사용할 변수의 예측에 사용되는 반복적 알고리즘의 계산순서 재정렬을 통한 수행 속도 개선

(Improvement of Iterative Algorithm for Live Variable Analysis based on Computation Reordering)

윤 정 한 † 한 태 숙 ††
(Jeong-Han Yun) (Han Taisook)

요 약 기존의 LVA를 수행하는 알고리즘은 반복적 정보흐름분석(Iterative Data Flow Analysis - DFA) 프레임워크에 따라 프로그램 전체를 반복적으로 스캔하면서 진행되어진다. *Zephyr*[1] 컴파일러의 경우 이와 같은 반복적 알고리즘으로 LVA를 수행하는 시간이 전체 컴파일 시간에서 약 7%를 차지하고 있다.

기존 LVA 알고리즘은 여러 가지로 개선할 점들이 있다. LVA를 수행하는 기존의 반복적 알고리즘은 알고리즘의 특성상 방문하지 않아도 되는 basic block들에 대한 방문이 잦고, 살아있는 변수들의 집합을 점차적으로 증가해 가면서 구하는 특성상 큰 변수들의 집합에 대한 연산을 계속 하게 된다.

우리는 기존의 알고리즘과 달리 사용된 변수들(USE set)에 대해 Control Flow Graph(CFG)에서 거슬러 올라가면서 LVA를 수행하는 반복적인 알고리즘의 개선안을 제안하고자 한다. 이는 기존의 알고리즘과 같은 결과를 내면서 더 빠른 알고리즘이다. DFA에서의 flow equation을 적용하는 순서를 바꿈으로써 많은 중복 계산을 줄일 수 있다. 이러한 방법으로 인해 basic block을 방문해야만 하는 횟수를 줄이면서 전체 수행 시간을 단축시킨다. 간단한 추가 구현만으로 *Zephyr* 컴파일러에서의 실험 결과에서 LVA만을 수행하는 시간에서 기존의 알고리즘보다 36.4% 짧은 시간을 사용하였고, 이는 전체 컴파일 시간을 2.6% 줄이는 효과를 가져왔다.

키워드 : 컴파일러, 최적화, 컴파일 시간, live variable analysis, data flow analysis

Abstract The classical approaches for computing Live Variable Analysis(LVA) use iterative algorithms across the entire programs based on the Data Flow Analysis framework. In case of *Zephyr* compiler, average execution time of LVA takes 7% of the compilation time for the benchmark programs.

The classical LVA algorithm has many aspects for improvement. The iterative algorithm for LVA scans useless basic blocks and calculates large sets of variables repeatedly.

We propose the improvement of iterative algorithm for LVA based on used variables' upward movement. Our algorithm produces the same result as the previous iterative algorithm. It is based on use-def chain. Reordering of applying the flow equation in DFA reduces the number of visiting basic blocks and redundant flow equation executions, which improves overall processing time. Experimental results say that our algorithm can reduce 36.4% of LVA execution time and 2.6% of overall computation time in *Zephyr* compiler with benchmark programs.

Key words : compiler, optimize, compile time, live variable analysis, data flow analysis

1. 서 론

1.1 연구 배경

Live Variable Analysis(LVA)는 정보흐름분석(Data Flow Analysis - DFA)의 하나로써 프로그램의 각 위치에서 어떤 변수들의 값을 기억하고 있어야 하는지 알아보는 분석기법(analysis)이다. LVA를 수행하여 알아

· 본 연구는 대학 IT연구센터 육성 · 지원사업의 연구결과로 수행되었음
† 학생회원 : 한국과학기술원 전산학과
dolgam@compiler.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
han@cs.kaist.ac.kr
논문접수 : 2003년 5월 19일
심사완료 : 2005년 6월 2일

낸 정보는 Dead code elimination, Code motion 등 많은 최적화 기법 및, Interference graph 만들 때 등 컴파일 수행 전반에 걸쳐 두루 사용된다. LVA가 뽑아내는 정보를 필요로 하는 최적화 기법들이 많아서 컴파일 시 LVA는 여러 번 수행된다.

Just-in-time(JIT) 컴파일러, Interactive Development Environment(IDE), Dynamic compilation system에서는 컴파일 속도가 빠른 것이 중요하다. VISTA [2]의 경우 Interactive Compilation이라는 개념을 제시하고 있다. 최적화 기법들의 적용 순서를 사용자가 이리저리 바꿔 보면서 컴파일 하고자 하는 프로그램에 가장 적당한 최적화 기법 적용 순서를 찾아가게 해 주는 것인데, 이를 위해서는 수많은 시행착오가 필수이다. 사용자가 다양한 가능성을 실험해 보기 위해서는 빠른 컴파일 속도가 중요시 된다.

본 논문에서는 컴파일 시간을 줄이기 위해 최적화 기법 자체가 아닌, 최적화 기법을 수행하기 위한 준비단계의 속도를 향상시키는 방법을 제시하였다. LVA가 한 번 수행되는데 드는 시간은 컴파일 하는 전체 시간에서 보았을 때 그리 큰 비중을 차지하지 않을지도 모른다. 하지만 여러 최적화 기법들이 LVA의 결과를 사용하고 있다. 또한 각각의 최적화 기법이 적용될 때마다 프로그램 코드가 변경되게 되고, 정확한 정보를 위해서는 코드에 변경이 있을 때마다 LVA를 다시 수행하여야 한다. 그러므로 하나의 프로그램을 컴파일 하는 동안 LVA가 수행되어야 하는 횟수가 많아져 컴파일 시간 전체에서 차지하는 비중이 높아지게 된다. LVA가 구하는 정보를 그대로 얻으면서 기존의 방식보다 더욱 빠른 알고리즘은 컴파일 시간 단축에 큰 도움이 될 수 있다는 것이다.

기존의 LVA 알고리즘을 분석하여 이것이 수행되는 과정에서의 불필요한 동작들을 알아보고 이러한 비효율적인 점들을 개선할 방법을 찾아보고자 한다.

1.2 문제 제기와 해결책

LVA를 수행하는 기존의 방법은 반복적 DFA 프레임웍(Iterative DFA framework[6])을 따르는 알고리즘이다. 이와는 달리 시간 복잡도가 1차인 T1-T2 변환 알고리즘[7]도 있지만, 이는 multi entry loop와 같은 형태의 코드에서는 작동하지 않는다는 점에서 실제적으로는 잘 사용되지 않고 있다.[1] 그 외에 DFA의 속도 향상을 위한 많은 알고리즘들이 나와 있지만 구현상의 어려움으로 실제 컴파일러에서는 그리 널리 사용되고 있지는 않다.

반복적 DFA 프레임웍은 알고리즘이 단순하여 이해 및 구현은 쉬우나 그리 성능이 좋지는 않다. 이와 같은 알고리즘이 실제 컴파일러에서 많이 사용되는 이유는 구현이 간단하면서도 그 반복 횟수가 실질적으로는 그리 많지 않다는 점에서다.[2]

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 반복적 DFA 프레임웍에서의 가장 큰 단점이다.

이와 같은 단점들을 해결하기 위한 본 논문에서의 기본 아이디어는 다음과 같다.

변수 v가 basic block B에서 사용되었다고 하자. v가 사용된 위치에서 그 정보를 직접 연결된 basic block들에만 알려주지 말고, 바로 그 변수가 정의되어 있는 basic block들까지 거슬러 올라가면서 그 경로에서 거쳐 가는 모든 basic block들에 v는 살아있어야 한다고 기록하는 것이다. 직접 연결되어 있는 basic block들끼리만 정보 교환을 하던 것을 정보가 필요한 하나의 체인에 직접 다 알려 주자는 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. 변경된 정보가 필요한 영역만을 좇아가면서 업데이트 해 주면 되는 것이다. 그러므로 기존 반복적 DFA 프레임웍에서 가장 큰 문제점인 "방문하지 않아도 되는 basic block의 방문"이 없어지게 된다. 또한 필요한 연산을 먼저 하게 되면서 불필요하게 반복되는 연산을 줄일 수 있게 된다. 기존 알고리즘의 비효율적인 점은 3.3절에서 자세히 다루도록 하겠다. 그리고 이를 어떻게 새로운 알고리즘이 극복하여 나가는지를 4장에서 알아보도록 한다.

1.3 논문 구성

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구들에 대한 소개로 반복적 DFA 프레임웍에 관한 내용들, 특히 DFA 프레임웍의 수행 시간 단축을 위한 기존의 연구들에 대해 중점적으로 알아본다. 3장에서는 기존 반복적 DFA 프레임웍에서의 LVA 알고리즘에 대해 정리한다. 그리고 기존 알고리즘에서의 문제점을 짚어보면서 개선의 여지를 생각해 본다. 4장은 새로이 제안한 LVA 알고리즘을 소개한다. 그리고 제안한 알고리즘의 결과가 기존의 알고리즘과 같은 정보를 추출하고 있음을 보이고, 5장에서는 두 가지 LVA 알고리즘의 성능 비교 및 분석을 하였다. 마지막으로 6장에서 본 논문의 결론과 향후 연구 과제, 특히 새로운 LVA 알고리즘을

1) T1-T2 transformation 알고리즘은 코드가 reducible할 경우에만 적용할 수 있다. 이에 대한 내용은 [7]을 참고하기 바란다.

2) 반복되는 횟수와 시간 복잡도에 대해 [8]에서 이야기하고 있다.

만든 아이디어가 앞으로 어떻게 확장 가능할 것인가에 대한 생각들의 정리를 해 보았다.

2. 관련 연구

LVA를 풀어 온 기존의 알고리즘은 반복적 DFA 프레임웍을 역순으로 LVA에 맞게 적용한 것이다. 기존의 방법 자체가 그러한 틀 속에서 발전해 오다 보니 LVA를 단독적으로 분리해서 연구하기보다 반복적 DFA 프레임웍 자체의 개선에 초점을 두고 있다. 최근에 이러한 프레임웍에서 벗어나서 Copy folding을 빠르게 해 보려는 연구도 있지만[4] 이러한 시도들이 그리 많지만은 않고 LVA는 프레임웍에 따르는 하나의 예제 정도로 처리되어 오고 있다.

2.1 Data Flow Analysis 프레임웍의 개선

DFA에 관한 프레임웍은 이전부터 소개되어 왔다[6,7,9]. CFG에서의 여러 가지 정보 추출기법들을 하나의 프레임웍으로 묶어서 수행 방법 및 그 구현을 간편화 시키려는 노력이었다. 반복적 DFA 프레임웍은 기존의 LVA 알고리즘의 근간이 되는 내용이므로 3장에서 좀 더 자세히 다루겠다.

DFA가 하나의 프레임웍으로 통일되면서 그에 포함되는 여러 가지 분석기법들에 대해 따로 연구가 진행되기보다는 이 프레임웍을 어떻게 개선하여 속도를 빠르게 할 것인가가 초점이 되어 왔다.

먼저 DFA의 프레임웍이 정보들을 서서히 증가시켜 나가면서(monotonic increase) 고정점(fixed point)에 도달할 때까지 진행하여 결론을 도출하는 시스템이라는 것을 이용한 것이 있다. 이와 같이 진행해 나가는 DFA 시스템에서 각각의 부분을 담당하는 방정식들 중 그 결과가 같은 것들을 묶어서 중복되는 연산을 줄여 속도의 향상을 가져오고자 하는 연구가 그 중 하나다[10]. 결과가 같은 방정식들로 전체 시스템을 파티션 하는 $O(n\log(n))$의 알고리즘을 만들어 전체를 나눈다. 그리하여 전체 시스템을 이루는 파티션들의 크기를 작게 하고 그로 인해 중복되는 계산을 줄임으로써 수행 시간을 단축시키고자 하는 것이다.

Reference chain[3]이 존재하는 경우 LVA를 빨리 할 수 있는 방법에 대해서도 연구가 있었다[11]. CFG에서 분기(branch)가 나타나는 곳마다 변수가 어느 곳으로 가면서 살아있어야 하는가를 reference chain을 이용해 표시해 나가면서 LVA를 수행하는 것이다. 이와 같이 하여 한 번의 스캔(one path algorithm)으로 LVA를 수행할 수 있게 된다. 이는 Single Static Assignment(SSA) form[4]에서의 성격과 비슷한 면이 있다.

앞에서 알아본 것은 이론적으로 DFA를 분석하여 개선하고자 한 노력들이다. 이와는 달리 통계적인 접근으로 시간을 단축하고자 한 연구도 있다. 결국 DFA는 프로그램에서 여러 가지 경로를 통해 계산을 하는 것이니 프로그램이 자주 수행되는 경로들을 미리 조사해서(profiling) 그 경로들을 통해 문제의 크기를 줄여 시간을 단축시키고자 하는 방법이다[12].

3. 기존 Live Variable Analysis 알고리즘

본 장에서는 LVA를 수행하는 기존 알고리즘에 대해 알아본다.

3.1 용어 및 개념 정리

LVA에 관련된 여러 가지 개념들과 용어들에 대한 정의를 정리해 보면 다음과 같다.

- 변수 v가 프로그램의 어떤 지점 p에서 살아있다(live)라고 하는 것은 p의 위치를 지나 프로그램이 진행되는 동안 적어도 한 번은 변수 v가 사용됨을 뜻한다.
- 변수 v가 프로그램의 어떤 지점 p에서 죽어있다(dead)라고 하는 것은 p라는 위치에서 변수 v가 살아있지 않다는 것을 뜻한다.
- 어떤 basic block을 B라고 할 때, 그 basic block 안에서 정의된 변수들의 집합을 $DEF(B)$라고 표현한다.

basic block B 내에서 변수값이 계산하여 대입한 변수들의 집합을 DEF(B)[5]라고 표현하겠다.

- 어떤 basic block을 B라고 할 때, 그 basic block 안에서 사용되고 있는 변수들 중 사용되고 있는 위치보다 위쪽 instruction에서 그 변수값을 정의하고 있지 않은 것들만을 모아서 $USE(B)$라고 한다.
- 어떤 basic block을 B라고 할 때, 그 블럭의 첫 부분(entry)에서 살아있는 변수들의 집합을 $IN(B)$라고 한다.
- 어떤 basic block을 B라고 할 때, 그 블럭의 마지막 부분(exit)에서 살아있는 변수들의 집합을 $OUT\ (B)$라고 한다.

IN과 OUT의 관계는 3.2절에 나타나 있다.

- LVA는 입력된 프로그램의 모든 basic block B마다 $OUT(B)$를 구하는 것이다.

3) 변수가 정의된 곳에서는 그 변수의 값이 사용된 곳들이 어디인지를 가리키고 있게 하고, 변수가 사용되고 있는 곳에서는 그 위치에서 그 변수의 값을 정의한 곳이 어디인지 가리키게 하는 체인(chain)이다. 이와 같은 체인을 만드는 과정을 Reaching Definition이라고 한다.

4) 모든 변수들에 대해 그 변수가 정의되는 것이 코드 상에서 단 한 번만 나타나는 프로그램의 형태이다.

5) 앞으로 특별히 basic block을 언급할 필요가 없는 경우에서는 DEF라고 사용하기도 하겠다. 이는 다음에서 정의하는 USE, IN, OUT에 대해서도 같다.

결론적으로 LVA는 모든 basic block들에 대해 그 곳에서 살아있어야 하는 변수들의 집합에 대한 정보를 알아내고자 하는데 그 목적이 있다.

3.2 LVA를 위한 기존의 알고리즘

전통적인 LVA를 수행하는 알고리즘은 DFA 프레임웍을 따르고 있다.

프레임웍의 구성요소들이 LVA에서 어떻게 정의되는지 알아보면 다음과 같다.

- *flow values* : $I(B)$는 3장 1절에서 정의해 놓은 $IN(B)$, $O(B)$는 $OUT(B)$이다. 즉 변수들의 집합이다.
- *meet operator* : basic block에서 자신의 predecessor들 중 한 군데의 basic block, 즉 한 노드에서만이라도 살아있는 변수는 자신의 basic block에서도 살아있어야 한다. 이는 집합들 간의 합집합을 의미하므로 meet operator는 합집합 연산이다.[6]
- *flow functions* : backward DFA의 형식이므로 $OUT(B)$를 넣으면 $IN(B)$가 나오는 함수이다. $f_B : OUT(B) - DEF(B) \cup USE(B) = IN(B)$
- flow equations : flow value들간의 관계를 나타낸다.
  $OUT(B) = \cup_{each\ successor\ S\ of\ B} IN(S)$
  $OUT(B) - DEF(B) \cup USE(B) = IN(B)$

DFA 프레임웍을 적용한 LVA 알고리즘은 그림 1과 같다.

그림 2, 3을 보면서 동작 원리에 대해 생각해 보자. 그림 2는 flow equation이 어떻게 적용되는지를 보여주는 그림이다. 아래쪽의 IN들을 모아서 OUT을 만들고, 그 OUT을 flow function에 넣어서 다시 IN을 만드는 과정을 그림으로 나타내고 있다. 굵은 화살표가 정보가 흘러가는 진행 방향을 나타내고 있다. 그림 3에서 입력된 코드에 루프가 있을 경우 진행되는 순서를 보여준다. 짙은 색으로 표현된 basic block이 자신의 IN, OUT이 바뀌는 basic block이다. 아래에서부터 Depth-first Search(DFS)의 역순으로 basic block을 방문하면서 IN, OUT을 만든다. successor들의 IN으로 OUT을 만들고, 만든 OUT을 flow function에 넣어 다시 IN을 만드는 과정의 반복이다. 입력된 프로그램에 루프가 없다면 이 알고리즘은 한 번의 스캔(그림 1에서의 for 루프를 전체로 한 번 다 수행한 것)으로 끝난다. 루프가 없다면 모든 basic block들이 자신의 successor들에게서만 영향을 받으므로 DFS의 역순으로 계산을 하면 그 계산 순서가 딱 맞아 떨어지기 때문이다. 하지만 루프가 있을 경우 DFS 순서 상 먼저 나오는 basic block이 나중에 나오는 basic block의 OUT값에 영향을 주게 된다. 이러한 경우 업데이트 된 OUT의 영향을 적용시키기 위해 다시금 그 basic block에 방문할 필요가 생기게 되는 것이다. 이런 이유로 그림 1에서 while루프가 필요하게 되고, 반복적 알고리즘의 모습을 갖추게 된다.

```
procedure OldLVA
  for each basic block B do
    IN(B) = USE(B)
  end
  while changes to any IN(B) occur do
    for each basic block B in reverse DFS order do
      OUT(B) = ∪ all successor S of B IN(S)
      IN(B) = USE(B) ∪ {OUT(B) − DEF(B)}
    end
  end
end of procedure
```

그림 1 반복적 DFA 프레임웍을 따르는 LVA 알고리즘

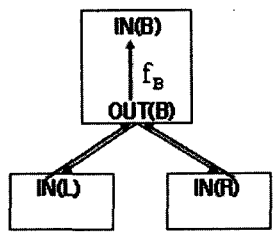

그림 2 flow equation 적용

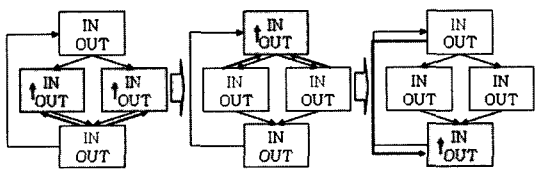

그림 3 기존의 LVA 알고리즘이 수행되는 과정

3.3 기존 알고리즘의 문제점

기존의 알고리즘은 일단 태생적으로 반복적 알고리즘이라는 단점을 가지고 있다. 본 논문에서의 실험 결과(그림 4[7] 참조) LVA 수행 시 while 루프가 평균 14.3

6) basic block B의 successor basic block이 P, Q라고 하자. $IN(P)$와 $IN(Q)$는 B의 exit에서 살아있어야 한다. 그러므로 $IN(P) \cup IN(Q) \subset OUT(B)$. 거꾸로 $OUT(B)$에 포함된 변수. 즉 B의 exit에서 살아 있어야 하는 변수들은 B가 실행된 후의 basic block의 입구에서 살아 있어야 한다. 정의가 된 곳에서부터 사용되는 곳까지 가는 모든 경로에서 살아가야 하기 때문이다. 그러므로 $OUT(B)$의 모든 변수들은 $IN(P)$ 또는 $IN(Q)$ 둘 중 하나에는 포함이 되므로 $OUT(B) \subset IN(P) \cup IN(Q)$. 즉, $OUT(B) = IN(P) \cup IN(Q)$

7) 벤치마크 프로그램 MediaBench[14]의 모든 프로그램에서 측정해 본 결과이다. Zephyr 컴파일러는 함수들 단위로 최적화를 실시하므로 벤치마크 프로그램에서의 프로그램 단위로 결과를 나누지 않는 것도 의미가 있다. 총 함수의 개수는 6515개이다. 실험 환경에 대해서는 5장에 기술해 두었다.

회 실행된다고 한다. 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.

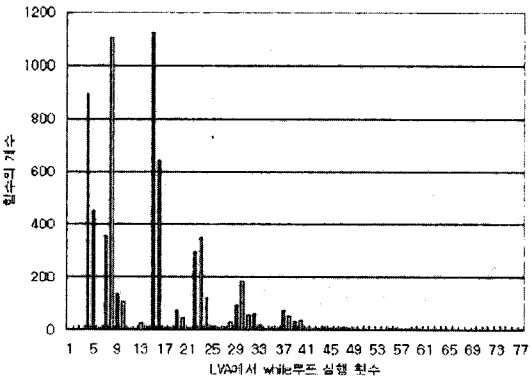

그림 4 기존 LVA 알고리즘에서 while루프가 수행되는 횟수

이와 같은 문제점 말고도 알고리즘 진행 방식에 의한 구조적인 문제점이 존재한다.

첫째, 프로그램의 전체에 대해 스캔을 하면서 OUT의 변화를 알아보고, 한 군데라도 변화가 있으면 또 다시 프로그램의 전체를 스캔하는 방식에 대한 것이다. 이와 같은 방식 때문에 결과값이 전혀 변하지 않은 basic block에도 반복적으로 방문을 하게 된다. *OUT*의 값이 첫 번째 스캔 이후로 계속 변하게 되는 곳은 프로그램에 루프가 존재하는 경우가 가장 많다. 대체로 프로그램에서 실제 루프를 이루는 코드가 길지는 않다. 코드 상으로 길지 않는 부분의 변화가 안정화 되어 결과가 나올 때까지 변화가 없는 대부분의 basic block들을 계속해서 방문하여 변화가 있었는지 없었는지 알아보는 것이 큰 비효율성을 낳고 있다.

둘째. OUT의 변화가 meet operator인 합집합에 의해 변수가 추가되어 가는 과정에서의 비효율성이다. DFA에서의 진행방법을 집합에서의 관점에서 비유해 보자면 다음과 같다. 1, 2, 3, 4, 5, 6, 7, 8, 9의 9개 항들이 흩어져 있고 이들을 합집합을 통해 {1, 2, 3, 4, 5, 6, 7, 8, 9}로 만들고자 한다. 이를 다음과 같이 만든다고 생각해 보자. {1}과 {2}를 먼저 합집합하고, 그 다음 결과물인 {1, 2}에다가 {2, 3}을 합집합. 그 결과인 {1, 2, 3}에 {2, 3, 4}를 합집합... 마지막으로 {1, 2, 3, 4, 5, 6, 7, 8}과 {2, 3, 4, 5, 6, 7, 8, 9}를 합집합 하여 {1, 2, 3, 4, 5, 6, 7, 8, 9}의 최종 결과를 얻는다고 말이다. 컴퓨터상에서 이와 같은 계산을 해야 할 경우 연산에 들어오는 집합의 크기 또한 속도에 큰 영향을 끼칠 수밖에 없다. 두 개의 집합을 입력받은 다음 서로 간에 공통적으로 들어있는 항과 한 집합에 만들어있는 항들을 찾아내어서 모든 항들을 하나의 집합으로 표현하여야 하는 과정을 생각해 보자면, 입력되는 집합의 크기가 작을수록 그 수행시간도 빨라짐은 당연하다 하겠다. 하지만, DFA의 프레임웍에서는 앞에서 언급한 집합의 성장 과정과 비슷한 과정을 거치면서 결과를 만들어 나간다. *IN*, *OUT*이 서서히 증가해 나가면서 *OUT*을 구하기 위해 *IN*들을 합집합, 또 다시 *IN*을 구하기 위해 *OUT* 값을 flow function에 넣는다. *IN*을 구하고 여기 새로운 항을 1개 추가하면 이 하나의 항을 *OUT*에 넣기 위해 이미 거대해져 버린 *IN*들을 다시 합집합 하여야 한다. 하나의 항만을 *OUT*에 직접 넣어 주지 못하고 *IN*에 추가한 다음 다시 합집합을 하는 것은 앞에서 예를 든 집합의 성장 과정과 비슷하다. 이는 컴퓨터상의 계산에서는 시간적 손실을 가져온다.

또 다른 예를 들어 생각해 보자.

그림 5에서처럼 루프 2개가 나열되어 있는 프로그램이 있다고 하자.

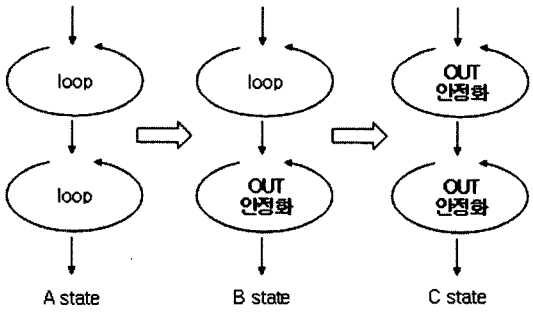

그림 5 루프가 나열되어 있는 코드에서의 이상적인 LVA 수행 순서

위쪽 루프에서는 아래쪽 루프의 *IN*값을 입력으로 받아서 *OUT*값을 계산한다. 그러므로 아래쪽 *IN*값의 계산이 완료되어 변함이 없는 상태에서 그 *IN*값을 입력으로 사용하여야 불필요한 계산을 피할 수 있다. 그림 5에서와 같이 아래쪽 루프의 계산을 모두 마친 B state로 간 다음에야 위쪽 루프에 대한 계산을 수행해야한다. 그 이전에 위쪽 루프에 대한 계산을 수행하는 것은 아래쪽 루프의 정보가 완성된 다음, 그 정보를 입력을 받아서 할 계산에 대한 부분적 반복일 뿐이다.

전통적 DFA를 따르는 LVA는 아래 루프 계산값이 완성되기 전에도 그 값들을 입력으로 하여 위쪽 루프에 대한 계산을 수행한다. 프로그램 전체에 대해 변화를 적

용해 나가기 때문이다. 아래쪽 루프의 정보가 완성되지 않고 조금만 바뀌어도 그것을 위쪽 루프에 적용하여 계산을 하게 되므로 위에서 언급한 합집합 계산에서의 오버헤드가 일어나게 된다.

필요 없이 basic block들을 방문하는 점, 완성된 정보가 아닌 것으로 서서히 증가해 나가는 방식, 마지막으로 추가해 줄 항의 수는 적지만 집합 자체는 거대해지면서 오는 합집합에서의 비효율, 이러한 점들이 기존 DFA의 프레임웍을 기반으로 한 LVA 알고리즘을 느리게 하고 있다.

4. 기존 Live Variable Analysis 알고리즘의 개선

앞에서 알아본 기존 LVA 알고리즘의 단점들을 보완하기 위해 본 논문에서 제안한 새로운 LVA 알고리즘에 대해 알아본다.

4.1 기본 아이디어

새로운 LVA 알고리즘의 핵심 아이디어는 다음과 같다. 기존의 알고리즘에서 정보가 직접 전달되는 범위를 넓히는 것이다. 바로 직접 붙어 있는 basic block들 간에서만 *IN*, *OUT*의 영향력으로 그 값을 구하는 대신 하나의 사용된 변수가 변화를 주어야 하는 모든 *OUT*들을 한 번에 찾아 올라가면서 모두 고쳐 준다.

지금까지는 *OUT*을 고치기 위해 연결된 basic block들만을 둘러보았다. *OUT*에 영향을 주는 것을 그 successor basic block들 *IN*으로 규정. 자신의 *OUT*을 업데이트하기 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. basic block의 입구 부분에서 살아 있어야 하는 변수들의 기본은 그 basic block에서 사용되고 있는 변수들의 집합이기 때문이다. 이 사용되고 있는 변수들은 자신의 값이 정의되어 있는 basic block까지의 경로들에서 모두 살아 있어야 한다. *USE*에 속한 변수에 대해서 기준을 가지고 수행을 하는 것이다. 현 basic block의 *USE*에 포함되어 있는 변수들에 대해서 프로그램의 흐름을 거꾸로 올라가면서 자신이 정의되어 있는 곳까지 모든 경로를 따라 찾아 올라가는 동안 그 경로에서 만나는 모든 basic block의 *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 6에서 간단한 예를 들고 있다. 사용되고 있는 변수 *x*가 자신의 값이 정의되어 있는 basic block까지 찾아 올라가는 과정을 나타내고 있다.

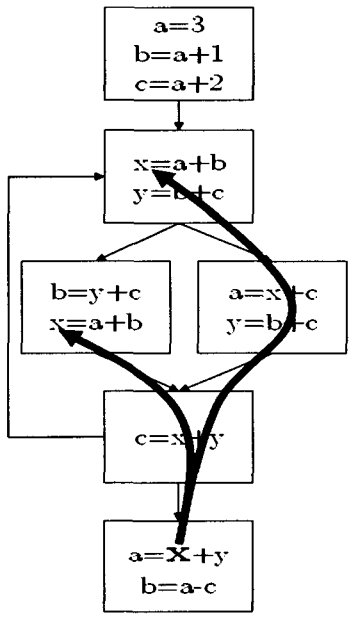

그림 6 기본 아이디어의 수행 과정

4.2 제안된 알고리즘

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

이러한 점을 보완하기 위해 부분적으로 수정하였다. 아래쪽에서 *USE*를 통해서 거슬러 올라가면서 거쳐 가는 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 반복적 DFA 프레임웍을 따르는 LVA 알고리즘이 가지는 단점 중 하나인 거대한 집합들의 반복된 합집합 연산에서 오는 부담감을 여기에서도 줄여 주

```
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
    IN(B) = USE(B)
    OUT(B) = ∪ all successor S of B USE(S)
  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 7 제안한 LVA 알고리즘

게 된다.

알고리즘의 슈도코드는 그림 7에 나타나 있다.

4.3 제안된 알고리즘의 장단점

4.3.1 장점

새로이 제안한 알고리즘의 장점은 언급한 바 있다. 이에 대한 정리를 해 보면 다음과 같다.

- OUT에 추가된 변수가 없는 basic block은 방문하지 않는다.
- 루프가 나열되어 있을 때 아래쪽 루프부터 OUT을 완성해 가면서 진행하여 완성되지 않은 정보로 위쪽 루프에 대한 연산을 진행하지 않을 수 있어 연산의 중복을 막을 수 있다.[8]
- 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.

기존의 방법에서 정보 전송의 영역을 넓힘으로써 필요 없는 basic block의 스캔 및 합집합에서의 효율을 높이는 효과를 가져 왔다.

표 1에서는 두 알고리즘이 수행되면서 basic block을 방문하는 횟수를 비교한 결과이다. 새로이 제안한 알고리즘이 방문하는 basic block의 수가 훨씬 적음을 알 수 있다. 새로운 알고리즘의 방문 횟수는 기존 알고리즘의 방문 횟수의 약 59% 정도이다.[9]

표 1 basic block 방문 횟수 비교

프로그램 종류	basic block 방문 횟수		new/old (%)
	new	old	
adpcm	3152	5980	52.7
epic	49883	79025	63.1
g721	15658	23433	66.8
ghostscript	2117698	3351949	63.2
jpeg	189054	340781	55.5
mesa	602098	995743	60.5
mpeg2	132940	286311	46.4
pegwit	33863	63948	53.0
pgp	267015	508172	52.5
rasta	61116	76234	80.2
gsm	31979	68739	46.5
합의 비율(%)	60.4	비율의 평균(%)	58.2

표 2는 LVA를 수행하면서 기존의 알고리즘과 새로이 제안한 알고리즘이 합집합을 수행하는 횟수와, 합집합 수행 시 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 basic block에서의 *USE* 집합을 보면 되므로 집합의 크기가 작아 부담감이 적은 편이다. 또한 집합 항의 개수 합에서 합의 비율은 100%를 넘어가는데 이는 rasta에서의 큰 차이가 전체에 큰 영향을 끼치어서 그렇게 되었다. rasta에 있는 코드들의 특성이 얼마

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

9) 각 프로그램들의 결과값을 다 합한 다음 비교를 하면 60.4%, 프로그램들마다의 비교 %를 가지고 평균을 내면 58.2%가 나온다. 평균을 나타내는 이 두 가지 기준에 대해서는 5장에서 다시 언급하겠다.

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	2235	3218	69.5	707904	1097600	64.5
epic	32562	35810	90.9	14931648	17329824	86.2
g721	10118	12621	80.2	3894976	4623744	84.2
ghostscript	1387365	1533002	90.5	865804320	729641344	118.7
jpeg	120909	171635	70.5	42655968	62500640	68.3
mesa	389970	489896	79.6	205974592	228940384	90.0
mpeg2	82947	131821	62.9	31496160	45852512	68.7
pegwit	21257	29813	71.3	7831776	9331840	83.9
pgp	172833	223438	77.4	71829472	77835488	92.3
rasta	36907	35362	104.4	16108224	12875520	125.1
gsm	21254	31871	66.7	7580512	10294144	73.6
연산 횟수	합의 비율 (%)		84.4	비율의 평균 (%)		78.5
집합 항			105.7			86.9

나 차이가 많이 난 것인지, 아니면 다음 절에서 말 할 단점이 유난히 부각된 것인지는 정확히 알 수 없다. 실제 실험 결과에서 집합들의 크기가 크게 나온 프로그램들도 속도는 더 빠르게 나오는 것을 알 수 있다. 자세한 결과는 5장을 참고하기 바란다.

4.3.2 단점

루프에 대한 *OUT*값을 먼저 안정화 시킨 다음 그 정보가 필요한 곳으로 *OUT*값을 전파시키는 것이 알고리즘의 장점이다. 하지만 이와 같은 동작을 때에 따라 하지 않을 수도 있다는 것이 큰 단점이다.

현재 제안한 알고리즘에서는 CFG에서 어느 edge가 루프를 형성하는 edge인지 구분하지 않고 아무 것이나 선택하게 되어 있다. 그러므로 아래쪽 루프부터 먼저 계산을 하고 올라가면 다행이지만, 먼저 작업을 할 basic block을 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 basic block의 *USE* 집합을 다시 합쳐서 올라가는 등. 부수적인 작업들이 필요하다는 점이 단점으로 작용할 수 있다. 기존의 알고리즘도 루프나 분기 등으로 코드가 크게 나뉘지 않는다면 반복이 그리 많지 않을 것이므로 이와 같은 부수적 작업에 의한 오버헤드가 더욱 크게 두드러질 수 있다.

4.4 두 알고리즘의 정확도 비교

4.4.1 직관적 관점

기존의 알고리즘에 대해 생각해 보자. *USE*에 존재하는 변수가 그 basic block의 predecessor인 basic block의 *OUT*에 추가된다. 추가된 그 변수는 *DEF*에 자신이 나타나지 않는다면 그 basic block의 *IN*에도 추가된다. 이처럼 *USE*에 나타난 변수는 *DEF*에 나타날 때까지 predecessor들을 올라가면서 *IN*, *OUT*을 거쳐 가며 추가되기를 반복하다 *DEF*에 그 변수를 포함하고 있는 basic block에서 *OUT*에는 포함되지만 *IN*에서 제외된다. *IN*에서 제외되는 순간까지 변수는 살아서 올라가며 *OUT*에 추가되는 것이다. *USE*에서 등장한 변수가 자신이 살아있어야 하는 basic block을 찾아서 거슬러 올라가는 과정을 인접한 basic block들 간의 연관성만으로 계산해 가면서(meet operator를 이용한 확장) 수행하고 있는 것이다.

이와 같은 작업을 새로운 알고리즘에서는 한 번에 *직접* 하는 차이점만 있을 뿐이다. 결국 같은 결과를 가져오는 작업인 것이다. *USE*, 즉 사용하는 변수에 대해서 그 변수가 정의된 곳까지 거슬러 올라가면서 *OUT*에 그 변수를 추가하는 것. 한 변수가 정의된 곳에서부터 그 변수가 사용된 곳까지의 경로에 존재하는 모든 basic block들의 *OUT*에 그 변수를 추가해 나가는 것을 두 알고리즘이 다른 방법으로 하고 있을 뿐이다.

근본적으로는 같은 일을 하는데, 그 진행 범위를 부분에서 전체로 확장하여 비효율적인 점들을 없애준 것이 새로운 알고리즘이다. 인접한 basic block간에서만 정보를 주고받으면서 bubble sort에서처럼 그 영향이 최종 목적지까지 도착하기를 기다리는 기존의 알고리즘은 근접한 것들끼리만 비교하는 것을 계속 반복하여 고정점에 도달할 때까지 진행하는 것이다. 목적지를 찾아서 직

접 최종 목적지까지 올라가는 insertion sort와 같은 새로운 알고리즘. *USE*에 포함된 변수들이 자신이 정의된 곳까지 거슬러 올라가고자 하는 것은 같다. 업데이트 효과를 직접 보게 하기 위한 움직임에서만 차이가 있을 뿐 궁극적으로는 같은 문제를 풀어서 같은 답을 내는 알고리즘이다.

4.4.2 두 알고리즘의 결과물이 같음에 대한 증명

두 알고리즘의 결과가 같음을 증명하고자 한다.

두 알고리즘의 결과라 할 수 있는 것은 두 알고리즘이 만들어 내는 프로그램의 각 basic block들의 *OUT* 집합들이 같음을 보이면 두 알고리즘의 결과가 같다는 것을 보이게 된다. *IN* 집합은 *OUT* 집합을 다 만들어 낸 다음 *OUT*으로부터 만들면 되므로 고려 대상에서 제외하겠다.

이를 위해 증명해야 할 두 가지 사실은 "기존의 알고리즘이 만들어 내는 각 basic block B에서의 $OUT(B)$가 새로이 제안한 알고리즘이 basic block B에서 만들어 내는 $OUT(B)$에 포함된다는 사실"과 그 역, 즉 "새로이 제안한 알고리즘이 만든 B에서의 $OUT(B)$가 기존의 알고리즘이 basic block B에서 만드는 $OUT(B)$에 포함된다"는 것이다.

편의를 위해 앞으로 기존의 알고리즘이 각 basic block B에서 만드는 $OUT(B)$를 $OUT_{old}(B)$라고 하고, 새로이 제안한 알고리즘이 만드는 것을 $OUT_{new}(B)$라고 하겠다.

먼저 전자에 대해 살펴보자. 일반성을 잃지 않기 위해, 임의의 basic block B에서 $OUT_{old}(B)$에 속하는 임의의 변수 v에 대해 생각해 보자. v가 $OUT_{old}(B)$에 있다는 것은 B의 CFG상에서 successor basic block들 중 적어도 하나의 *IN*에 v가 포함된다는 것을 뜻한다. 이 basic block을 L이라고 하면, $IN(L)$에 포함된다는 것은 기존 알고리즘의 방법에 따르면 USE(L)에 속하거나 $OUT_{old}(L) - DEF(L)$에 속해야 한다. 이와 같이 반복하여 내려가다 보면 기존 알고리즘의 출발점으로 가게 되는데, *IN*을 만드는 초기값이 *USE*이므로 *USE*에 v가 포함되는 basic block이 존재하게 된다(사실 이는 변수가 살아있다는 정의 자체가 가지는 의미이다). CFG상에서 basic block들의 successor들만을 따라 내려가다가 *USE*에 v가 있는 basic block을 발견했다는 것이다. 또한 거기까지 내려가는 경로 상에서 *DEF*에는 변수 v가 속한 적이 없다. 이와 같은 상황을 새로이 제안한 알고리즘에 적용해 본다면 *USE*에 있는 변수 v가 거슬러 올라가면서 만나는 basic block들의 *DEF*에 v가 없었다는 것이므로 basic block B에까지 도달하여 $OUT_{new}(B)$에 변수 v를 추가하게 된다. $OUT_{old}(B)$에 속하는 모든 변수가 $OUT_{new}(B)$에 속한다는 뜻이 되므로 $OUT_{old}(B)$는 $OUT_{new}(B)$에 포함된다.

후자도 마찬가지 방법이다. 일반성을 잃지 않기 위해, 임의의 basic block B에서 $OUT_{new}(B)$에 속하는 임의의 변수 v에 대해 생각해 보자. 새로이 제안한 알고리즘의 구조상 프로그램 진행 경로 중에 있는 어느 한 basic block의 *USE*에 변수 v가 포함되는 basic block S가 존재하여야 하고, CFG상에서 B에서 S까지 가는 경로 상의 basic block들의 *DEF*에는 변수 v가 포함되지 않는다는 것을 뜻한다. B에서 S까지 가는 경로가 $B - B_1 - B_2 - \ldots - B_n - S$ 라고 하자. $USE(S)$에 변수 v가 존재하므로 $IN(S)$에 변수 v가 포함된다. 기존 알고리즘대로 이 경로를 따라 올라가면서 알고리즘을 적용시켜 보면 $OUT_{old}(B_n)$에 $IN(S)$가 포함되므로 변수 v도 존재한다. $DEF(B_n)$에 변수 v가 존재하지 않으므로 $IN(B_n)$에도 변수 v가 포함되어 있다. 이와 같이 계속 반복하면 $OUT_{old}(B)$에 변수 v가 포함된다는 것을 알 수 있다. 이를 통해 $OUT_{new}(B)$의 모든 변수가 $OUT_{old}(B)$에 존재함을 알 수 있으므로 $OUT_{new}(B)$는 $OUT_{old}(B)$에 포함됨을 알 수 있다.

$OUT_{old}(B)$와 $OUT_{new}(B)$가 서로를 포함하는 관계에 있으므로 이 두 집합은 같게 된다. 임의의 basic block에 대해 성립하므로 기존의 알고리즘이 입력된 프로그램에서의 전체 basic block들마다 각각 만들어 내는 *OUT* 집합과 새로이 제안한 알고리즘이 만드는 것이 같음을 증명한 것이다. 즉, 두 알고리즘이 만들어 내는 *OUT* 집합의 값이 모두 같고, 결론적으로 두 알고리즘은 같은 결과를 만들어 낸다는 것이 증명되었다.

5. 구현 및 실험 결과

5.1 실험 환경

실험을 한 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.

벤치마크 프로그램으로 MediaBench[14]를 사용하였다.

5.2 실험 결과

두 알고리즘의 성능 비교 실험에 대한 결과이다. 결과는 아래의 2가지로 요약된다.

- 전체 컴파일 과정에서 LVA가 차지하는 비율
- 두 알고리즘의 수행 속도 비교

성능 비교에서의 비율을 나타내는 방법으로 2가지를 사용하였다. 하나는 모든 값들을 합해서 비율, 나머지

표 3 벤치마크 프로그램들의 크기(줄 수)

프로그램 종류	줄 수
adpcm	9614
epic	112546
g721	43893
ghostscript	41425770
jpeg	902094
mesa	1116293
mpeg2	376504
pegwit	86644
pgp	778961
rasta	167886
gsm	190613

표 4 전체 컴파일 수행 과정 중 LVA가 수행되는 횟수

종류	LVA	VPO while	LVA/VPO
adpcm	67	14	4.8
epic	539	116	4.7
g721	277	59	4.7
ghostscript	35174	7555	4.7
jpeg	4417	960	4.6
mesa	5772	1252	4.6
mpeg2	1954	423	4.6
pegwit	807	173	4.7
pgp	3066	662	4.6
rasta	655	141	4.7
gsm	688	147	4.7
합의 비율(%)	4.6	비율의 평균(%)	4.7

하나는 비교 대상들 간의 비를 먼저 구한 다음 그 비에 대한 평균이다.

이렇게 두 가지를 구한 이유는 다음과 같다. 현재 실험하는 대상이 되는 파일들은 하나의 패키지[10]로 묶인 여러 가지 프로그램들이다. 하나의 프로그램으로 묶여 있는 여러 개의 C 파일들을 하나의 패키지란 이유로 그들 간에 어떤 유사성과 공통적인 특성이 있을 수 있음을 인정하는 것이다. 물론 하나의 패키지를 구성하는 파일들은 항상 같이 컴파일 되어야 한다는 점이 그들 각각의 컴파일 수행시간의 합에 충분한 의미를 부여한다. 하지만 이렇게 생각할 수도 있다. 평균적으로 하나의 파일, 아니 하나의 함수에 대해 컴파일 할 때 두 알고리즘들 간의 성능 비교에서는 하나의 패키지로 묶이지 않은 파일들도 동등한 위치에서 바라볼 수 있다는 것이다. 그러므로 패키지 단위로 얼마나 좋아졌는가의 비율의 평균도 의미를 가지지만, 각 파일들을 컴파일한 시간의 총합, 즉 모든 패키지를 컴파일 하는 데 든 시간의 총합도 의미를 가진다. 그러므로 여기에서는 패키지에서의 비율에 대한 평균과 모든 패키지를 수행하면서 나온 값[11]들의 총합에 의한 비율도 제시하여 두었다.

5.2.1 LVA가 전체 컴파일 과정에서 차지하는 비율

먼저 LVA가 컴파일 전 과정이 수행되는 시간의 측면에서 보았을 때 어느 정도를 차지하는가를 알아보고자 한다. 다른 주요 최적화 기법들에 비해 한 번 수행되는 시간이 짧다고 할 수 있는 LVA가 전체적으로 큰 비중으로 가질 수 있기 위해서는 일단 먼저 그 수행 횟수가 많아야 한다.

이를 알아보기 위해 Zephyr의 최적화 과정이 수행 되면서 입력된 프로그램 코드가 변화가 없을 때까지 최적화 과정을 반복적으로 적용한다는 특징에서 다른 최적화 기법들이 수행되는 횟수와 LVA가 수행되는 횟수를 비교해 보았다. 그에 대한 실험 결과는 표 4에 나타나 있다. 여기서 다른 최적화 기법 수행 횟수의 기준은 VPO[12]의 구조에서 최적화 기법들을 적용하여 결과 코드가 변화가 없을 때까지 계속 루프(loop)를 돌면서 반복하는데, 이 루프의 횟수를 세어서 비교 대상으로 하였다. 한 번 루프가 돌면서 기본적으로 다른 최적화 기법들은 한 번씩 수행되기 때문에 적당한 비교 대상이 된다.

결과적으로 다른 최적화 기법들이 1번 수행 될 때 LVA는 4~5번 정도 실행됨을 알 수 있다. 정확히는 각 패키지별의 비율에 대한 평균은 다른 최적화 기법 1번당 4.7회, 총합에서의 비율은 4.6회이다. 이 결과를 보았을 때 다른 최적화 기법들과 LVA의 수행 시간을 비교할 때는 LVA이 수행 시간에 4.65정도를 곱해 준 후 비교를 하는 것이 적절하다고 할 수 있다.

그러면 실제 전체 컴파일 수행 시간에서 LVA가 차지하는 시간의 비율을 알아보자.

표 5에서 사용한 LVA 알고리즘은 기존 DFA 프레임웍을 따른 알고리즘이다. 기존의 컴파일러 그 자체에서 LVA가 차지하는 그 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(합의 비율은 7.0%, 비율의 평균은 7.5%)을 알 수 있다.

5.2.2 두 알고리즘의 수행 속도 비교

본 논문에서 제시한 새로운 LVA 알고리즘과 기존의 알고리즘간의 수행 시간을 비교하였다. Zephyr 컴파일러에 기존에 구현되어 있는 LVA 알고리즘이 *DEF*,

10) 여기에서 패키지(package)라 함은 하나의 프로그램을 구성하는 여러 개의 파일들을 하나로 묶어서 다룬다는 의미로 사용하였다.

11) 여기에서 나온 값이라 함은 컴파일 수행 시간이 될 수도 있고, 방문한 basic block의 개수가 될 수도 있다. 앞에서는 예를 들어 설명하다 보니 컴파일 수행 시간만을 언급했는데, 그 외의 값들도 존재하므로 혼동하지 않기를 바란다.

12) VPO란 Zephyr 컴파일러에서 코드 최적화를 담당한 부분의 이름이다. VPO는 최적화 기법들을 일련의 순서대로 적용하면서 코드의 변화가 없을 때까지 계속 반복하여 적용하는 구조로 되어 있다.

표 5 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	0.0845	1.126	7.5
ghostscript	11.440	177.355	6.5
jpeg	1.310	18.805	7.0
mesa	3.243	49.16	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
합의 비율(%)	7.0	비율의 평균(%)	7.5

*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, 새로이 제안한 알고리즘을 new라고 하면 $\frac{old-new}{old}\times 100$으로 구하였다.

실험 결과는 표 6과 같다. 전체 컴파일 수행 시간에서 7% 정도를 차지하던 기존의 알고리즘과는 달리 새로운 LVA 알고리즘은 전체 컴파일 수행 시간에서 합의 비율 4.6%, 비율의 평균 4.2%를 차지하고 있다. 이는 전체 컴파일 수행 시간에서의 개선치가 약 2%~4%가 됨(합의 비율 2.6%, 비율의 평균 3.5%)을 의미한다. LVA 자체 성능 개선을 본다면 40%에 육박하는(합의 비율 36.4%, 비율의 평균 46.8%) 수준이다. 기존의 알고리즘에서 수행 시간을 약 절반으로 줄여준다.

6. 결론 및 향후 연구 과제

6.1 결론

기존의 DFA 프레임웍을 충실히 따라서 LVA를 수행하는 알고리즘은 여러 가지 단점이 있다. 반복적 알고리즘이라는 점에서 오는 상황에 따른 오랜 수행 시간과 직접 연결된 basic block들만 서로 영향을 주는 방식에 의해 결과값이 서서히 증가함에 따른 비효율적인 움직임 등이 바로 그것이다.

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

이러한 개선 방법은 구현이 복잡하지 않다. DFS 순서로 수행하는 반복 알고리즘에서 조금의 수정만으로 높은 성과를 올릴 수 있다는 점이 본 알고리즘의 가장 큰 장점이다.

LVA 자체에 대해서는 얼마나 빨라졌나를 알아보면 시간의 합으로 비교하였을 때 36.4%가 빨라졌고, 빨라진 비율의 평균은 46.8%였다. Zephyr의 기존 구현 때문에 LVA 알고리즘 자체의 시간 측정에서 *DEF*, *USE*를 구하는 부분도 포함하여 측정하였다. 이런 점에서 실제로 *OUT*만을 구하는 부분에서의 속도 향상은 더욱

표 6 두 가지 LVA 알고리즘의 수행 시간 비교

프로그램 종류	집합 연산 횟수		LVA 개선(%)	참여 집합 항 개수 합		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
연산 횟수	합의 비율 (%)		36.4	비율의 평균 (%)		46.8
집합 항			2.6			3.4

클 것으로 추정된다.

전체 컴파일 수행시간에서 LVA가 차지하는 비율은 시간의 합으로 조사한 경우 전체 컴파일 수행 시간에서 LVA가 자치하는 비율은 7.0%였고, 전체 컴파일 수행 시간과 LVA의 비율의 평균은 7.5%였다. 그다지 큰 비중을 차지하지 않을 줄로만 알았던 LVA가 전체 컴파일 수행 시간에서 꽤 큰 비중을 차지하고 있음을 알 수 있다. 또한 이 비율이 새로운 알고리즘을 사용하였을 경우 합의 비율은 4.6%, 비율의 평균은 4.2%로 떨어지는 것도 확인할 수 있다.

새로이 제안한 LVA 알고리즘을 사용하면 컴파일 수행 시간이 기존의 전체 컴파일 수행 시간보다 합의 비율로는 2.6%, 비율의 평균으로는 3.5% 빨라졌다. 여러 개의 파일들을 한 번에 컴파일 하여야 하는 경우가 많다는 것을 고려할 때 패키지별로 고려치 않고 그들을 모두 독립적으로 생각하여 시간의 합으로 비교하는 것도 의미가 있다. 그리고 각각의 경우에서 몇 퍼센트씩 빨라지는가를 평균하여 하나의 파일에 대해 이 파일의 기존 컴파일 수행 시간으로부터 새로운 알고리즘의 적용 시 얼마나 빨라질 것인가에 대한 추측을 할 수 있음에서 비율의 평균도 의미를 가진다. 그러므로 합의 비율과 비율의 평균 둘 모두를 나타내었다. 본 논문에서 프로그램 유형에 따른 어떠한 조사도 정확히 하지 않은 관계로 프로그램들 간의 비율들보다는 각 파일들에 걸린 시간들을 모두 합쳐서 계산하는 합의 비율에 더욱 의미를 부여하는 것이 좋다. 프로그램 하나하나를 단위로 보지 않고 파일 하나하나, 함수 하나하나를 단위로 보는 것이 특별한 분류가 있지 않은 현재로써는 더욱 의미가 있다.

6.2 향후 연구 과제

향후의 연구 과제로는 우선적으로 다양한 컴파일러에서 실험해 보는 것이다. Zephyr 컴파일러가 사실상 그리 널리 사용되고 있는 컴파일러는 아니므로 좀 더 실용적으로 많이 사용되고 있는 컴파일러에서의 실험도 큰 의미를 가지게 된다. 특히 JIT 컴파일 시스템에서 구현하여 성능 향상이 얼마나 있는가를 알아보는 것도 큰 의미가 있을 것이다. 또한 다른 컴파일러들에서 LVA를 구현해 놓은 부분에서 나름의 구현 기법들에서 DFA 프레임웍을 구현하는 데에서 오는 차이가 있을 수 있으므로 이러한 실험은 더욱 중요하다.

다음으로는 이 논문에서 사용한 아이디어가 왜 기존 알고리즘보다 속도가 빨라지는지에 대한 좀 더 이론적인 근거를 찾아보는 것이 필요하다. 프로그램들의 다양한 패턴들 중에서 어떠한 패턴에서 기존의 알고리즘이 더 빠른 경우도 있을지. 어떤 패턴일 때 새로운 알고리즘이 최대 효율을 발휘하게 되는지. 더 빨라지는 이유에는 본 논문에서 생각해 본 점들 이외에는 또 어떤 점들이 있을지. 다양한 상황에 대한 분석을 하여 좀 더 근본적으로 둘 간의 성능 차이가 나는 이유에 대한 분석이 필요하겠다. 간단히 살펴보아도 basic block 방문 횟수가 줄어드는 비율과 시간 단축의 비율이 비례하지 않는 것을 보았을 때 다른 여러 가지 요소들이 있다고 생각해 볼 수 있다. 단순한 수행 시간 측정만으로는 근본적인 원인과 장단점을 알 수 없으므로 이에 대한 좀 더 많은 패턴들을 기반으로 한 분석이 절대적으로 필요하다. 이와 같은 조사를 통해 본 논문에서 제시하고 있는 알고리즘을 더욱 다듬을 수 있고, 더 멀리는 새로운 DFA 프레임웍으로 확장시켜볼 수 있을 것이다.

참 고 문 헌

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

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

[3] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. In *ACM Transactions on Programming Languages and Systems*, 21(5) pages 895~913, September 1999.

[4] Zoran Budimlic, Keith D. Cooper, and Timothy J. Harvey. Fast Copy Coalescing and Live-Range Identification. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25~32, 2002.

[5] Keith D. Cooper, Timothy J. Harvey, and Linda Torczon. How to Build an Interference Graph. In *Software-Practice and Experience*, VOL. 1(1), 1, January 1988.

[6] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. In *Acta Informatica*, 7(3) pages 305~317, July 1977.

[7] Susan L. Graham, Mark N. Wegman. A Fast and Usually Linear Algorithm for Global Flow Analysis. *JACM* 23(1): 172~202, 1976.

[8] C. Samuel Hsieh. A fine-grained data-flow analysis framework. In *Acta Informatica*, 34(9) pages 653~665, August 1997.

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

[10] 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.

[11] 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.
[12] 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*, pages 72~84, 1998.
[13] S. W. K. Tjiang and J. L. Hennessy. Sharlit: A tool for builing optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 82~93, 1992.
[14] *MediaBench* Homepagehttp://www.cs.ucla.edu/~leec/mediabench/

윤 정 한
1997년~2001년 한국과학기술원 전산학과(학사). 2001년~2003년 한국과학기술원 전자전산학과 전산학전공(석사). 2003년~현재 한국과학기술원 전자전산학과 전산학전공 박사 과정. 관심분야는 프로그래밍 언어, 컴파일러, 임베디드 시스템, 프로그램 분석

한 태 숙
정보과학회논문지 : 소프트웨어 및 응용
제 32 권 제 3 호 참조