비슷한 LOC (lines of code)를 갖지만 사이클이 없는 프로그램에 비해 매우 크다. 이 장에서는 정적 분석 시 거대 함수 호출 사이클이 발생하는 프로그램에 대해 소개하고, 이를 분석하는데 얼마나 많은 비용이 소모되는지 설명한다. 또한 이러한 사이클이 발생한 원인에 대해서도 간략히 기술한다.

1.1 동기

스파스 분석(sparse analysis) 프레임워크[1]를 이용하여 100만 라인의 C 프로그램을 분석할 수 있다. 이 기술을 통해 안전하고 정확하게 대형 C 프로그램을 분석할 수 있는 실용적인 정적 분석기를 구현할 수 있다. 스파스 분석 프레임워크는 요약 해석[2, 3, 4]에 기반하며, 흐름을 고려한(flow-sensitive) 분석으로, 프로그램 분석 시 필요한 정보만을 가져와 빠르게 계산을 수행해 낸다.

하지만 정적 분석 시 나타나는 함수 호출 관계에 사이클이 존재한다면, 스파스 프레임워크를 통한 분석 시간은 매우 길어진다. 정적 분석은 안전한 분석을 위해 실제 실행과 관계 없이 코드 상에 나타나는 모든 함수 호출을 추적할 필요가 있다. 이로 인해 실제 실행되는 함수 호출보다 복잡한 함수 호출 관계를 가질 수 있고, 실제보다 더 복잡한 사이클이 나타날 수도 있다.

정적인 함수 호출 관계에 사이클이 있다는 것은 정적 분석을 통해 도출한 함수 호출 그래프 상에 SCC (strongly connected component)가 존재한다는 뜻이다. 함수 호출 그래프는 유향 그래프로, 정점은 함수 이름, 간선은 호출 함수에서 피호출 함수로의 호출 관계로 표현한다. 이 그래프에서 SCC가 존재하면 함수 호출에 사이클이 있다고 할 수 있다. 그림 1.1는 sendmail-8.13.6의 SCC

Programs	**LOC**	**maxSCC**	**DU(s)**	**Main(s)**	**Total(s)**
tar-1.13	20K	13	30	10	40
less-382	23K	46	96	39	135
nano-2.3.2	25K	36	145	56	201
make-3.76.1	27K	61	65	19	84
wget-1.9	35K	13	59	8	67
screen-4.0.2	45K	65	348	37	385
a2ps-4.14	64K	9	87	13	100
xboard-4.7.0	67K	89	620	320	940
mutt-1.4.2.3	87K	155	1,354	1,080	2,434
sendmail-8.13.6	130K	68	3,746	1,860	5,606
vim60	227K	1,306	25,412	45,879	71,291
bash-4.2	302K	435	4,746	1,680	6,426

표 1.1: 프로그램 별 분석 시간

그래프를 표현한 것이다. SCC에 속한 함수는 총 68개로 이 함수들이 서로 복잡한 호출 관계에 있음을 볼 수 있다.

표 1.1을 보면 최대 SCC의 크기가 클수록 정적 분석에 더 많은 시간이 필요함을 알 수 있다. 코드의 크기가 비슷하지만 SCC는 3배 이상 차이나는 tar-1.13과 less-382의 경우 분석 시간이 less-382가 3배 이상 소요된다. a2ps-4.14와 xboard-4.7.0의 경우도 코드 크기는 비슷하지만, 분석 시간은 10배 가까이 차이가 나며, SCC도 10배 가까이 차이나는 것을 볼 수 있다.

프로그램 분석 시간이 SCC의 크기에 크게 의존하는 것을 알 수 있다. 그림 1.2은 make-3.76.1의 코드를 임의로 변경하여 SCC의 크기를 변경하고 각각에 대해 분석을 해 본 결과이다. SCC 크기가 작을 수록 분석이 빨리 끝남을 알 수 있다.

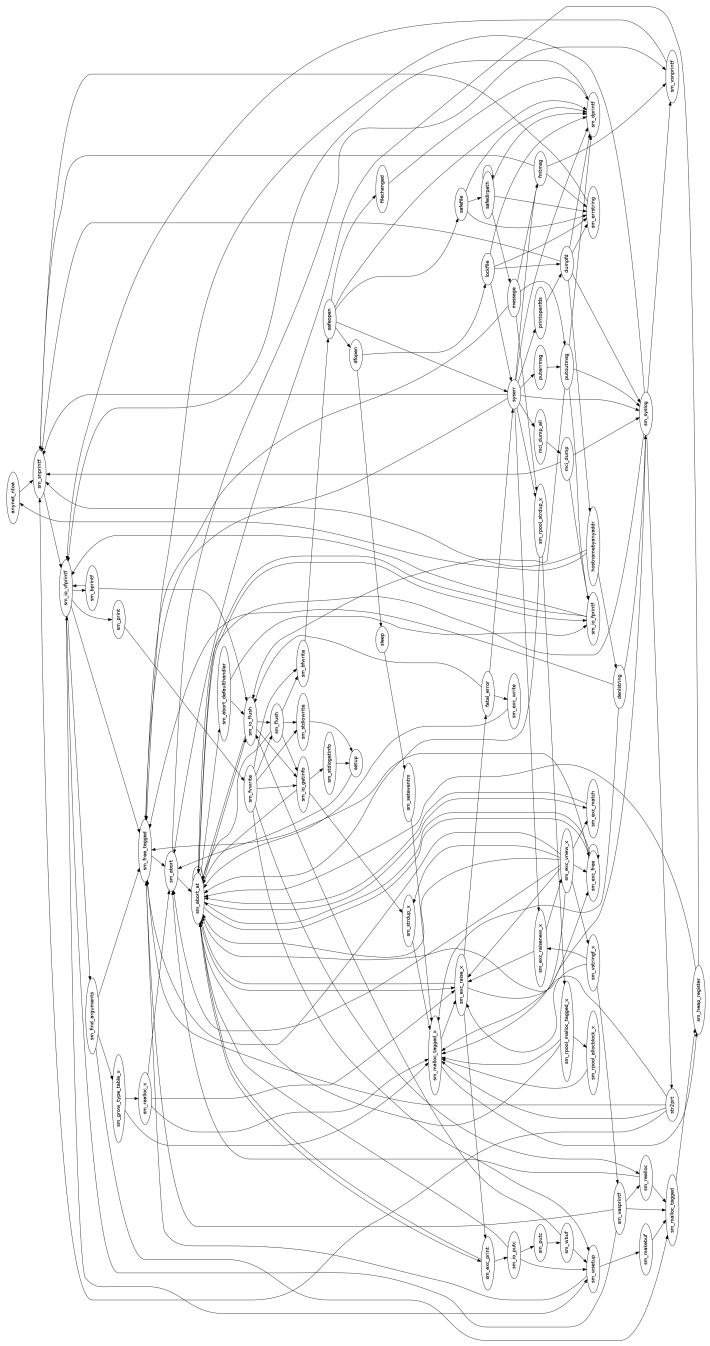

그림 1.1: Sendmail-8.13.6의 SCC 그래프

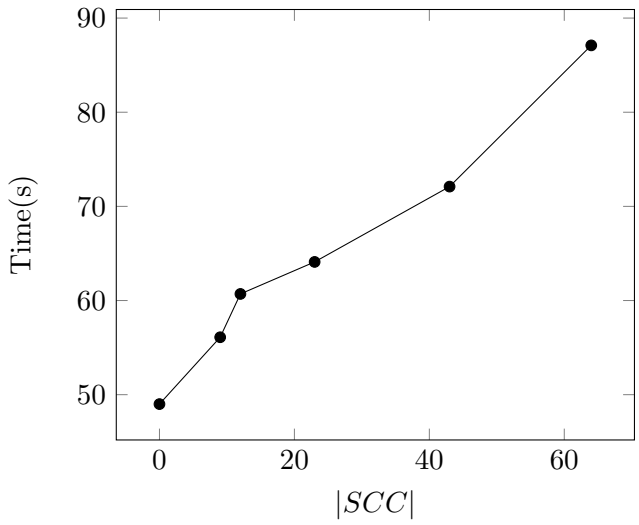

그림 1.2: SCC 크기와 분석 시간의 관계 (make-3.76.1)

1.2 사이클의 원인

정적 분석 시 프로그램에 사이클이 발생하는 원인은 다양하다. 실제 사이클은 프로그래머의 의도에 의해 만들어질 수 있으며, 의도하지 않았지만 우연에 의해 만들어지기도 한다. 또한 정적 분석 중에 가짜 사이클이 생길 수도 있다. 정적 분석은 안전한 분석을 위해 실제 실행 여부와 상관없이 모든 실행 가능한 경로를 포섭해야 하기 때문이다.

[6]에서는 사이클의 원인으로 잡일코드, 예외코드, 가짜 실행 경로로 분류하고 각각에 대해 예를 들어 설명하고 있다. 잡일코드는 프로그램 실행에는 큰 영향이 없는 디버깅 코드, 에러 코드 등을 말하며, 예외코드는 메모리 할당 실패와 같이 일반적인 상황에서는 발생하기 힘든 예외적인 상황을 처리하기 위한 코드를 말한다. 가짜 실행 경로는 그림 1.3처럼 실제 프로그램 실행에서는 실행이 불가능한 경우를 말한다. 하지만 함수 호출 문맥을 고려하지 않은 (context-insensitive) 정적 분석을 할 경우 가짜 실행 경로를 판별하기가 어렵고, 이러한 가짜 실행 경로로 인해 사이클이 발생할 수 있다.

```
1    quit() {
2        edit(NULL);
3    }
4    edit(char* filename) {
5        if (filename == NULL)
6            return (edit_ifile(NULL_IFILE));
7    }
8    edit_ifile(IFILE ifile) {
9        if (ifile == NULL_IFILE) {
10           return 0;
11       }
12       open_altfile(...);
13   }
```

그림 1.3: 가짜 실행 경로의 예 (less-382). `quit`에서 `open_altfile`로의 프로그램 흐름은 불가능하다.

1.3 논문의 구성

이 논문은 다음과 같이 구성된다. 2장에서는 함수 호출 사이클의 문제 상황과 개선책을 설명하고, 3장에서는 구현 세부 사항과 실험 결과에 대해 다룬다. 4장에서는 본 논문의 한계점과 이를 보완하기 위한 아이디어에 대해 소개하고, 5장에서 결론을 내린다. 부록에는 실험을 위해 제거한 함수 호출 목록을 나열하고 있다.

제 2 장 개선 방안

흐름을 고려(flow-sensitive)한 정적 분석에서 나타나는 프로그램의 복잡한 사이클은 분석기의 성능에 있어 커다란 걸림돌이다. 이 장에서는 사이클이 왜 프로그램 분석 속도에 영향을 주는 지 논하고, 이에 대한 해결책으로 사이클을 끊고 흐름을 고려하지 않은(flow-insensitive) 전분석 결과를 이용해 정적 분석 속도를 개선하는 방안에 대해 설명한다.

2.1 문제 상황

스파스 분석(sparse analysis)은 흐름을 고려한 정적 분석으로, 효율적인 분석을 위해 프로그램의 데이터 의존 관계를 이용한다[1]. 데이터 의존 관계는 프로그램의 변수 값이 정의되는 지점(def)과 해당 변수가 사용되는 지점(use)를 연결하는 변수 의존 관계 그래프(def-use graph)로 표현한다. 예를 들어 변수 `x`가 프로그램 i 지점에서 정의되고, j 지점에서 사용된다면 정점 i에서 정점 j로 향하는 간선은 `x`의 정보를 갖는다. 이를 통해, 프로그램 분석 시 필요한 변수의 정보만을 전달하는 것이 스파스 분석의 핵심이다.

데이터 의존 관계는 함수 내부 뿐만 아니라 함수 간에도 존재한다. 지역변수는 함수 내부에서 정의되고 사용되지만, 전역변수는 여러 함수에서 정의되고 사용될 수 있기 때문에 함수 간의 변수 의존 관계가 필요하게 된다. 따라서 함수를 호출할 때 피호출 함수 및 피호출 함수 안에서 불리는 다른 함수들(transitive callees) 모두에서 사용하는 전역 변수들의 정보를 변수 의존 관계 그래프에 넣어줘야 안전한 분석이 가능하다.

하지만 함수 호출 관계에 사이클이 존재하면 함수 간 데이터 의존 관계가 복잡해진다. 만약 피호출 함수가 SCC를 이루는 함수 중 하나라면 피호출 함수와 여기서 불리는 하위 함수들(transitive callees)은, 모든 정점이 연결되어

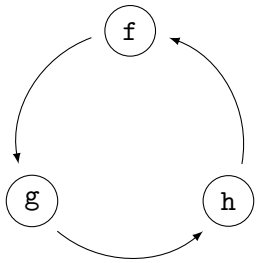

그림 2.1: 간단한 사이클

있는 SCC의 특성 상, SCC에 속한 모든 함수가 된다. 따라서 SCC에 속한 함수가 호출될 때에는 SCC에 속한 모든 함수에서 사용하는 모든 변수의 정보를 전달해줘야 한다.

이로 인해 SCC에 속한 함수를 호출할 때는 실제 필요한 것보다 많은 정보를 전달하게 된다. 예를 들어 그림 2.1와 같이 함수 f, g, h가 사이클을 이루고 있을 때, 함수 f에서 함수 g로 전달할 변수 정보는 함수 f, g, h 모두에서 사용하는 변수이어야 한다. 함수 g에서 h, h에서 f로 전달할 변수의 정보도 사이클에 포함되기 때문에 f에서 g로 전달하는 변수 정보와 동일하다. 만약 함수 f, g, h가 사이클을 이루지 않고 순차적으로 실행된다고 하면 f에서 g를 호출할 때는 g와 h에서 사용하는 변수 정보만 전달하면 되고, g에서 h를 호출할 때는 h에서만 사용하는 변수 정보만 전달하면 된다.

그림 2.2는 사이클이 있는 간단한 프로그램의 변수 의존 관계 그래프가 스파스 분석에서 지나치게 복잡함을 보여준다. 그래프의 정점은 코드의 라인을 의미하고 간선 위의 변수가 해당 프로그램 포인트를 통해 전달되는 변수들의 이름이다. 코드만을 보면 변수 x는 4번 라인에서 정의되어 5, 7, 8 라인으로 전달되고, 변수 y는 8에서 정의되어 9, 3, 4로 전달되는 정보만 존재하면 된다. 하지만 스파스 프레임워크는 안전한 변수 관계 그래프를 만들기 위해 부가적인 정보를 추가하고, 이로 인해 복잡한 그래프가 생성된다.

결국 사이클로 인해 정적 분석 시 전달해야 할 정보가 많아져 분석 속도가 저하된다. 그림 2.3는 프로그램 분석 시 전파되는 변수의 개수가 사이클의 여

```
1     int x = 0;
2     int y = 0;
3     void f(void) {
4         x = y;
5         g();
6     }
7     void g(void) {
8         y = x;
9         f();
10    }
```

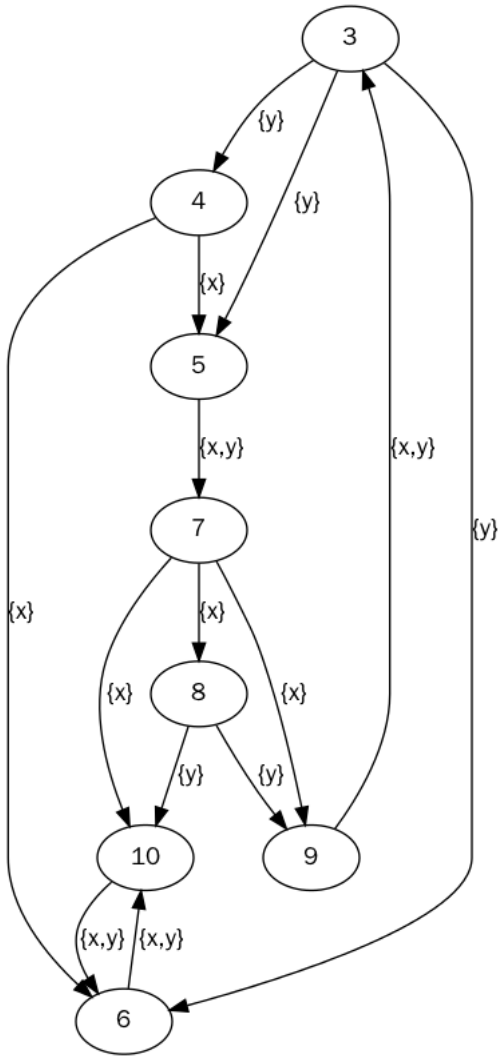

그림 2.2: 사이클이 있는 간단한 프로그램의 변수 의존 관계 그래프

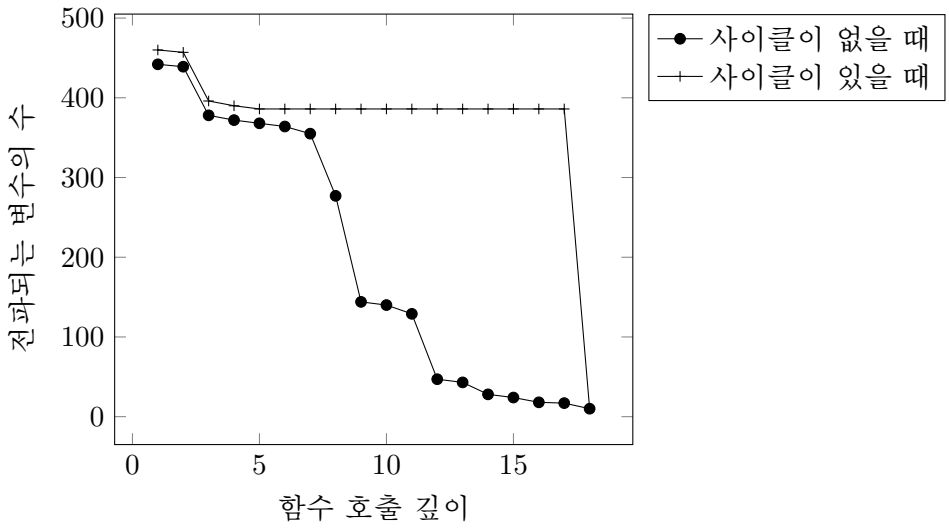

그림 2.3: 함수 호출 깊이에 따라 전파되는 변수의 수 (make-3.76.1)

부에 따라 어떻게 바뀌는 지 보여준다. 이 그래프는 make-3.76.1의 `main` 함수에서 시작하여 `log_working_directory` 함수까지의 실행 흐름에 대해 스파스 분석을 할 때 전달되어야 하는 변수의 개수를 사이클이 있을 때와 SCC를 모두 제거하여 사이클이 없을 때로 나눠 그래프로 나타낸 것이다. 앞서 설명했듯이, 사이클에 속한 함수는 사이클에 속한 모든 함수에서 사용하는 변수의 정보를 필요로 하므로, 동일한 SCC에 속해 있는 함수들에 전달해야 할 변수의 정보는 모두 동일하다. 그림 2.3에서도 사이클이 있는 경우 중간에 변수의 개수가 변하지 않고 일정하게 유지됨을 볼 수 있다. 사이클이 없는 경우는 함수 호출 깊이가 깊어질수록 하위에서 불리는 함수가 적어지므로 필요한 정보가 감소하는 것을 그래프를 통해 알 수 있다.

이 외에도 사이클로 인해 본분석 시 고정점 계산을 위한 연산(fixpoint iteration)이 더 필요한 것도 분석 시간이 오래 걸리는 한 요인이다. 만약 함수 호출 간에 사이클이 없다면, 프로그램 분석은 `main` 함수부터 시작하여 함수가 호출되는 순서대로 순차적으로 진행하면 된다. 이 때는 함수 내부에 존재하는 루프로 인한 고정점 계산만 수행하면 된다. 하지만 함수 호출 간에 사이클이

있다면, 함수 간에 데이터 의존 관계에도 사이클이 생기게 되고, 이를 계산하기 위한 추가의 고정점 계산을 필요로 하게 된다. 결국 사이클이 복잡해 질수록 해를 구하기 위한 연산이 더 필요하고, 이로 인해 전체 분석 시간은 사이클이 없는 프로그램보다 증가할 수 밖에 없다.

2.2 전분석

우리는 스파스 분석에서 제안하는 흐름을 고려하지 않은(flow-insensitive) 전분석 결과를 그대로 이용할 것이다. 스파스 분석은 전분석을 통해 데이터 의존 관계를 파악하며, 이러한 전분석 결과는 모든 프로그램 포인트에서 프로그램의 요약된 의미를 모두 포섭한다[1].

흐름을 고려한 분석과 그렇지 않은 분석에서 프로그램의 요약된 의미는 아래와 같이 표현할 수 있다. 흐름을 고려한 분석에서의 요약된 의미를 $\mathbf{lfp}\hat{F}$, 고려하지 않은 분석에서의 요약된 의미를 $\mathbf{lfp}\hat{F}_{pre}$라고 하면,

$$\begin{aligned} \mathbf{lfp}\hat{F} &\in \mathbb{C} \to \hat{\mathbb{S}} \\ \mathbf{lfp}\hat{F}_{pre} &\in \hat{\mathbb{S}} \\ \hat{\mathbb{S}} &= \mathit{Var} \to \hat{\mathbb{V}} \end{aligned}$$

이다. 여기에서 $\mathbb{C}$는 프로그램 포인트의 집합이고, $\hat{\mathbb{S}}$는 프로그램의 요약된 상태이다. 프로그램의 요약된 상태 $\hat{\mathbb{S}}$는 프로그램 변수 Var와 그에 대응하는 요약된 값 $\hat{\mathbb{V}}$로 표현한다. 흐름을 고려한 분석은 각 프로그램 포인트마다 그에 대응하는 프로그램 상태가 존재하지만, 흐름을 고려하지 않은 분석은 하나의 상태만 존재한다.

전분석은 흐름을 고려하지 않는 분석으로 $\mathbf{lfp}\hat{F}_{pre}$는 전분석의 결과이다, 전분석의 요약된 실행 함수(abstract semantic function) $\hat{F}_{pre} \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$는 아래와 같이 정의한다.

$$\hat{F}_{pre} = \lambda \hat{s}. \bigsqcup_{i \in \mathbb{C}} \hat{f}_i(\hat{s})$$

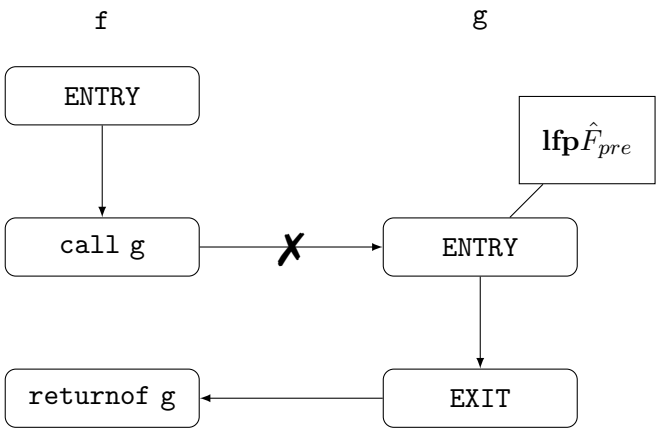

그림 2.4: 전분석 결과를 이용하여 그림 2.1의 사이클을 끊는 예

$\hat{f}_i \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$는 프로그램 포인트 i에서의 요약된 실행 함수, $\hat{s} \in \hat{\mathbb{S}}$은 요약된 프로그램의 상태를 나타낸다. 이를 통해 전분석은 모든 프로그램 포인트에서의 요약된 상태를 모두 모은(join) 것임을 알 수 있다.

따라서 흐름을 고려한 분석에서의 프로그램의 요약된 의미 $\mathbf{lfp}\hat{F}$와 전분석 결과 $\mathbf{lfp}\hat{F}_{pre}$에는 다음의 관계가 성립한다.

$$\forall i \in \mathbb{C}.\ \mathbf{lfp}\hat{F}(i) \ \sqsubseteq\ \mathbf{lfp}\hat{F}_{pre} \tag{2.1}$$

2.3 해결책

함수 간 복잡한 데이터 의존 관계를 피하기 위해, 사이클에 속한 함수 호출 경로를 끊고 피호출 함수의 진입(entry) 노드의 입력으로 전분석 결과를 넣어준다. 그림 2.1과 같이 함수 `f`와 `g`가 사이클에 속해 있다고 할 때, `f`에서 `g`로의 호출 경로를 끊는다고 가정하자. 그리고 `g`의 진입 노드에 전분석 결과 $\mathbf{lfp}\hat{F}_{pre}$를 넣어준다. 이를 그림으로 표시하면 그림 2.4와 같이 나타낼 수 있다.

호출 경로가 끊겼기 때문에 함수 간 데이터 의존 관계가 사라진다. 그림 2.4에서는 `f`에서 `g`로의 의존 관계가 사라졌다. 대신 기존의 데이터 의존 관계를 안전하게 포섭할 수 있는 수단으로 `g`의 진입점에 $\mathbf{lfp}\hat{F}_{pre}$를 넣어주었다. $\mathbf{lfp}\hat{F}_{pre}$는 프로그램에서 사용하는 모든 변수의 값을 갖고 있으므로, 기존 `f`에서 `g`로

전달하는 변수의 정보를 모두 포함한다. 이를 통해 f에서 g로 있었던 변수 의존 관계 그래프는 g의 진입점에서부터 시작되도록 변경된다.

전분석 결과를 함수의 진입 노드의 입력으로 넣고 분석을 진행해도 안전하다. 왜냐하면 전분석 결과는 모든 프로그램 포인트 i에서의 요약된 의미 $\mathbf{lfp}\hat{F}(i)$보다 크며(2.1), 프로그램의 요약 실행 함수들은 단조(monotonic) 함수이기 때문이다.

2.4 안전성 증명

프로그램의 요약 의미 방정식을 아래와 같이 표현할 수 있다. 프로그램 포인트 i에서의 요약된 상태를 $\hat{s}_i$, 요약된 실행 함수를 $\hat{f}_i$라고 하자 $(1 \leq i \leq n)$.

$$\begin{aligned} \hat{s}_1 &= \hat{f}_1(\hat{s}_1, \cdots, \hat{s}_n) \\ &\vdots \\ \hat{s}_n &= \hat{f}_n(\hat{s}_1, \cdots, \hat{s}_n) \end{aligned}$$

이는 다시 쓰면 아래와 같이 하나의 함수 $\hat{F}$으로 쓸 수 있고, 이 방정식의 해는 $\mathbf{lfp}\hat{F}$이다.

$$\begin{pmatrix} \hat{s}_1 \\ \vdots \\ \hat{s}_n \end{pmatrix} = \hat{F} \begin{pmatrix} \hat{s}_1 \\ \vdots \\ \hat{s}_n \end{pmatrix}$$

전분석 결과를 프로그램 포인트 i의 입력으로 사용하는 프로그램의 요약된 실행 함수를 $\hat{F_\bigstar}$라고 하자. $\hat{F_\bigstar}$는 $\hat{F}$과 동일하지만 입력 $\hat{s}_i$ 대신 $\mathbf{lfp}\hat{F}_{pre}$를 쓴다. 따라서 아래의 식이 성립한다.

$$\hat{F_\bigstar} \begin{pmatrix} \hat{s}_1 \\ \vdots \\ \hat{s}_i \\ \vdots \\ \hat{s}_n \end{pmatrix} = \hat{F} \begin{pmatrix} \hat{s}_1 \\ \vdots \\ \mathbf{lfp}\hat{F}_{pre} \\ \vdots \\ \hat{s}_n \end{pmatrix}$$

변형된 프로그램의 방정식은 아래와 같이 쓸 수 있고, 이것의 해는 $\mathbf{lfp}\hat{F_\bigstar}$이다.

$$\begin{pmatrix} \hat{s}_1 \\ \vdots \\ \hat{s}_n \end{pmatrix} = \hat{F_\bigstar} \begin{pmatrix} \hat{s}_1 \\ \vdots \\ \hat{s}_n \end{pmatrix}$$

Lemma 1. $\mathbf{lfp}\hat{F} \sqsubseteq \mathbf{lfp}\hat{F_\bigstar}$ *이다.*

Proof. 모든 프로그램 포인트 i에서의 요약 실행 함수 $\hat{f}_i$는 단조 함수이므로, $\hat{F}$과 $\hat{F_\bigstar}$도 단조 함수이다. $\hat{s}_i \sqsubseteq \mathbf{lfp}\hat{F}(i)$이고, (2.1)에 의해 $\mathbf{lfp}\hat{F}(i) \sqsubseteq \mathbf{lfp}\hat{F}_{pre}$ 이므로 $\hat{s}_i \sqsubseteq \mathbf{lfp}\hat{F}_{pre}$이다. 따라서,

$$\hat{F} \sqsubseteq \hat{F_\bigstar} \tag{2.2}$$

이다. 또한 $\hat{F_\bigstar}$의 도메인은 $\hat{F}$의 도메인과 동일하다. 따라서 두 도메인은 갈로아 연결이 되어 있고($\alpha = \gamma = id$), (2.2)과 fixpoint transfer theorem[4]에 의해 $\mathbf{lfp}\hat{F} \sqsubseteq \mathbf{lfp}\hat{F_\bigstar}$ 이다. □

제 3 장 구현 및 실험

3.1 구현

스파스 버전의 Sparrow[1]를 기반으로 분석기를 구현하였다. 구현은 크게 대상 함수 찾기와 사이클 끊기의 두 단계로 나눌 수 있다.

3.1.1 대상 함수 호출 경로 찾기

사이클을 끊기 위한 후보로 SCC 그래프의 다리(strong bridge)를 선택한다. SCC 그래프에서 다리란 SCC 그래프의 간선 중 하나로, 이를 끊을 경우 SCC가 둘 이상의 작은 SCC로 나뉘는 간선을 말한다. 또한 이를 선형 시간 안에 찾을 수 있는 알고리즘[5]이 존재하여 빠르게 계산할 수 있다.

하나의 SCC 그래프 안에는 SCC를 나눌 수 있는 여러 개의 다리가 존재하므로, 그 중 다리를 끊었을 때 SCC 크기를 최소로 만드는 것을 선택한다. 그림 2.1를 예로 들면 사이클에 있는 모든 간선이 SCC의 다리가 된다. 이 중 SCC의 크기를 가장 작게 만드는 간선 만을 선택하고 이 작업을 SCC 크기가 0이 될 때까지 반복한다. 여기에서 SCC의 크기를 가장 작게 하는 다리를 "*주요 SCC 다리*"라고 부르자. 주요 SCC 다리 또한 다수 존재할 수 있는데, 이 경우는 그 중 하나를 임의로 선택한다.

3.1.2 사이클 끊기

전분석을 통해 수집된 함수 호출 그래프에서 3.1.1에서 수집한 간선들을 제거한다. 그리고 호출 경로(간선)를 제거할 때 피호출 함수의 진입점의 입력 값으로 전분석의 결과를 넣어준다. 이를 통해 함수 호출 그래프 상의 SCC가 모두 사라지게 되며, 이는 곧 프로그램 내 함수 호출 사이클이 모두 사라짐을 의미한다. 이렇게 변경된 함수 호출 그래프를 바탕으로 데이터 의존 관계를

생성하고 이를 본분석에서 사용한다.

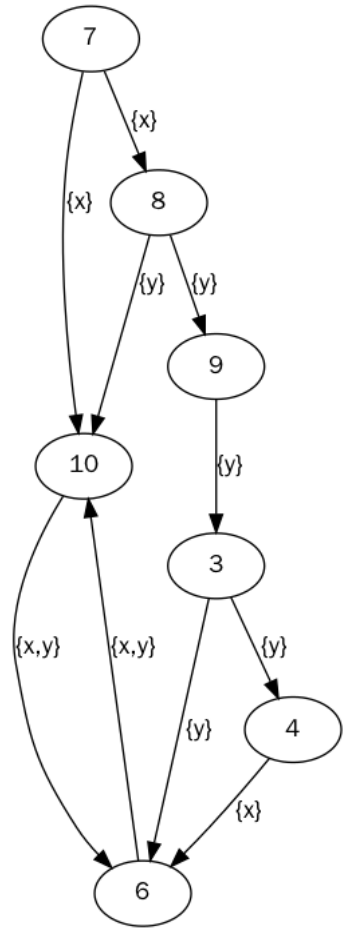

그림 3.1: 변경된 변수 의존 관계 그래프. 그림 2.2에서 f에서 g로의 경로를 끊은 후의 그래프

사이클에 속한 함수 간의 의존 관계가 사라졌기 때문에 데이터 의존 관계는 단순해지고 분석 속도도 빨라진다. 그림 3.1는 그림 2.2에 있는 간단한 프로그램에서 함수 f에서 g로의 호출 경로를 끊은 후의 변수 의존 관계 그래프이다. 그림에서 볼 수 있듯이 원래의 그래프보다 단순한 그래프가 생성되었다. 기존에는 변수 x가 4번 라인에서 정의되어 5, 7, 8 라인으로 전파가 됐지만, 사이클을 끊었기 때문에 기존의 경로 대신 g의 진입점인 7번 라인부터 전파가 시작되

Programs	**Before(s)**	**After(s)**	**Speed↑**
less-382	135	101	1.3 x
nano-2.3.2	201	126	1.6 x
make-3.76.1	84	59	1.4 x
screen-4.0.2	385	302	1.3 x
xboard-4.7.0	940	643	1.5 x
mutt-1.4.2.3	2,433	1,540	1.6 x
sendmail-8.13.6	5,606	4,381	1.3 x
vim60	71,291	31,781	2.2 x
bash-4.2	6,426	4,409	1.5 x

표 3.1: 사이클 제거 후 분석 시간 비교

도록 변경되었다. 이와 같이 간단하게 변경된 그래프를 통해 분석을 진행하기 때문에 더 빠른 시간 안에 분석을 마칠 수 있다.

3.2 실험 결과

SCC 사이즈가 40개 이상인 벤치마크 프로그램에 대해 실험을 진행하였다. 제거한 함수 호출 모음은 A.1에 나열하였다.

표 3.1은 프로그램에 있는 모든 사이클을 제거한 후의 분석 시간 비교 결과이다. **Before(s)**열은 사이클을 끊기 전 원 분석 시간(초)이며, **After(s)**열은 사이클을 모두 제거한 후의 분석 시간(초)를 의미한다. **Speed↑**는 **Before** 대비 **After**의 속도가 몇 배 더 빨라졌는지를 의미한다.

표 3.2는 프로그램에 있는 모든 사이클을 제거한 후 알람 수의 비교 결과이다. **Delta**는 알람의 증감을 표시한 열로 **Inc**는 새로 추가된 알람의 수, **Dec**는 기존 알람 중 줄어든 알람의 수를 뜻한다. make-3.76.1을 예로 들면, 178개의 알람이 증가했고, 기존의 알람 22개가 감소했다는 의미이다. $\mathbf{Alarms}_{\mathsf{FI}}$는 흐름을 고려하지 않은 분석을 했을 때의 전체 알람 개수이다.

Programs	**Before**	**After**	**Delta**		**Alarms$_{FI}$**
			Inc	**Dec**	
less-382	664	666	2	0	711
nano-2.3.2	2,464	2,496	37	5	2,574
make-3.76.1	1,556	1,712	178	22	2,113
screen-4.0.2	4,243	4,335	332	240	5,969
xboard-4.7.0	2,724	2,619	54	159	3,065
mutt-1.4.2.3	3,751	5,398	1,657	10	6,814
sendmail-8.13.6	6,156	6,245	114	25	7,457
vim60	13,499	13,490	88	97	14,324
bash-4.2	7,293	7,289	100	104	8,285

표 3.2: 사이클 제거 후 알람 수 비교

실험 결과를 통해 사이클을 제거한 프로그램의 분석 속도가 향상됨을 알 수 있다. 작은 프로그램에서는 약 1.3배 정도의 속도 개선이 있었고, vim60처럼 SCC의 크기가 1000이 넘는 복잡한 사이클을 갖는 프로그램에서는 2배 이상 분석 속도가 개선되는 것을 볼 수 있다.

또한 알람의 수를 통해 정확도 손실은 크지 않음을 알 수 있다. 일반적으로 흐름을 고려하지 않은 분석은 흐름을 고려한 분석보다 부정확하며, 이는 분석 후 알람이 더 많이 나오는 것으로 가늠할 수 있다. 하지만 사이클을 제거한 후 분석한 결과를 보면 흐름을 고려하지 않은 분석보다 알람의 수가 적으며, 흐름을 고려한 원 분석의 알람 수와 큰 차이를 보이지 않고 있다. 이는 사이클을 끊고 분석을 진행하는 것의 정확도 손실이 그렇게 크지 않음을 보여준다고 할 수 있다. 단, 알람이 줄어드는 이유에 대해서는 4.1에서 논한다.

제 4 장 한계 및 보완 사항

이론적으로는 안전하지만 실용적인 분석기 구현을 위해 일부 안정성과 정확도를 포기한 이유와 이를 보완할 수 있는 아이디어에 대해 기술한다.

4.1 부정확한 주소 요약으로 인한 안전성 손실

요약 해석에서 알 수 없는 값은 흔히 ⊤이라는 값으로 요약한다. Sparrow 역시 알 수 없는 포인터 주소 값에 대해 ⊤을 사용한다. 보통 포인터는 해당 포인터 변수가 가리키는 모든 주소의 집합으로 요약을 하는데, 알 수 없는 라이브러리 함수가 반환하는 포인터는 그 주소를 정확히 알 수 없으므로 ⊤으로 처리한다. 또한 C 언어는 포인터에 대한 산술 연산이 가능하므로 이러한 연산 과정을 통해 주소가 ⊤이 될 수도 있다.

하지만 ⊤ 주소를 참조하는 경우 문제가 발생한다. 요약된 세계에서의 ⊤ 주소는 프로그램이 사용하는 메모리의 모든 주소를 가리키는 값이기 때문에 ⊤ 주소를 정확하게 처리할 수 있는 방법이 없다. 예를 들어,

```
*p = lib();
```

과 같은 대입문이 있고, 변수 p의 요약된 값이 ⊤이라고 하자. 이 구문을 제대로 처리하기 위해서는 모든 메모리 주소의 값에 함수 lib의 반환 값을 넣어줘야 한다. 하지만 함수 lib의 의미를 알 수 없는 경우, 정적 분석은 ⊤의 값을 반환 값으로 사용한다. 이로 인해 실제 p가 가리키는 변수가 아닌 불필요한 변수의 값도 함께 ⊤으로 갱신이 되고 이는 부정확한 분석 결과를 초래한다. 사실 모든 변수의 값이 ⊤이 되는 정적 분석은 의미가 없다. 이런 문제를 피하고자 Sparrow의 경우, ⊤ 주소를 참조하는 코드에 대해서는 아무런 작업도 하지 않도록 설계가 되어 있다.

전분석 결과는 많은 ⊤ 주소를 갖을 수 있으므로 분석 결과가 더 부정확해지고, 일부 분석을 제대로 못하는 경우가 발생한다. 전분석은 흐름을 고려하지 않기 때문에 원래의 결과보다 더 큰 값을 갖게 된다. 이로 인해 더 많은 ⊤이 포함될 수 있고, 이를 본분석의 입력으로 넣어주게 되면 ⊤ 주소 참조와 같은 문제로 분석이 제대로 안되는 경우가 발생한다. 표 3.2에서 알람의 수가 줄어드는 원인도 여기에 있다.

이에 대한 대안으로 ⊤ 주소를 참조하는 경우 ⊤의 값을 출력하는 방법을 생각해 볼 수 있다. 하지만 이렇게 할 경우, 과도한 ⊤ 값으로 인해 다른 주소의 값이 ⊤이 되는 경우가 발생할 수 있다. 전분석에서 주소가 ⊤이 되는 경우, 변수 의존 관계 그래프를 부정확하게 구성하여 안전하지 않은 분석이 된다.

이를 보완하는 다른 방법으로 라이브러리 인코딩을 들 수 있다. 프로그램에서 사용하는 모든 라이브러리 함수들의 의미를 미리 분석기의 입력으로 넣어 보다 정확한 분석 결과를 기대할 수 있다. 단, 모든 라이브러리 함수를 인코딩하는 것은 현실적으로 어렵고, 분석기의 성능을 저하시킬 수 있으므로, 이에 대한 적절한 타협점을 찾는 것이 필요하다.

4.2 최적의 함수 호출 경로 선택

함수 호출 사이클에서 어느 호출 경로를 끊고 분석하느냐에 따라 분석 결과가 달라질 수 있다. 피호출 함수의 진입점에 넣은 전분석의 부정확한 값이 프로그램의 서로 다른 지점으로 전파되면서 서로 다른 결과를 낼 수 있기 때문이다. 표 4.1는 screen-4.0.2에서 서로 다른 함수 호출 경로를 끊고 실험한 결과이다. 각 실험마다 알람의 수가 다른 것을 알 수 있다.

따라서 최대한 정확도가 높은 결과를 낼 수 있는 함수 호출 경로 조합을 찾아야 한다. 3.1.1에서 언급했듯이 하나의 SCC에는 여러 개의 주요 SCC 다리가 존재할 수 있는데 우리는 이 중 하나를 임의로 선택하고 있다. 가장 간단한 방법은 모든 주요 SCC 다리 조합에 대해 분석을 수행하고 그 중 하나를 선택하는 것인데 이는 현실적으로 불가능하다. Screen-4.0.2의 예를 들면, 사이클을 모두

	Alarms
실험1	4,335
실험2	4,339
실험3	4,336

표 4.1: Screen-4.0.2의 실험 결과[a]

[a]실험 별로 끊은 함수 목록은 A.2 참조.

제거할 수 있는 주요 SCC 다리의 조합은 전부 672개에 달하며, 일반적으로 그 수는 SCC에 속한 정점의 개수에 비례하여 지수적으로 증가하기 때문이다.

아직 이에 대한 명확한 해결책은 없다. 각 함수 별로 전분석 결과를 넣고 분석을 한 후, 각 결과를 보고 어떤 패턴의 함수 호출을 제거할 때 더 부정확한 결과를 내는 지 실험적으로 확인할 필요가 있다.

제 5 장 결론

이 논문은 정적 분석에서 나타날 수 있는 함수 호출 사이클 제거를 통해 C 프로그램의 분석 속도를 개선하는 방법을 제시하였다. 일부 C 프로그램에서 정적 분석 중 나타나는 사이클이 분석의 성능에 커다란 영향을 주는 것을 보였고, 스파스 분석의 변수 관계 그래프가 복잡해 지는 이유에 대해 설명하였다. 이에 대한 해결책으로 주요 SCC 다리를 찾아 사이클에 속한 함수 호출 경로를 제거하고, 피호출 함수의 진입점에 흐름을 고려하지 않은 분석 결과를 넣어주어 분석을 진행하는 방법을 제시하였다. 사이클을 끊고 분석해도 이론적으로 안전함을 증명하고, 실용적인 분석기 구현을 위해 일부 안전성과 정확도를 포기하는 대신 분석 속도를 평균 1.5배 정도 개선함을 실험을 통해 보였다.

참고문헌

[1] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi, "Design and Implementation of Sparse Global Analyses for C-like Languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (ACM SIGPLAN Notices Volume 47 Issue 6)*, June 2012. pp. 229-238.

[2] Patrick Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1977. pp. 238-252.

[3] Patrick Cousot and Radhia Cousot, "Systematic design of program analysis frameworks," in *Proceedings of ACM Symposium on Principles of Programming Languages*, 1979. pp. 269-282.

[4] Patrick Cousot and Radhia Cousot, "Abstract Interpretation Frameworks," in *Journal of Logic and Computation*, 1992. pp. 511-547

[5] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni, "Finding Strong Bridges and Strong Articulation Points in Linear Time" in *International Conference on Combinatorial Optimization and Applications*, 2010. pp. 157-169.

[6] 유병준, "정적분석에서 허위 함수 호출 사이클의 조사," 석사학위논문, 전기컴퓨터공학부, 서울대학교, 2013.

부록

A.1 함수 목록

다음은 각 벤치마크 프로그램 별로 사이클을 끊기 위해 선택한 함수 호출 모음이다. '호출함수:피호출함수'의 형식으로 되어 있다.

- less-382 (6개)

  `cmd_left:cmd_rshift, edit_ifile:reedit_ifile, error:get_return, fch_get:wait_message, quit:edit, repaint:jump_loc`

- nano-2.3.2 (8개)

  `die:die_save_file, die:unpartition_filestruct, do_justify:do_input, do_prompt:get_prompt_string, do_replace:do_search, open_buffer:read_file, parse_include:parse_rcfile, search_init:do_gotolinecolumn`

- make-3.76.1 (6개)

  `allocated_variable_expand_for_file:variable_expand_for_file, ar_member_date:f_mtime, ar_member_date:file_exists_p, die:reap_children, fatal:die, update_file:update_file_1`

- screen-4.0.2 (16개)

  `AclSetPerm:AclSetPermWin, AddLoadav:GetLoadav, ChangeWindowSize:KillWindow, DoAction:StuffKey, DoAction:process_fn, DoCommand:DoAction, Msg:MakeStatus, PUTCHARLP:RAW_PUTCHAR, Panic:FinitTerm, RefreshHStatus:ShowHStatus, RefreshLine:DisplayLine,`

SendErrorMsg:MakeClientSocket, SetRendition:SetFont,

ShowHStatus:ClearArea, readpipe:Panic, xseteuid:Panic

- xboard-4.7.0 (10개)

  AutoPlayGameLoop:AutoPlayOneMove, CheckFlags:GameEnds,

  DisplayFatalError:ExitEvent, ExitEvent:GameEnds,

  ForwardInner:PauseEvent, GetEngineLine:ParseArgsFromString,

  ParseArgs:ParseSettingsFile, PauseEvent:HandleMachineMove,

  PauseEvent:Reset, SendToProgram:GameEnds

- mutt-1.4.2.3 (13개)

  _mutt_expand_path:mutt_default_save,

  _mutt_get_field:_mutt_enter_string, km_dokey:retry_generic,

  mutt_compose_menu:mutt_index_menu, mutt_do_pager:mutt_pager,

  mutt_enter_string:_mutt_enter_string,

  mutt_free_body:mutt_free_header, mutt_getch:mutt_query_exit,

  mutt_message_to_7bit:transform_to_7bit,

  mutt_parse_messageRFC822:mutt_parse_part,

  mutt_parse_multipart:mutt_parse_part,

  mutt_pattern_exec:perform_and, mutt_pattern_exec:perform_or

- sendmail-8.13.6 (32개)

  callsubr:rewrite, db_map_open:aliaswait,

  db_map_store:db_map_lookup, dropenvelope:returntosender,

  dropenvelope:savemail, finis:dropenvelope, maplocaluser:recipient,

  printopenfds:dumpfd, readaliases:parseaddr, reply:smtpquit,

  sendall:dowork, sendtolist:recipient, setup:sm_abort,

  sm_abort_at:sm_abort_defaulthandler, sm_abort_at:sm_exc_match,

  sm_abort_at:sm_exc_raise_x, sm_abort_at:sm_io_flush,

sm_bfwrite:safeopen, sm_exc_free:sm_abort_at,
sm_exc_print:sm_abort_at, sm_exc_print:sm_io_putc,
sm_exc_raise_x:fatal_error, sm_exc_raisenew_x:sm_exc_vnew_x,
sm_free_tagged:sm_abort, sm_io_flush:sm_flush,
sm_io_fprintf:sm_abort_at, sm_io_getinfo:sm_abort_at,
sm_io_getinfo:sm_strdup_x, sm_io_vfprintf:sm_bprintf,
sm_io_vfprintf:sm_find_arguments, sm_io_vfprintf:sm_wsetup,
sm_print:sm_fvwrite

- vim60 (51개)

  ExpandOne:ExpandFromContext, RealWaitForChar:xterm_update,
  SFtextChanged:SFupdatePath, alloc:lalloc, alloc_clear:lalloc,
  apply_autocmds:apply_autocmds_group, change_indent:shift_line,
  check_for_delay:ui_delay, check_state_ends:syn_update_ends,
  clip_get_selection:do_pending_operator,
  cmd_runtime:do_in_runtimepath, diff_check_fill:diff_check,
  do_cmdline:do_debug, do_cmdline:do_one_cmd, emsg2:emsg,
  emsg:msg_attr, emsg:redir_write, enter_buffer:open_buffer,
  eval1:eval2, ex_diffupdate:buf_write,
  fill_input_buf:read_error_exit, fill_input_buf:settmode,
  flush_buffers:inchar, get_attr_entry:set_hl_attr,
  get_expr_register:getcmdline, getcmdline:ex_window,
  init_highlight:do_highlight,
  ins_compl_check_keys:ins_compl_next, ml_append:open_buffer,
  ml_delete_int:ml_replace, ml_open_file:findswapname,
  ml_open_file:mf_close_file, ml_replace:open_buffer,
  msg_attr_keep:msg_end, msg_puts_attr:get_keystroke,
  out_flush:ui_write, plines_win_nofold:ml_get_buf, reg:regbranch,

screenalloc:lalloc, set_option_value:set_bool_option,

set_option_value:set_num_option,

set_option_value:set_string_option, set_termname:gui_init,

shell_resized:set_shellsize, update_screen:win_update,

update_topline:scroll_cursor_bot, validate_cursor:curs_columns,

vgetc:vgetorpeek, vim_regexec:vim_regexec_both, vpeekc:vgetorpeek,

wait_return:jump_to_mouse

- bash-4.2 (55개)

  _rl_dispatch:_rl_dispatch_subseq,

  _rl_read_init_file:rl_parse_and_bind,

  _rl_ttymsg:rl_forced_update_display,

  _run_trap_internal:parse_and_execute, assoc_dispose:hash_flush,

  call_expand_word_internal:expand_word_internal,

  cat_file:file_error, cleanup_dead_jobs:waitchld,

  command_subst_completion_function:rl_completion_matches,

  cond_and:cond_term, copy_arith_for_command:copy_command,

  copy_case_clause:copy_command, copy_coproc_command:copy_command,

  copy_for_command:copy_command,

  copy_function_def_contents:copy_command,

  copy_group_command:copy_command, copy_if_command:copy_command,

  copy_subshell_command:copy_command,

  copy_while_command:copy_command, declare_builtin:declare_internal,

  dispose_command:dispose_function_def,

  error_prolog:get_name_for_error,

  execute_command:execute_command_internal,

  execute_command_internal:execute_connection,

  execute_command_internal:execute_simple_command,

execute_command_internal:time_command,

execute_in_subshell:execute_command_internal,

expand_amble:brace_expand,

expand_words_no_vars:expand_word_list_internal,

expcomma:expassign, extract_command_subst:extract_delimited_string,

extract_command_subst:xparse_dolparen,

extract_dollar_brace_string:skip_double_quoted,

finddirs:glob_vector, gmatch:extmatch, gmatch_wc:extmatch_wc,

param_expand:parameter_brace_expand,

parse_and_execute:execute_command_internal,

parse_and_execute:parse_command,

parse_compound_assignment:read_token,

parse_comsub:parse_matched_pair, phash_flush:hash_flush,

print_arith_for_command:make_command_string_internal,

print_case_clauses:make_command_string_internal,

print_for_command:make_command_string_internal,

print_function_def:make_command_string_internal,

print_group_command:make_command_string_internal,

print_if_command:make_command_string_internal,

print_select_command:make_command_string_internal,

print_until_or_while:make_command_string_internal,

readtok:expr_streval,

rl_get_next_history:rl_get_previous_history,

skipsubscript:skip_matched_pair, term:expr,

unbind_variable:makunbound

A.2 Screen-4.0.2의 함수 목록

아래는 표 4.1에서 각 실험 별로 사이클을 끊기 위해 선택한 함수 호출 모음이다. '호출함수:피호출함수'의 형식으로 되어 있다.

- 실험1 (16개)

  AclSetPerm:AclSetPermWin, AddLoadav:GetLoadav,
  ChangeWindowSize:KillWindow, DoAction:StuffKey,
  DoAction:process_fn, DoCommand:DoAction, Msg:MakeStatus,
  PUTCHARLP:RAW_PUTCHAR, Panic:FinitTerm,
  RefreshHStatus:ShowHStatus, RefreshLine:DisplayLine,
  SendErrorMsg:MakeClientSocket, SetRendition:SetFont,
  ShowHStatus:ClearArea, readpipe:Panic, xseteuid:Panic

- 실험2 (16개)

  AclSetPerm:AclSetPermWin, ChangeWindowSize:KillWindow,
  DoAction:StuffKey, DoAction:process_fn, DoCommand:DoAction,
  GetLoadav:secfopen, Msg:MakeStatus, PUTCHARLP:RAW_PUTCHAR,
  Panic:FinitTerm, Panic:xseteuid, RefreshHStatus:ShowHStatus,
  RefreshLine:DisplayLine, SendErrorMsg:MakeClientSocket,
  SetRendition:SetFont, ShowHStatus:ClearArea,
  runbacktick:readpipe

- 실험3 (15개)

  AclSetPerm:AclSetPermWin, ChangeWindowSize:KillWindow,
  DoAction:StuffKey, DoAction:process_fn, DoCommand:DoAction,
  Msg:MakeStatus, PUTCHARLP:RAW_PUTCHAR, Panic:FinitTerm,
  Panic:RemoveStatus, Panic:xseteuid, RefreshLine:DisplayLine,
  Resize_obuf:RemoveStatus, SendErrorMsg:MakeClientSocket,
  SetRendition:SetFont, ShowHStatus:ClearArea

Abstract

Improving the Speed of Global Static Analysis by Removing Static Function Call Cycles in C Programs

Jinyoung Huh
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

This paper presents a technique for improving the speed of static analysis by removing static function call cycles in C programs. Static function call cycles estimated during flow-sensitive analysis significantly degrade the performance of the analysis. This paper describes how to disconnect such call cycles and how to utilize the result of flow-insensitive analysis to enhance the performance. The theoretical soundness of this technique is proved. To implement a practical analyzer, there are some loss of precision and soundness, but the experimental result shows that the analysis time is reduced by about 1.5 times on average for benchmark programs.

Keywords: static analysis, program analysis, abstract interpretation, function call cycle

Student Number: 2012-20885