17-2-R02

계산 진행 방식 변화를 통한 점증적 유효변수 찾기 알고리즘 수행속도 향상

(Improvement of Iterative Live Variable Analysis by Computation Reordering)

윤정한 · 한태숙

한국과학기술원 전자전산학과 전산학전공

dolgam@pllab.kaist.ac.kr han@cs.kaist.ac.kr

요 약

자료흐름 분석(DFA : Data Flow Analysis)를 수행하는 방식에는 크게 점증적 방법(iterative method)와 구역합침 방법(elimination method)가 있다. 구역합침 방법과 달리 추가적인 작업 및 정보가 필요 없고, 구현이 간단하다는 장점 때문에 실제 컴파일러에서 주로 사용되고 있는 방법은 점증적 방법이다.

하지만 널리 사용되고 있는 것과는 달리 이에 관한 연구는 그리 활발하지 않았다. 80년대에 정해진 내용에 따라 깊이 우선 순서(depth-first order)로 반복하는 특성은 변하지 않은 채 20년이 넘게 사용되어 오고 있다. 지금까지의 알고리즘만으로도 속도 면에서 충분하다고 생각해 왔기 때문이다. 하지만 컴파일 과정에서 DFA가 사용되는 경우는 참으로 많다. 비슷한 구조의 것들이 여러 개가 있게 되는 DFA의 경우 조금의 속도 향상도 전체적으로는 큰 영향을 끼칠수 있게 된다.

이에 대해 본 논문에서는 점증적 DFA 방법론(iterative DFA framework)에서 계산 진행 방식에 대한 개선점을 찾아보았다. 일단 전체 방법론에 대해 적용하기보다 유효변수 조사(LVA: live variable analysis)의 경우에 대해서 새로운 계산 진행 방식을 알고리즘을 적용, 실험하였다. 자주 사용되는 분석으로써 대표적인 LVA는 Zephyr 컴파일러의 경우 전체 컴파일 시간에서 약 7%를 차지하고 있다. 이 LVA에 새로운 계산순서를 적용한 결과 기존 알고리즘보다 36.4%빨리 완료함을 알 수 있었다. 이는 전체 컴파일 시간에서 2.6%를 줄이는 효과를 가져왔다. 이와 같은 시도들이 DFA 전체에 적용된다면 컴파일 시간 단축에 큰 효과가 있을 것이다. 이는 JIT(Just in Time) 컴파일 시스템에서는 프로그램의 성능과 직결되므로 그 영향력이 크게 작용한다.

1. 서 론

1.1. 연구 배경

자료흐름 분석(DFA)에는 많은 분석 기법들이 포함되어 있다. 진행 방향에 따라서 순방향(forward) DFA와 역방향(backward) DFA로 분류한다. 순방향 DFA에는 사용-정의 연결(Use-Def chain), 유효식 찾기(available expression), 변수복사 삭제(copy propagation) 등이 있고, 역방향 DFA에는 유효변수 조사

(LVA), 정의-사용 연결(Def-Use chain) 등이 있다. 하나하나의 복잡도는 그렇게 크지 않지만 비슷한 구조의 여러 가지 기능이 있으면서 자주 사용되므로 컴파일 전체 시간에서 큰 비중을 차지하게 된다. Zephyr[13]에서의 실험 결과 LVA의 경우 다른 최적화 기법이 한 번 실행되는 전체 컴파일 경로 상에서 평균 4번 정도 실행이 되는 것을 알 수 있었다. 4배 자주 실행된다는 것은 그 자신의 복잡도에 4배를 한 다음 다른 기능들과 비교해야 한다는 것을 의미한다. 전체 컴파일 시간을 줄이는 데 있어서 사소한 부분이라고 여겨지던 부분에서도 컴파일 성능에 손해를 주지 않고도 큰 향상을 가져올 수 있음을 알 수 있다.

이와 같은 컴파일 속도의 향상은 JIT 컴파일러나 대화식 개발 환경(interactive development environment), 동적 컴파일 시스템(dynamic compilation system)에서 그 중요성이 더욱 부각된다. 상호 개발 환경과 비슷한 개념으로 대화식 컴파일 시스템(Interactive compilation system)[12]이 있다. 대화식 컴파일 시스템이라는 것은 최적화 기법들의 적용 순서를 사용자가 이리 저리 바꿔 보면서 컴파일 하고자 하는 프로그램에 가장 적절한 최적화 기법들의 적용 순서를 찾아볼 수 있게 해 주는 해 주는 시스템이다. 이와 같은 경우 사용자가 수많은 컴파일 시행착오를 거치게 되는 것이 필연적인데 이 때 컴파일 시간의 단축은 사용자에게 커다란 장점이 될 것이다.

본 논문에서는 앞에서 언급하였듯이 널리 사용되고 있는 점증적 DFA 방법론에서의 깊이 우선 순서의 계산이 가지는 문제점을 제시하고 이를 해결할 수 있는 방법을 LVA에 적용하여 본다. 다른 정보를 필요로 하는 것이 없이 단지 순서만을 바꿈으로써 DFA의 수행 속도의 향상이 있음을 실험을 통해 보인다. 그리고 각 계산 순서의 알고리즘들이 가지는 장단점에 대해서 생각해 본다. 그리고 점증적 DFA 방법론 개선해 나갈 방향에 대해 논의한다.

1.2. 문제 제기와 해결책

기존 점증적 DFA 방법론을 간략히 설명하자면 다음과 같다. 입력된 코드 전체를 훑어가면서 직접 연결되어 있는 단위구역(basic block)들 간에 정보교환을 한다. 이와 같은 일을 반복하면서 얻고자 하는 정보가 변하지 않을 때까지 계속 코드 전체를 훑어 가는 것이다.

이와 같이 하는 알고리즘이 실제적으로 많이 사용되는 이유는 구현이 간단하면서도 그 반복 횟수가 실질적으로는 그리 많지 않다는 이유 때문이다[1].

하지만 반복적이라는 태생적인 성격 때문에 반복 횟수가 현실적으로 얼마나 될지는 알 수 없고, 그 횟수가 꽤 커지는 경우도 발생함을 알 수 있다. 또한 일부분에서만 정보 변경이 있어도 코드 전체를 스캔해야 하는 것은 점증적 DFA 방법론에서의 가장 큰 단점이라 할 수 있다. 계산 순서를 깊이 우선 순서로 잡아 둔 다음 바로 연결되어 있는 단위구역들 사이에서만 정보를 교환하는 구조이므로 일부분이 변경되면 그 영향력이 어디까지 끼칠지 알 수 없으므로 다음에도 프로그램 전체를 읽어가면서 그 영향력을 인접한 단위구역들에게 전달하는 것이다. 바로 인접한 것에만 정보를 전달하므로 코드 전체 스캔의 횟수가 많아지게 되는 것이다.

문제점은 한 가지 더 있다. 기존 점증적 DFA 방법론은 인접한 단위구역에는 무조건 자신이 현재 가지고 있는 정보를 전달해 준다. 그러므로 반복문과 같이 한 번 읽는 것만으로는 정보를 완성하지 못 하고 몇 번을 읽으면서 변화하는 정보를 계속 갱신해 주어야 하는 곳에서도

1) 반복되는 횟수와 시간 복잡도에 대한 상한을 [6]에서 이야기하고 있다.

정보를 완성하기 전인 미완성인 정보를 계속 전달하여 정보가 완성된 다음에 한 번에 할 수 있는 연산을 여러 번 반복하게 되는 문제점이 바로 그것이다.

이러한 문제점들을 해결하기 위해 새로운 계산 진행 방식을 도입하였다. 직접 인접한 단위구역에만 정보를 전달하면서 자신의 영향력을 전파하는 것이 아니라 자신이 가진 정보를 가지고 그 정보가 필요한 곳을 모두 찾아가면서 먼저 수행하는 것이다. 인접한 basic block에 정보를 전달한 다음 정보를 전달받은 단위구역에서 다시 재귀적으로 정보를 전달하면서 전달할 정보가 없어질 때까지 이를 먼저 반복해 나가는 것이다. 이와 같이 함으로써 부분적으로 갱신하기 위해 코드 전체를 확인해 보아야 하는 일을 막을 수 있다. 갱신되어야 하는 곳을 먼저 능동적으로 찾아가면서 정보를 추가해 가기 때문에 현재 갱신하지 않아도 되는 단위구역을 방문할 필요가 없게 되는 것이다. 이와 동시에 루프와 같은 곳에서 아직 완성되지 않은 정보들을 전파하지 않고 기다리게 함으로써 필요 없는 계산도 없애는 효과를 볼 수 있다.

본 논문에서는 이와 같은 아이디어를 LVA에 적용하여 구현하였다. 알고리즘을 이해하기 위해 LVA에 대해서 생각해 보자. 어떠한 변수가 살아있다는 것은 변수 값이 정의된 다음 그 값이 사용되는 위치까지 그 정보가 저장되어 있어야 한다는 것을 의미한다. 이에 대한 출발점은 정의된 위치가 아닌 변수가 사용된 위치이다. 정의된 변수가 사용이 안 될 수는 있지만, 사용되는 변수가 정의되지 않았다면 그 코드는 잘못된 것이기 때문이다.

본 논문의 기본 아이디어는 다음과 같다. 변수 v가 단위구역 B에서 사용되었다고 하자. v가 사용된 위치에서 그 정보를 직접 연결된 단위구역들에만 알려주지 말고, 바로 그 변수가 정의되어 있는 단위구역들까지 거슬러 올라가면서 그 경로에서 거쳐가는 모든 단위구역들에 v는 살아있어야 한다고 기록하는 것이다. 직접 연결되어 있는 단위구역들끼리만 정보 교환을 하던 것을 정보가 필요한 하나의 체인에 직접 다 알려 주는 것이다.

이와 같이 하면 변경된 정보가 어디까지 그 영향력을 미치는지를 알 수 있으므로 부분에서 정보가 변경되었을 때 코드 전체를 스캔하면서 그 영향력의 범위를 확인해 줄 필요가 없다. 변

```
procedure OldLVA
  for each basic block B do
    ∈(B) = USE(B)
  end
  while changes to any OUT(B) occur do
    for each basic block B in reverse DFS order do
      OUT(B) = ∪ all>essor ScfB ∈(S)
      ∈(B) = USE(B) ∪ OUT(B) − DEF(B)
    end
  end
end of procedure
```

(그림 1) 점증적 DFA 방법론을 따르는 LVA 알고리즘

경된 정보가 필요한 영역만을 좇아가면서 갱신해 주면 되는 것이다. 그러므로 기존 점증적 DFA 방법론에서 문제점인 “방문하지 않아도 되는 단위구역의 방문”이 없어지게 된다. 또한 필요한 연산을 먼저 하게 되면서 불필요하게 반복되는 연산을 줄일 수 있게 된다. 기존 알고리즘의 비효율적인 점은 3.4절에서 자세히 다루었다. 그리고 이를 어떻게 새로운 알고리즘이 극복하는지를 4장에서 기술하였다.

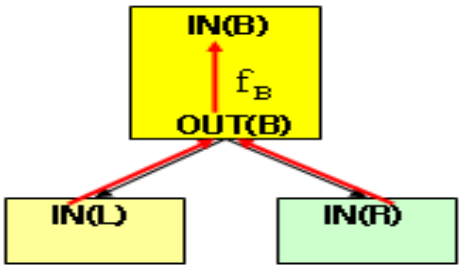
1.3. 논문 구성

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구들에 대한 소개로 DFA 방법론에 관한 기존의 연구들에 대해 간단히 정리하였다. 3장에서는 기존의 점증적 LVA 알고리즘을 알아보고 이 알고리즘이 가지는 단점을 분석하였다. 4장은 새로이 제안한 계산 진행 방식을 적용한 새로운 LVA 알고리즘을 소개한다. 5장에서 두 가지 LVA 알고리즘의 성능 비교 및 분석을 하고, 마지막으로 6장에서는 결론을 맺고 새로운 알고리즘의 보완해야 할 점들과 향후 연구 과제, 특히 새로운 LVA 알고리즘을 만든 아이디어를 일반적 DFA 방법론으로 어떻게 확장할 것인가에 대하여 생각해 본다.

2. 관련 연구

여러 가지 분석 기법들을 하나의 틀로 묶어 DFA라는 이름을 붙인 후 이들에 대한 연구는 두 가지 방향이 주를 이루었다. 하나는 점증적 방법[2]이고, 하나는 구역합침 방법[3]이다. DFA의 전반적 내용에 대해서는 [4]에 잘 소개되어 있다. 구역합침 방법에 대한 정리는 [7]에 잘 되어 있다. 가장 최근으로는 DJ 그래프를 이용한 구역합침 DFA방법론이 소개되었다[8]. 이는 기존의 구역합침 방법들이 가지는 가장 큰 문제점인 실행 흐름 그래프(CFG: Control flow graph)가 하나로 합쳐(reducible)질 수 있지 않을 때는 그래프에 변형을 주어야 한다는 점을 해결하였다. 그래프의 변형을 주지 않아도 된다는 장점으로 인해 기존 구역합침 방법들보다 수행 속도가 현저히 빨라지게 되었다.

DFA가 하나의 큰 틀로 통일되면서 그에 포함되는 여러 가지 분석기법들에 대해 따로 연구가 진행되기보다는 이 틀을 어떻게 개선하여 속도를 빠르게 할 것인가가 초점이 되어 왔다. 대부분의 연구들이 80년대를 전후하여 이루어져서 최근에 그리 활발한 연구가 되고 있다고는 할 수 없다.



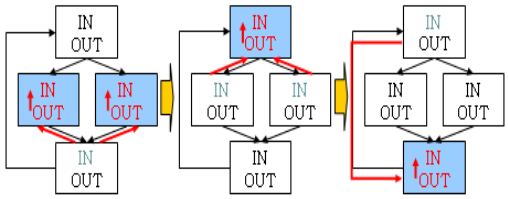
(그림 2) 정보 함수 적용

이와 같이 두 가지 큰 줄기에서 벗어난 여러 다른 시도들[9][10][11]도 있었다. 같은 계산을 하는 구역들을 합쳐서 중복되는 계산을 줄이거나[9], 사용-정의 연결 등의 기존에 존재하는 정보를 이용하여 LVA를 빠르게 하고자 하는 시도[10]. 이들과 달리 프로파일(profile)을 통한 확률적으로 진행 방향을 정하여[11] 계산 속도 향상을 꾀하는 방식 등 여러 가지 시도들이다. 하지만 구현상의 복잡성, 그리고 기존 DFA 방법론이 가지는 성능만으로도 충분하다고 생각해 왔다. 때문에 실제 컴파일러에서 그리 널리 사용되고 있지는 않다. 기존의 방법 성능이 그리 나쁘지 않으면서 구현이 간단하다는 큰 장점 때문이다.

3. 기존 Live Variable Analysis 알고리즘

3.1. LVA를 위한 기존의 알고리즘

모든 단위구역에 대하여 *DEF(B)*, *USE(B)*는 미리 계산된 상태에서 알고리즘은 시작된다. 최소한 단위구역에서 사용하는 변수들은 입구에서 살아있어야 하기 때문에 *USE(B)*가 *IN(B)*의 초기값이 된다[2]. 정보 방정식(flow equation)에 따라 *OUT(B)*은 자신의 후계단위구역(successor)들의 *IN(S)*를 모두 합집합한 것이다[3]. 그리고 *IN(B)*은 정보 함수에 *OUT(B)*을 넣어서 나오는 값이 된다. 이와 같은 과정을 반복하면서 *IN*, *OUT*을 만들어 나가고, 그 값이 변하지 않을 때까지 계속 하는 것이다. 점증적 DFA 방법론을 적용하여 깊이 우선 순서로 계산하는 LVA 알고리즘은 그림 1에 나타나 있다.



(그림 3) 기존 LVA 알고리즘의 수행 과정

그림 2, 3을 보면서 동작 원리에 대해 생각해 보자. 그림 2는 정보 함수가 어떻게 적용되는지를 보여주는 그림이다. 아래쪽의 *IN*들을 모아서 *OUT*을 만들고, 그 *OUT*을 정보 함수에 넣어서 다시 *IN*을 만드는 과정을 그림으로 나타내고 있다. 화살표는 정보가 흘러가는 진행 방향을 나타내고 있다. 그림 3에서 입력된 코드에 반복문이 있을 경우 진행되는 순서를 보여준다. 짙은 색으로 표현된 단위구역이 자신의 *IN*, *OUT*이 바뀌는 단위구역이다. 아래에서부터 깊이 우선 순서의 역순으로 단위구역을 방문하면서 *IN*, *OUT*을 만든다. 후계단위구역들의 *IN*으로 *OUT*을 만들고, 만든 *OUT*을 정보 함수에 넣어 다시 *IN*을 만드는 과정의 반복이다. 입력된 프로그램에 반복문이 없다면 이 알고리즘은 한 번 코드를 전체적으로 읽기(그림 1에서의 for 반복문을 전체로 한 번 다 수행한 것)만으로 끝난다. 반복문이나 분기문이 없다면 모든 단위구역들이 자신의 후계단위구역들에게서만 영향을 받으므로 깊이 우선 순서의 역순으로 계산을 하면 그 계산 순서가 딱 맞아떨어지기 때문이다. 하지만 반복문이나 분기문에 의해 정보가 완성되지 않은 상태에서 그 다음으로 진행이 있을 수 있는 경우 갱신된 *OUT*의 영향을 적용시키기 위해 다시금 그 단위구역에 방문할 필요가 생기게 되는 것이다. 이런 이유로 그림 1에서 while 반복문이 필요하게 되고, 반복적 알고리즘의 모습을 갖추게 된다.

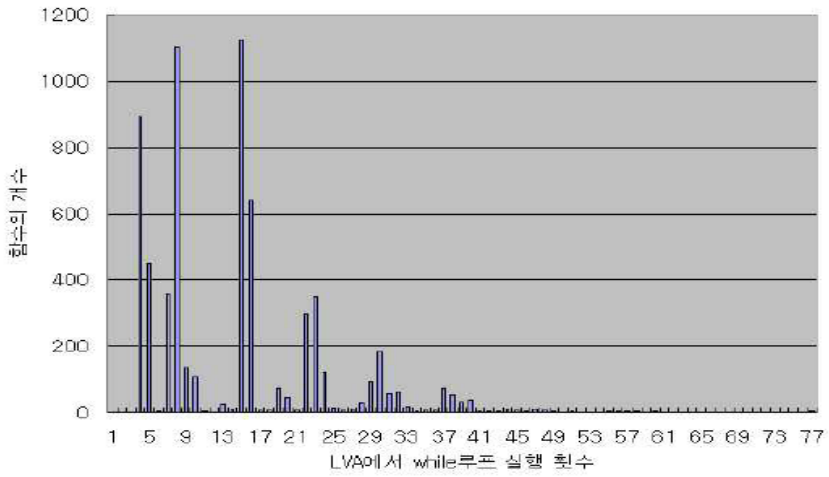
3.2. 기존 알고리즘의 문제점

본 논문에서 *Zephyr*[13] 컴파일러로 실험한 결과(그림 4 참조)[4] LVA 수행 시 while 반복문이

2) DFA에 대한 자세한 설명은 여기서 생략한다. 이에 대한 내용은 [4][5]에 잘 나와 있다.

3) LVA는 유효식 찾기와 대조적인 모습을 보인다. 일단 LVA는 역방향 DFA, 유효식 찾기는 순방향 DFA. 그리고 두 단위구역으로부터 정보를 받았을 때 LVA는 합집합을 하지만 유효식 찾기는 교집합을 한다. 도메인이 레티스(lattice)라는 점에서 교점 연산자(meet operator)와 합점 연산자(join operation)의 차이라 할 수 있다. 이와 같은 차이는 알고리즘에서 계산 순서를 생각하는데 큰 차이가 있다. 이에 관해서는 6장 결론에서 다시 언급하겠다.

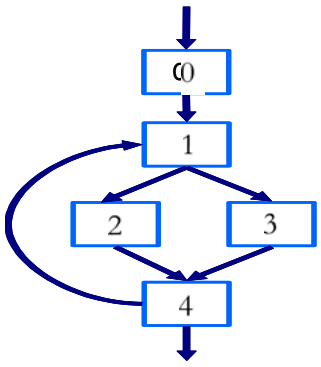
4) 성능평가 프로그램 MediaBench[14]의 모든 프로그램에서 측정해 본 결과이다. Zephyr 컴파일러는 함수들 단위로 최적화를 실시하므로 성능평가 프로그램에서의 프로그램 단위로 결과를 나누지 않는 것도 의미가 있다. 총 함수의 개수는 6515개이다. 실험 환경에 대해서는 5장에서 서술하였다.



(그림 4) 기존 LVA 알고리즘에서 while 반복문이 수행되는 횟수 분포

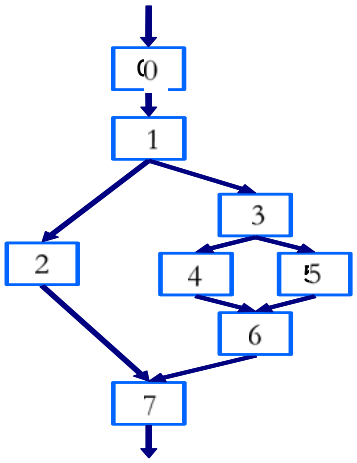
평균 14.3회 실행되었다. 최소 1회부터 최대 77회까지 존재했으며, 13회에서 15회에 대부분의 함수들이 집중되어 있었다. 평균값인 14.3회라 함은 그리 적은 횟수라 할 수만은 없다. 경우에 따라 그 횟수가 아주 크게도 나올 수 있음을, 즉 LVA 한 번의 수행에서도 많은 시간을 소요하게 될 수 있음을 시사하고 있다.

이와 같이 while 반복문이 많은 횟수로 실행되어야 하는 배경이 기존 알고리즘의 문제점이다. 원인이라고 한다면 "직접 연결된 단위구역에 무조건 정보 전달"이라는 말로 표현할 수 있을 것이다. 그렇다면 이 원인이 어떠한 현상으로 나타나는지에 대해 아래에서 2가지로 요약해 보겠다.



(그림 5) 반복문의 예제

첫째. 반복문의 정보를 완성치 않고 계산을 진행한다. 그림 5에서 반복문이 있는 CFG의 예를 보여주고 있다. LVA의 경우를 생각하고 있으므로 LVA의 진행 방향은 CFG에서 나타난 화살표를 역행하는 방향임을 인지하고 그림을 보기 바란다. 밑에서부터 정보를 만들어 올라가기 때문에 4번부터 시작을 하여 정보를 위로 보낸다. 4번이 2, 3번에게 주고, 2번은 1번에게, 3번도 1번에게 정보, 즉 4번의 *IN*값을 전달한다. 이 때 1번에서 문제가 발생한다. 기존 알고리즘은 1번이 4번에게 IN값을 전달함과 동시에 그 윗 방향인 0번 단위구역으로도 *IN*값을 전달하도록 되어 있다. 이 때 1번에서 전달된 *IN*값에 의해 4번의 *OUT*값이 변하게 되고 이는 결국 1번의 *IN*값에 영향을 끼치게 되어 있다. 즉 아직 1번의 *IN*값은 완성되지 않은 상태인 것이다. 하지만 이렇게 완성되지 않은 1번의 *IN*값은 알고리즘 상 0번으로 전달되게 되고 0번은 그 정보에 의해 자신을 갱신한 후 계속 위로 진행하게 된다. 1번의 *IN*값이 완성된 다음에도 이와 같은 일이 반복되게 되는데 이렇게 보았을 때 1번의



(그림 6) 분기문의 예제

*IN*값이 완성되기 전에 한 계산들은 모두 1번의 *IN*이 완성된 이후의 연산에 포함되는 일이 됨을 알 수 있다. 즉 하지 않아도 되는 연산을 1번 단위구역 위쪽 전체에 대해서 행하게 된다는 것이다. 이는 반복문이 아래쪽에 있을수록 그 영향력이 더더욱 커진다.

두 번째로 분기점에서의 정보 수집이 있다. 7번에서부터 시작한 정보가 1번까지 도착하는 데 있어서 깊이 우선 순서를 사용하였을 때 2, 3, 4, 5, 6번 모두의 정보를 가지고 올라가는 것이 실패할 수 있다. 2번으로 전달된 *IN*값을 1번으로 보내고, 6번으로 전달된 값이 4, 5번으로 전달될 때 1번이 3번보다 먼저 0번으로 출발해 버릴 수 있다는 것이다. 이 경우 1번의 *IN*값을 0번으로 전달한 다음에야 3번에서부터 *IN*값을 전달받아 1번의 *IN*값을 완성할 수 있게 된다. 이렇게 됨으로써 어쩔 수 없이 while 반복문을 한 번 더 돌려서 이전 while 반복문에서 1번의 값이 변경되었으니 전체 코드를 스캔하면서 인접한 단위구역으로 정보 보내기를 실행해 주어야 한다. 이는 결국 루프에서와 마찬가지로 완성되지 않은 정보를 0번으로 보냄으로써 중복되는 계산을 한다는 단점과 함께 필요 없이 한 번의 루프를 더 돌아야 한다는 점에서 수행 시간을 지연시킨다.

이와 같이 인접한 단위구역으로 정보를 무조건 보내야 한다는 점과 그 순서가 단순히 깊이 우선 순서에 의존하고 있다는 점에서 여러 가지 문제점이 생김을 알 수 있다. 복잡한 CFG들에 대해서 여러 가지 문제점들이 생길 수 있으나 근본적 형태는 위에서 알아 본 2가지 형태로 요약될 수 있다. 이와 같은 현상으로 인해 중복되는 계산도 많아지게 된다. 또한 일부분의 정보 변화를 위해서 코드 전체를 읽어가므로 정보가 변하지 않아도 되는 단위구역들을 다 방문하여야 하는 오버헤드도 발생함을 알 수 있다.

4. 새로운 진행 방식의 Live Variable Analysis 알고리즘

앞에서 알아본 기존 LVA 알고리즘의 단점들을 보완하기 위해 본 논문에서 제안한 새로운 진행 방식의 LVA 알고리즘에 대해 소개하고자 한다. 계산 진행 방식을 바꿈으로써 입력되는 코드 전체에 대한 반복이 아닌 부분부분의 정보 완성을 먼저 한다는 의미에서 이름을 "비반복적 LVA 알고리즘"이라고 하겠다. 부분에 대한 정보를 완성하는 방법은 반복적인 구조를 가지지만 코드 전체적으로 보았을 때는 "값이 변하지 않을 때까지 반복하기"라는 모습을 없앴으므로 이와 같은 이름을 붙여 보았다.

4.1. 기본 아이디어

새로운 LVA 알고리즘의 핵심 아이디어는 다음과 같다. 기존의 알고리즘에서 정보가 직접 전달되는 범위를 넓히는 것이다. 바로 직접 붙어 있는 단위구역들 간에서만 *IN*, *OUT*의 영향력으로 그 값을 구하는 대신 하나의 사용된 변수가 변화를 주어야 하는 모든 *OUT*들을 한 번

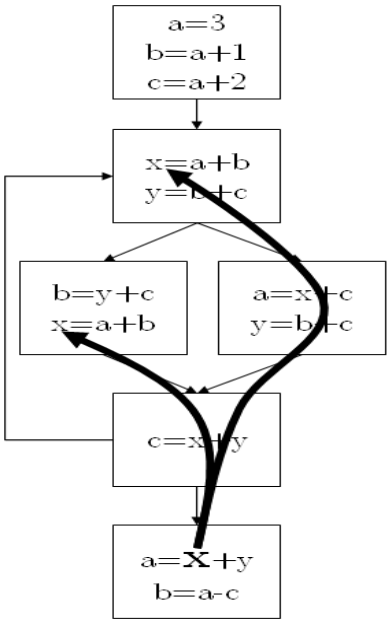
에 찾아 올라가면서 모두 고쳐 준다.

지금까지는 *OUT*을 고치기 위해 연결된 단위구역들만을 둘러보았다. *OUT*에 영향을 주는 것을 그 후계단위구역들을 *IN*으로 규정. 자신의 *OUT*을 갱신하기 위해 *IN*들을 합하였다. 즉, 알고리즘 진행의 주체가 *OUT*이었다고 할 수 있다.

*OUT*이 아닌 *USE*의 입장에서 생각을 해 보자. *IN*의 초기값은 *USE*에서 출발한다. 단위구역의 입구 부분에서 살아 있어야 하는 변수들의 기본은 그 단위구역에서 사용되고 있는 변수들의 집합이기 때문이다. 이 사용되고 있는 변수들은 자신의 값이 정의되어 있는 단위구역까지의 경로들에서 모두 살아 있어야 한다. *USE*에 속한 변수에 대해서 기준을 가지고 수행을 하는 것이다. 현 단위구역의 *USE*에 포함되어 있는 변수들에 대해서 프로그램의 흐름을 거꾸로 올라가면서 자신이 정의되어 있는 곳까지 모든 경로를 따라 찾아 올라가는 동안 그 경로에서 만나는 모든 단위구역의 *OUT*에 그 변수를 추가시켜 주면 LVA가 수행된다.

다시 정리해 보면, 프로그램 전체를 스캔하면서 점증적인 진행을 하는 것이 아니라, 추가해야 하는 항들만을 추가시켜 주는 방식이다. *USE*에 포함되어 있는 변수들은 그 변수가 정의되어 있는 곳에서부터 그 변수가 사용되고 있는 바로 그 위치에 이르게 되는 모든 경로 상에서 그 값이 기억되어 있어야 한다. 이는 그 변수가 살아있어야 함을 뜻하는 것이고, 그 경로를 변수가 사용된 위치에서부터 거꾸로 찾아 올라가면서 *OUT*에 그 변수를 추가시켜주면 된다.

그림 7에서 간단한 예를 들고 있다. 사용되고 있는 변수 *x*가 자신의 값이 정의되어 있는 단위구역까지 찾아 올라가는 과정을 나타내고 있다.



(그림 7) 기본 아이디어의 수행 과정

4.2. 비반복적 알고리즘

앞에서 알아 본 새로운 알고리즘에 대한 기본적인 생각들을 그대로 구현을 하자면 실행 상에서 단점이 있다. 각 단위구역들에서 매번 따로 *USE*를 사용하여 거슬러 올라가는 작업을 반복해야 한다는 것이다. 즉, 매번 단위구역마다 사용된 변수들로부터 출발을 반복한다는 것이다.

이러한 점을 보완하기 위해 부분적으로 수정하였다. 아래쪽에서 *USE*를 통해서 거슬러 올라가면서 거쳐가는 경로에서의 사용된 변수들도 가지고 올라가는 것이다. 경로를 타고 올라가면서 자신의 정의를 찾아감과 동시에 같이 올라가야 하는 사용된 변수들을 같이 데리고 올라가는 것이다. 이와 같은 작업을 통해 기본 아이디어가 동작하는 것을 좀 더 효율적으로 하게 해 주었다. 결국 합집합을 하면서 올라갈 것을 여러 번에 나누어서 하지 않고 한 번에 모아서 처리하게 함으로써 3장에서 언급한 기존 점증적 DFA 방법론을 따르는 LVA 알고리즘의 단점 중 하나인 거대한 집합들의 반복된 합집합 연산에서 오는 부담감을 여기에서도 줄여 주게 된다. 알고리즘의 의사코드는 그림 8에 나타나 있다.

```
procedure NewLVA
  subprocedure
        OutUpdate(B:basic block,
                  UseSet: variable set)
    for each predecessor P of B do
      UseSet = UseSet − OUT(P)
      if UseSet ≠ φ do
        OUT(P) = OUT(P) ∪ UseSet
        UseSet = UseSet − DEF(P)
        if P.visited ≡ FALSE do
          UseSet = UseSet ∪ USE(P)
          P.visited = TRUE
        end
        if UseSet ≠ φ do
          OutUpdate(P, UseSet)
        end
      end
    end
  end of subprocedure

  for each basic block B do
    B.visited = FALSE
  end
  for each basic block
        B in reverse DFS order do
    if B.visited ≡ FALSE do
      OutUpdate(B, USE(B))
    end
  end
end of procedure
```

(그림 8) 제안한 LVA 알고리즘

4.3. 비반복적 알고리즘의 장단점

4.3.1. 장점

제안한 비반복적 알고리즘의 장점은 다음과 같다.

- OUT에 추가된 변수가 없는 단위구역은 방문하지 않는다.
- 반복문이 나열되어 있을 때 아래쪽 반복문부터 OUT을 완성해 가면서 진행하여 완성되지 않은 정보로 위쪽 반복문에 대한 연산을 진행하지 않을 수 있어 연산의 중복을 막을 수 있다[5].
- 합집합 연산을 나누어서 서서히 증가해 나가는 방식으로 하지 않고, 정보를 모아 한 번에 하게 되어 해야 하는 연산의 횟수, 그리고 합집합 시 관리해야 하는 집합들의 크기를 줄인다.

먼저 코드 전체에 대한 반복적 실행을 하지 않으므로 기존의 방식 때처럼 정보가 변하지 않아도 되는 단위구역을 방문해야 하는 일이 없어졌다. 정보가 변경되어야만 하는 곳을 재귀적 구조로 쫓아감으로써 변경되지 않는 곳에는 가지도 않게 된 것이다. 이는 전체 코드를 다 읽는 것이 알고리즘 진행의 기본 단위인 비효율적 구조를 없애는 큰 효과를 가져왔다.

다음으로 반복문 구조에서 반복문의 정보를 완성한 다음에 정보의 진행을 하게 되는 장점이 있다. 반복문에서 반복문 방향으로 가는 곳을 먼저 계산하게 된다면 이에 대한 완성을 한 후 다음 진행을 하게 되므로 기존 알고리즘의 단점에서 나타나는 효과를 없앨 수 있게 된다.

마지막으로 추가해 줄 정보들 중에서 이미 추가되어 있을 정보들은 버리면서 진행하므로 합집합 연산 시 양쪽 집합이 모두 거대해지는 현상을 방지한다. 기존 알고리즘에서는 자신이 가지고 있는 모든 정보를 매번 반복적으로 전달하므로 이미 집합이 커져버린 상태의 OUT과 IN

5) 항상 막는 것이 아니라 막는 경우도 있고, 기존 알고리즘과 비슷하게 움직여 버릴 수도 있다. 이에 대한 자세한 내용은 4.3.2절을 참조하기 바란다.

의 합집합 연산이 반복된다. 하지만 새로운 진행 방식에서는 추가해 줄 정보를 가지고 올라가면서 이미 아래쪽에서 추가된 정보들은 제거를 하면서 올라가므로 그 위에서는 미리 계산에서 포함된 항들을 계속 합집합 연산하는 일을 없애주는 것이다. 이는 합집합 연산의 횟수 자체를 줄이는 것은 아니지만 합집합 연산 하나하나의 오버헤드를 줄이는 효과를 가져와 진행 속도를 빠르게 한다.

(표 1) basic block 방문 횟수 비교

프로그램 종류	basic block 방문 횟수		new/old (%)
	new	old	
adpcm	3,152	5,980	52.7
epic	49,883	79,025	63.1
g721	15,658	23,433	66.8
ghostscript	2,117,698	3,351,949	63.2
jpeg	189,054	340,781	55.5
mesa	602,098	995,743	60.5
mpeg2	132,940	286,311	46.4
pegwit	33,863	63,948	53.0
pgp	267,015	508,172	52.5
rasta	61,116	76,234	80.2
gsm	31,979	68,739	46.5
비율의 평균	-	-	**58.2**
총합의 비율	**3,504,456**	**5,800,315**	**60.4**

표 1[6]은 두 알고리즘이 수행되면서 단위구역을 방문하는 횟수를 비교한 결과이다. 제안한 알고리즘이 방문하는 단위구역의 수가 훨씬 적음을 알 수 있다. 제안된 알고리즘의 방문 횟수는 기존 알고리즘의 방문 횟수의 약 59% 정도이다.

표 2는 LVA를 수행하면서 기존의 알고리즘과 새로이 제안한 알고리즘이 합집합을 수행하는 횟수와, 합집합 수행시 대상이 되는 두 집합의 크기를 합해 나간 누적값을 나타내는 것이다. 표 2에서 볼 수 있듯이 대부분의 경우 합집합 연산의 횟수도 줄어들고, 합집합 시 관리해야 하는 집합의 크기도 작음을 알 수 있다. 횟수가 커지는 경우가 있는데, 이는 거슬러 올라가면서 *USE*들을 합하면서 한 번에 올라가기 위해 하는 작업이 수행하는 합집합들도 포함되기 때문이다. 이 합집합은 단지 자신의 단위구역에서의 *USE* 집합을 보면 되므로 집합의 크기가 작아 부담감이 적은 편이다. 또한 집합 항의 개수 합에서 총합의 비율은 100%를 넘어가는데 이는 프로그램 *rasta*에서의 큰 차이가 전체에 큰 영향을 끼치어서 그렇게 되었다. 프로그램 *rasta*에 있는 코드들의 특성이 얼마나 차이가 많이 난 것인지, 아니면 다음 절에서 말할 단점이 유난히 부각된 것인지는 정확히 알 수 없었다. 실제 실

6) 총합의 비율과 "비율의 평균에 대해서는 [1]에서 자세히 설명하고 있다. 여기서 간략히 언급하자면 총합의 비율은 각각의 성능평가 프로그램 실행 결과의 합을 한 후 총합으로 비교 대상들 간의 비율을 낸 것이고, 비율의 평균은 각 프로그램들마다 비교 대상들 간의 비율을 계산한 다음 그 비율들의 평균을 낸 것이다.

(표 2) 집합 연산을 하는 횟수와 집합 연산 시 사용되는 집합들 항의 개수 합

프로그램 종류	집합 연산 횟수		new/old (%)	참여 집합 항 개수 합		new/old (%)
	new	old		new	old	
adpcm	2,235	3,218	69.5	707,904	1,097,600	64.5
epic	32,562	35,810	90.9	14,931,648	17,329,824	86.2
g721	10,118	12,621	80.2	3,894,976	4,623,744	84.2
ghostscript	1,387,365	1,533,002	90.5	865,804,320	729,641,344	118.7
jpeg	120,909	171,635	70.5	42,655,968	62,500,640	68.3
mesa	389,970	489,896	79.6	205,974,592	228,940,384	90.0
mpeg2	82,947	131,821	62.9	31,496,160	45,852,512	68.7
pegwit	21,257	29,813	71.3	7,831,776	9,331,840	83.9
pgp	172,833	223,438	77.4	71,829,472	77,835,488	92.3
rasta	36,907	35,362	104.4	16,108,224	12,875,520	125.1
gsm	21,254	31,871	66.7	7,580,512	10,294,144	73.6
비율의 평균	-	-	**78.5**	-	-	**86.9**
총합의 비율	**2,278,357**	**2,698,487**	**84.4**	**1,268,815,552**	**1,200,323,040**	**105.7**

험 결과에서는 집합들의 크기가 크게 나온 프로그램들도 속도는 더 빠르게 나오는 것을 5장에서 알 수 있을 것이다.

4.3.2. 단점

4.3.1절에서 2번째로 언급한 장점에 대해서 다시 고찰하여 보자.

반복문에 대한 *OUT*값을 먼저 안정화시킨 다음 그 정보가 필요한 곳으로 *OUT*값을 전파시키는 것이 알고리즘의 장점인데, 이와 같은 동작을 하지 않을 수도 있다는 것이다.

현재 제안한 알고리즘에서는 CFG에서 어느 방향이 반복문을 형성하는 쪽인지 구분하지 않고 아무 것이나 선택하게 되어 있다. 그러므로 아래쪽 반복문부터 먼저 계산을 하고 올라가면 다행이지만, 먼저 작업을 할 단위구역을 잘못 선택하게 되면 이와 같은 장점이 사라지게 되는 것이다.

먼저 가야 하는 경로에 대한 정보가 없으므로 항상 기대하는 대로 작동한다고 할 수는 없다는 것이 단점이다. 분기문이 있는 경우 점프하는 곳을 먼저 가 본다는 정도의 시도는 해 볼 수 있지만, 먼저 갈 방향을 매번 결정하는 것도 성능에 부담을 주는 행위이므로 장단점이 있을 수 있다.

그리고 *USE* 집합들을 합쳐 올라가면서 *OUT*에 포함된 값들은 지우고, 방문한 단위구역의 *USE* 집합을 다시 합쳐서 올라가는 등. 부수적인 작업들이 필요하다는 점이 단점으로 작용할 수 있다. 기존의 알고리즘도 루프나 분기문 등으로 코드가 크게 나뉘지 않는다면 반복이 그리 많지 않을 것이므로 이와 같은 부수적 작업에 의한 부담(overhead)이 더욱 크게 두드러질 수 있다.

4.4. 새로운 알고리즘의 안정성 증명

기존 점증적 DFA 방법 알고리즘과 그 결과가 같음을 [1]에서 증명해 두었다.

[*Theorem*] 기존 점증적 LVA 방법 알고리즘과 비반복적 LVA 알고리즘은 그 결과가 같다.

자세한 증명은 [1]에 있으므로 여기서는 생략

한다. 대신 간략하게 그 아이디어를 보이자면 다음과 같다. 기존의 알고리즘은 직접 연결된 단위구역들간에만 정보를 주고받으면서 이를 반복하여 하나의 단위구역에서 변경된 정보가 영향력을 끼치는 전체 영역으로 점점 그 전달 영역을 확장시킨다. 이와 같이 기존의 알고리즘이 직접 연결된 단위구역들 사이에서만의 작용을 반복함으로써 이루는데 반해 비반복적 알고리즘은 영향력이 끼칠 전체 영역을 직접 한 번에 찾아간다는 차이점을 가진다. 추가해 줄 정보를 들고 가면서 더 이상 추가될 필요가 없을 때까지(모두 다 KILL에 의해 지워질 때까지) 계속 올라가면서 정보를 추가해 주는 것이다.

정보가 변한 단위구역이 있으면 계속 반복을 해 나가면서 정보가 전달되지 않는 경우를 피하는 기존의 방식 대신에 직접 전달되어야 하는 곳을 다 가 버림으로써 그러한 확인 절차가 필요 없게 되는 차이점만 있는 것이다. 정보를 전달하는 것은 같으나 그 진행 방식만이 다르다는 것. 기존의 알고리즘이 바로 인접한 것들만을 비교하는 버블정렬(bubble sort)와 같은 방식이라면 새로운 알고리즘은 하나하나가 자신의 영역을 완성하면서 나가는, 즉 하나하나가 자신의 위치를 정해가면서 진행하는 삽입정렬(insert sort)와 같다고 할 수 있다.

5. 구현 및 실험 결과

5.1. 실험 환경

구현 및 실험 환경은 다음과 같다. 새로이 개선한 알고리즘과 기존의 알고리즘을 Zephyr 컴파일러에서 C언어로 구현하여 그들간의 성능을 비교해 보았다. 실험을 한 환경은 SUN Blade 1000 workstation system으로 CPU 750MHz, RAM 512MB, Cache 8MB, 그리고 사용 OS는 Solaris 8이다. 성능평가 프로그램으로 MediaBench를 사용하였다.

5.2. 실험 결과

새로운 계산 순서의 적용을 LVA에 대해서 하였으므로 실험은 이를 위주로 진행하였다. 자세한 실험 결과는 [1]에 자세히 나타나 있다. 자세한 설명을 위해서는 [1]을 참고하기 바란다. 여기에서는 간략히 결과만을 언급하기로 하겠다.

결과는 아래의 2가지로 요약된다.

- **전체 컴파일 과정에서 LVA가 차지하는 비율**
- **기존의 LVA와 새로운 LVA 알고리즘의 수행 속도 비교**

5.2.1. LVA가 전체 컴파일 과정에서 차지하는 비율

Zephyr에서의 실험 결과, 다른 최적화 기법들이 1번씩 실행될 때 LVA는 평균 4.7번 실행되었다. 실험한 모든 성능 평가 프로그램들이 골고루 4.7에 가까운 수치를 보여주는 것도 알 수 있었다. 이는 LVA의 복잡도는 단일 실행의 4배에서 5배를 한 후 다른 최적화 기법들과 비교해 주어야 한다는 주장에 더욱 확신을 주고 있다. 그러면 실제 전체 컴파일 수행 시간에서 LVA가 차지하는 시간의 비율을 알아보자.

표 3에서 사용한 LVA 알고리즘은 기존 DFA 방법론을 따른 알고리즘이다. 기존의 컴파일러 그 자체에서 LVA가 차지하는 비율을 알아보고자 한 것이다. 한 번의 수행 시간은 그리 길다고 할 수 없는 LVA가 수행 횟수가 많아지면서 전체에서 차지하는 수행 시간의 비율이 7%를 넘어 서고 있음(총합의 비율은 7.0%, 비율의 평균은 7.5%)을 알 수 있다. 여기에서는 LVA만을 조사하였지만 만약 모든 DFA에 소요되는 수행 시간을 조사하였다면 이의 몇 배가 되는 결과가 나올 것은 당연하다고 할 수 있다.

(표 3) 전체 컴파일 수행 시간 중 LVA에 소요되는 시간의 비율

프로그램 종류	LVA(s)	Total(s)	LVA/Total(%)
adpcm	0.016	0.398	4.0
epic	0.341	3.818	8.9
g721	0.085	1.126	7.5
ghostscript	11.440	177.355	6.5
jpeg	1.310	18.805	7.0
mesa	3.243	49.160	6.6
mpeg2	1.056	11.506	9.2
pegwit	0.284	3.526	8.0
pgp	2.141	16.669	12.9
rasta	0.358	4.564	7.8
gsm	0.266	6.154	4.3
비율의 평균	-	-	**4.7**
총합의 비율	**20.540**	**293.081**	**4.6**

5.2.2. 두 알고리즘의 수행 속도 비교

본 논문에서 제시한 비반복적 LVA 알고리즘과 기존 깊이 우선 순서를 사용하는 점증적 방법 알고리즘간의 수행 시간을 비교하였다. Zephyr 컴파일러에 기존에 구현되어 있는 LVA 알고리즘이 *DEF*, *USE*를 구하는 부분도 LVA 알고리즘에 포함되어 있는 점 때문에 시간 측정에서 LVA 알고리즘의 시간은 *DEF*, *USE*를 구하는 부분도 포함된 값이다. 그리고 기존의 알고리즘에서 새로운 알고리즘으로의 개선 %를 구할 때는 기존의 알고리즘을 old, 비반복적 알고리즘을 new라고 하면 $\frac{old-new}{old} \times 10$으로 구하였다.

(표 4) 두 가지 LVA 알고리즘의 수행 시간 비교

프로그램 종류	LVA 수행시간(s)		LVA 개선(%)	전체 컴파일 수행시간(s)		Total 개선(%)
	new	old		new	old	
adpcm	0.008	0.016	50.0	0.390	0.398	2.0
epic	0.149	0.341	56.3	3.626	3.818	5.0
g721	0.018	0.0845	79.3	1.059	1.126	6.0
ghostscript	6.958	11.440	39.2	172.873	177.355	2.5
jpeg	0.860	1.310	34.4	18.355	18.805	2.4
mesa	2.729	3.243	15.9	48.646	49.16	1.1
mpeg2	0.655	1.056	38.0	11.105	11.506	3.5
pegwit	0.172	0.284	39.5	3.414	3.526	3.2
pgp	1.279	2.141	40.3	15.807	16.669	5.2
rasta	0.141	0.358	60.6	4.347	4.564	4.8
gsm	0.103	0.266	61.3	5.991	6.154	2.7
비율의 평균	-	-	**46.8**	-	-	**3.4**
총합의 비율	**13.072**	**20.540**	**36.4**	**285.613**	**293.081**	**2.6**

실험 결과는 표 4와 같다. 전체 컴파일 수행 시간에서 7% 정도를 차지하던 기존의 알고리즘과는 달리 새로운 LVA 알고리즘은 전체 컴파일 수행 시간에서 총합의 비율 4.6%, 비율의 평균 4.2%를 차지하고 있다. 이는 전체 컴파일 수행 시간에서의 개선치가 약 2%~4%가 됨(총합의 비율 2.6%, 비율의 평균 3.5%)을 의미한다. LVA 자체 성능 개선을 본다면 40%에 육박하는(총합의 비율 36.4%, 비율의 평균 46.8%) 수준이다. 기존의 알고리즘에서 수행 시간을 3분의 1 이상 줄이는 효과를 가져왔다.

6. 결론 및 향후 연구 과제

6.1. 결론

기존의 점증적 DFA 방법론을 충실히 따라서 LVA를 수행하는 알고리즘은 여러 가지 단점이 있다. 실행 순서를 단순히 깊이 우선 순서로 정하면서 직접 연결된 단위구역들만 서로 영향을 주는 방식에 의해 결과값이 서서히 증가함에 따른 비효율적인 움직임 등이 바로 그것이다. 한 부분이 변하여도 코드 전체를 읽어보아야 하는 문제점, 완성되지 않은 정보를 전달함으로 인해 생기는 중복되는 계산들. 입력된 코드의 반복문이나 분기문 부분에서 이와 같은 점들이 발생하게 된다.

이를 해결하여 좀 더 빠른 LVA 수행을 위해 새로운 알고리즘을 고안하였다. 기존 알고리즘에서 인접한 단위구역들 간에만 영향력을 주면서 이와 같은 계산을 무조건 반복한다는 점이 비효율성의 가장 큰 원인이므로 이를 개선하고자 진행 방식을 바꾼 것이다. *OUT*에 변화를 가져올 변수들에 대해 그 변수가 *OUT*에 추가되어야 하는 모든 경로를 직접 바로 따라 가면서 해당 변수가 프로그램의 *OUT*들에 줄 영향력을 한 번에 마무리 짓는 것이다. 이와 같이 진행 방식을 바꿈으로써 방문하여야 하는 단위구역의 수도 줄어들었고, 교점 연산자인 합집합 연산 시 그 연산의 대상이 되는 집합들의 크기에 의한 부담도 줄일 수 있었다.

물론 이와 같은 진행도 문제점이 있다. 진행할 방향이 여러 가지인 경우 어느 곳을 먼저 진행하겠다는 정보가 전혀 없이 아무 것이나 선정한다는 점이 원래 의도한 바의 움직임을 보이지 않게 할 수 있다는 것이다. 반복문의 정보를 먼저 완성하고 진행하고자 하는 의도와는 달리 어느 방향이 반복문인지 모르기 때문에 반복문의 정보를 완성을 하지 않은 채 다른 방향으로 계산을 먼저 진행시켜 나갈 수도 있다는 점이다.

하지만 이와 같은 문제점에도 불구하고 기존 점증적 LVA 방법보다 총합의 비율에서 36.4% 빨라졌고. 비율의 평균에서는 46.8% 빨라졌다[7]. 이는 전체 컴파일 수행 시간을 총합의 비율에서 2.6%, 비율의 평균에서 3.4% 빨라지는 효과를 가져왔다.

6.2. 향후 연구 과제

기존 점증적 DFA 방법론은 깊이 우선 순서에 의존하고 있다. 이 논문에서는 이에 대한 개선안을 하나 제시해 본 것이다. 하지만 아직 LVA에만 적용한 제한성이 있으므로 이를 전체 DFA로 넓혀 나가는 것이 가장 중요하다고 하겠다.

현재의 모양 그대로를 DFA 전체로 확장하는 데는 몇 가지 문제점이 있다. 첫째. 소개한 알고리즘의 단점에서 말한 진행 방향을 정하지 않은 것을 해결해야 한다.

둘째. 지금의 알고리즘은 LVA에 치중하다 보니 DFA의 전체적인 특징을 반영하지는 못 하

7) Zephyr의 기존 구현 때문에 LVA 알고리즘 자체의 시간 측정에서 *DEF, USE*를 구하는 부분도 포함하여 측정하였다. 이런 점에서 실제로 *OUT*만을 구하는 부분에서는 개선 비율이 더욱 클 것으로 추정된다.

고 있다. 예를 들어 유효식 찾기의 경우, 여러 정보가 만날 때 교집합을 한다는 차이점이 있다. 한 방향씩 먼저 다 진행을 하는 지금의 알고리즘을 적용할 경우 추가하는 연산을 다 한 다음 방향에서 다시 같은 경로를 따라가면서 다시 다 결과에서 제거해 주는 진행이 일어날 수 있다. 중복된 연산을 없애려 하다가 오히려 중복된 연산을 생성할 수 있다는 것이다.

셋째. 여러 개의 단위구역들로부터 정보를 받는 단위구역의 경우 자신이 받을 정보들을 다 받은 후에 진행하는 것이 가장 이상적이다. 하지만 지금의 경우 반복문에 집중을 하다 보니 여러 정보가 모이는 곳에서 다른 정보들이 오기까지 기다리지는 않는다. 물론 기존 알고리즘이 깊이 우선 순서도 이러한 성질을 만족하지 못한다는 게 문제점이다. LVA에 적용함에 있어서는 추가하여 줄 정보들이 이미 추가되어 있는지 확인함으로써 중복되는 계산은 피했으나 확인하는 과정이 필요하다는 것도 오버헤드로 작용한다. 이러한 문제점을 해결하여야 중복되는 연산을 최소화할 수 있다.

점증적 DFA 방법론의 발전 역사상 가장 문제점은 추가적인 데이터 구조의 도입을 꺼린다는 것이라고 생각한다. 추가적인 데이터 구조의 도입 자체가 부담이 될 수 있다고 생각할 수 있지만 이는 DFA 그 자체만을 보기 때문에 할 수 있는 오해다. 컴파일 전체를 두고 생각한다면 DFA를 수행하는 그 순간에 이미 사용 가능한 정보들이 많이 있을 수 있다는 것이다. 추가적으로 만들지 않아도, 추가적으로 만든다고 해도 결국 나중에 만들어야 할 것의 생성 시점을 조금 일찍 가져가는 것만이 되는 경우가 많다. 예를 들어 컴파일 최적화 기법들을 적용해 나가기 위해서는 루프 구조 파악이나 깊이 우선 검색 트리 등은 이미 만들어져 있거나 만들어야 할 것이다. 새로이 만든 알고리즘에서 문제점으로 제시된 "어느 방향이 반복문 구조인지 몰라서 그 쪽부터 먼저 실행하지 못 하는" 것도 이와 같은 정보를 이용한다면 좀 더 쉽게 해결할 수도 있을 것이다.

분기문이나 반복문에서의 계산 우선순위를 잘 선택하기 위해 사용할 수 있는 정보들을 수집하여 잘 활용함으로써 그 우선순위를 정할 수 있다면 DFA의 수행 시간 단축에 큰 도움을 줄 것이라 생각한다.

그러므로 지금까지의 생각들을 잘 정리한 다음 일반적 컴파일러에서 DFA를 수행할 때 얻을 수 있는 정보를 파악하고, 이를 잘 활용하여 새로운 계산 순서를 정하는 데 에 관한 연구가 계속 진행되어야 할 것이다.

참 고 문 헌

[1] 윤정한, *Control Flow Graph상의 USE 집합을 중심으로 하여 Live Variable Analysis를 수행하는 비반복적 알고리즘*. 한국과학기술원, 석사학위논문, 2003.

[2] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. In *Acta Informatica*, 7(3) p305~317, July 1977.

[3] Susan L. Graham, Mark N. Wegman. A Fast and Usually Linear Algorithm for Global Flow Analysis. *JACM* 23(1), p172~202, 1976.

[4] M. S. Hecht, *Flow Analysis of Computer Programs.* Elsevier North-Holland, Amsterdam. 1977.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986

[6] C. Samuel Hsieh. A fine-grained data-flow analysis framework. In *Acta Informatica*, 34(9) p653~665, August 1997.

[7] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. In *ACM Computing Surveys*, Vol. 18, No. 3, September 1986.

[8] Vugranam C. Sreedhar, Guang R. Gao and Yong-fong Lee. A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs. In *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 2, p388~435, March 1998.

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the Cost of Data Flow Analysis By Congruence Partitioning. In *International Conference on Compiler Construction*, April 1994.

[10] Michael P. Gerlek, Michael Wolfe, and Eric Stoltz. A Reference Chain Approach for Live Variables. Technical Report CSE 94-029, Oregon Graduate Institute, April 1994.

[11] Glenn Ammons and James R. Larus. Improving Data-flow Analysis with Path Profiles. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, p72~84, 1998.

[12] Wankang Zhao, Baosheng Cai, et al. VISTA: A System for Interactive Code Improvement. In *Proceddings of Language, Compilers, and Tools for Embedded Systems*, 2002.

[13] *Zephyr* Homepage http://www.cs.virginia.edu/zephyr/ May 1999.

[14] MediaBench Homepage http://www.cs.ucla.edu/~leec/mediabench

윤 정 한

1997년~2001년 한국과학기술원 전산학과(학사)
2001년~2003년 한국과학기술원 전자전산학과 전산학전공(석사)
2003년~현재 한국과학기술원 전자전산학과 전산학 전공 박사 과정

관심분야: 프로그래밍 언어, 컴파일러.

한 태 숙

1972년~1976년 서울대학교 전자공학과(학사)
1976년~1978년 한국과학기술원 전산학과(석사)
1990년~1995년 Univ. of Noth North Caroina at Chapel Hill(박사)
1996년~현재 한국과학기술원 전자전산학과 부교수

관심분야: 프로그래밍 언어론, 함수형 언어