제37회 한국정보처리학회 춘계학술대회 논문집 제19권 1호 (2012. 4)

워크리스트 알고리즘에 기반한 자료 의존성 그래프 생성기 구현

이종원*, 윤종희**, 김용주*, 백윤흥*
*서울대학교 공과대학 전기컴퓨터공학부
**강릉원주대학교 과학기술대학 컴퓨터공학과
e-mail : jwlee@sor.snu.ac.kr, jhyoun@cs.gwnu.ac.kr, yjkim@sor.snu.ac.kr, ypaek@snu.ac.kr

Implementation of a Data Dependence Graph Generator Based on Worklist Algorithm

Jongwon Lee*, Jonghee M. Youn**, Yongjoo Kim*, Yunheung Paek*
* Dept. of Electrical and Computer Engineering, Seoul National University
** Dept. of Computer Science and Engineering, Gangneung-Wonju National University

요 약

컴파일러의 명령어 스케줄링 기법이 제대로 동작하기 위해서는 자료 의존성 그래프가 필요하다. 본 논문에서는 워크리스트 알고리즘을 사용한 자료 의존성 그래프 생성 구현에 대하여 설명하고자 한다.

1. 서론

컴파일러는 C 나 Java 와 같은 상위 수준 프로그래밍 언어로 기술된 소스 코드를 입력으로 받아들여 어셈블리 코드를 생성하는 개발 도구이다. 컴파일러의 코드 생성 과정은 크게 명령어 선택(instruction selection), 레지스터 할당(register allocation), 명령어 스케줄링(instruction scheduling)의 세 가지로 나뉘어진다. 이 중 명령어 스케줄링은 명령어 간의 자료 의존성을 토대로 코드 내의 명령어의 위치를 재배치하여 실행 순서를 바꾸거나 VLIW 와 같은 병렬 아키텍처에서는 동시 실행 가능한 명령어의 조합을 찾아내어 프로그램의 성능 향상을 꾀하는 기법이다. 이러한 명령어 스케줄링이 가능하도록 하기 위해서는 명령어 간의 자료 의존성을 나타내는 자료 의존성 그래프(Data Dependence Graph)가 명령어 스케줄러의 입력으로 제공되어야 한다.

본 논문에서는 자료 흐름 분석(Data Flow Analysis)에 주로 사용되는 워크리스트 알고리즘(Worklist Algorithm)을 기반으로 한 자료 의존성 그래프 생성기 구현에 대하여 설명하고자 한다. 2 장에서 워크리스트 알고리즘에 대해 알아보고, 자료 의존성의 종류에 대해서 살펴본 뒤, 3 장에서 자료 의존성 그래프 생성기의 구현 내용이 이어질 것이다. 4 장에서는 자료 의존성 그래프 생성기가 구현된 환경과 실험 결과에 대해서 논하고, 5 장에서 끝맺도록 하겠다.

2. 배경

2.1 워크리스트 알고리즘

컴파일러는 코드 생성을 위하여 다양한 종류의 자료 흐름 분석을 수행한다. 이러한 자료 흐름 분석에는 도달 정의 분석(reaching definition analysis), 라이브 변수 분석(live variable analysis) 등이 대표적인데 이들 분석은 모두 워크리스트 알고리즘을 사용하여 이루어진다[1]. 워크리스트 알고리즘은 다음과 같은 상황 아래에서 동작한다. 우선 노드(node)의 집합 N 과 엣지(edge)의 집합 E 로 구성된 임의의 그래프 G=<N,E>를 가정하자. 그래프 G 는 진입 노드(entry node)와 출구 노드(exit node)를 소유하고, N 에 속하는 임의의 노드 B 에 대하여 in(B), out(B) 를 정의한다. in(B)와 out(B)가 의미하는 값을 격자(lattice)라고 하는데, 이는 워크리스트 알고리즘을 통해 얻고자 하는 자료의 속성으로 볼 수 있다. 예를 들어, 도달 정의 분석의 경우에 격자는 어떤 변수 값에 값을 쓰는 명령어, 즉, 그 변수를 정의하는 명령어가 되고, 라이브 변수 분석의 경우에는 프로그램 내에서 사용되는 변수가 격자에 해당된다. in(B)는 B 의 입력, out(B)는 B 의 출력을 의미하며, 둘은 Figure 1 에 나타난 수식 관계를 만족시켜야 한다.

$$\text{in(B)} = \begin{cases} Init & for\ \ B = entry \\ \bigcap_{P \in Pred(B)} out(P) & otherwise \end{cases}$$

$$\text{out(B)} = F_B(in(B))$$

Figure 1. 자료 흐름 수식(Data Flow Equations)

Figure 1 에서 Init 은 초기값을 의미하는데, 진입 노드의 입력이 이에 해당한다. 진입 노드를 제외한 나머지 노드의 입력은 전임자 노드(predecessor node)들의

출력값들로부터 결정된다. 이 때, 출력값들로부터 결정하는 방식은 분석하고자 하는 자료의 특성에 따라 달라질 수 있다. 그 방식에는 합집합 연산, 교집합 연산 등이 사용될 수 있고, 그 외에도 자료의 속성에 따라 다른 종류의 연산이 사용될 수 있다. 그리고 $F_B()$는 노드 B 내부에서의 자료 흐름 처리를 의미하고, 이 역시 분석 대상 자료에 따라 달라진다. 워크리스트 알고리즘은 진입 노드를 제외한 나머지 노드들로 이루어진 초기 워크리스트에서 출발하여, 이 워크리스트로부터 노드를 하나씩 선택한 뒤 Figure 1 에 나타낸 연산을 거친다. 이러한 과정을 워크리스트에 더 이상 노드가 남아있지 않을 때까지 반복하면 수행이 끝나게 된다.

2.2 자료 의존성의 종류

명령어 사이의 자료 의존성은 흐름 의존성(flow dependence), 반의존성(anti-dependence), 출력 의존성(output dependence) 세 가지로 분류된다[1]. 명령어가 자료를 전달하고 보관하는 저장 공간은 레지스터와 메모리의 두 종류이므로, 명령어 간의 자료 의존성은 레지스터를 통한 세 가지와 메모리를 통한 세 가지를 합쳐 총 여섯 가지가 존재함을 알 수 있다.

3. 자료 의존성 그래프 생성 알고리즘

자료 의존성 그래프의 생성은 소스 코드를 구성하는 함수(function) 단위로 이루어진다. 컴파일러는 함수 단위로 기본 블록(basic block)들의 그래프인 제어 흐름 그래프(control flow graph)를 생성하는데, 이 제어 흐름 그래프를 입력으로 하여 자료 의존성 그래프를 생성한다. 이 장에서는 자료 의존성 그래프 생성 문제가 어떻게 워크리스트 알고리즘 문제로 치환될 수 있는지에 대해 설명한다. 또한 자료 의존성 그래프가 생성되는 두 가지 단계에 대해 각각 설명한다. 자료 의존성 그래프의 생성의 첫 단계는 기본 블록 단위로 자료 의존성 그래프를 생성하는 것이고, 모든 기본 블록들에 대한 작업이 끝나면 두번째 단계로 워크리스트 알고리즘을 적용하여 함수 단위의 자료 의존성 그래프 생성이 완료된다.

3.1 자료 의존성 그래프 생성 문제를 워크리스트 알고리즘 문제로 치환

자료 의존성 그래프 생성 문제가 어떻게 워크리스트 알고리즘으로 치환될 수 있는지 알아보자. 워크리스트 알고리즘을 적용하려면 우선 그래프를 정의해야 하는데, 함수의 제어 흐름 그래프가 이에 대응된다. 제어 흐름 그래프에서는 기본 블록이 노드에 해당하고, 기본 블록들의 연결 관계가 엣지로 표현된다. 그래프를 정의하고 나면, 다음으로 격자를 정의해야 한다. 격자는 레지스터에 대한 자료 의존성과 메모리에 대한 자료 의존성에 따라 다르게 정의된다. 우선 레지스터에 대한 자료 의존성에 대해 살펴보자. 서로 다른 두 명령어 사이에 같은 레지스터를 사용하는 경우에 자료 의존성이 존재하게 된다. 이 때, 레지스터는 원천 오퍼랜드(source operand)나 목적지 오퍼랜드(destination operand) 두 가지 종류로 사용된다. 따라서 임의의 두 명령어 사이의 특정 레지스터에 대한 자료 의존성 여부를 찾기 위해서는 각각의 레지스터에 대해 그 레지스터를 원천 오퍼랜드로 사용하는 명령어들과 목적지 오퍼랜드로 사용하는 명령어들에 대한 정보를 알고 있어야 한다. 이러한 정보들을 각각의 기본 블록들에 대해 수집하고, 이 정보를 바탕으로 서로 다른 블록 사이의 명령어들의 자료 의존성을 찾을 수 있다.

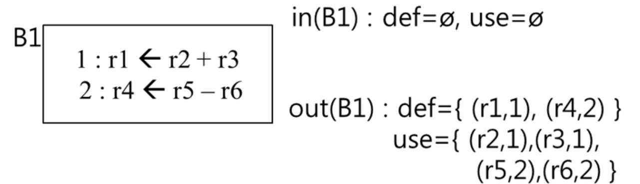

Figure 2. 레지스터 자료 의존성의 격자 정의 예제

Figure 2 는 명령어 두 개로 구성된 기본 블록 B1 에서 위의 격자 정의를 바탕으로 in(B1)과 out(B1)값을 구한 결과를 보여주고 있다. def 는 레지스터와 그 레지스터를 목적지 오퍼랜드로 사용하는 명령어들의 집합을 나타내고, use 는 레지스터와 그 레지스터를 원천 오퍼랜드로 사용하는 명령어들의 집합을 나타낸다. 모든 기본 블록들에 대해서 이와 같이 in, out 값을 계산하고, 이를 바탕으로 서로 다른 블록 사이의 명령어들의 레지스터 자료 의존성을 찾는다.

다음으로 메모리에 대한 자료 의존성의 경우에 격자를 정의하는 방식을 살펴보자. 메모리를 접근하는 명령어는 메모리로부터 값을 읽는 명령어와 메모리에 값을 쓰는 명령어 두 종류로 나뉜다. 따라서 우선 전체 명령어 중에서 메모리 접근 명령어들을 선택한 뒤, 메모리 접근 명령어들 중에서 서로 같은 메모리 주소를 참조하는 명령어들을 찾아내어 자료 의존성을 나타내게 된다.

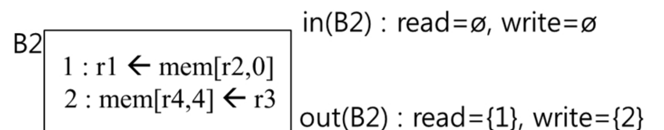

Figure 3. 메모리 자료 의존성의 격자 정의 예제

Figure 3 는 메모리 접근 명령어 두 개로 이루어진 기본 블록 B2 에서의 in, out 값을 보여준다. read 는 메모리로부터 값을 읽는 명령어들의 집합이고, write 은 메모리에 값을 쓰는 명령어들의 집합이다. 모든 기본 블록들에 대해서 이와 같이 in, out 값을 계산하고, 이를 바탕으로 서로 다른 블록 사이의 명령어들 사이의 메모리 자료 의존성을 찾는다.

이렇게 자료 저장 장소에 따른 격자를 정의한 뒤, 각각의 노드의 in 값을 계산하는 연산을 정의해야 한다. Figure 1 에 나타난 것처럼 진입 노드가 아닌 노드의 경우 전임자 노드들의 출력값으로부터 자신의 in 값을 계산해야 하는데, 자료 의존성 그래프 생성 문제의 경우 이는 합집합 연산이 되어야 한다. 자신 노

드와 전임자 노드 사이에 존재할 수 있는 모든 자료 의존성 관계를 찾아야 하기 때문에 전임자 노드들의 모든 출력값들을 합하여 자신 노드 내의 명령어들과의 자료 의존성 여부를 찾아야 한다. 마지막으로 노드 내부에서의 자료 흐름 처리를 통해 in 과 out 의 관계를 결정하는 $F_B()$ 를 정의해야 하는데, 이 역시 합집합으로 정의된다. 입력으로 들어온 in 값에 자신 노드 내부의 정보를 합하여 out 으로 내보내면 되기 때문이다.

이로써 자료 의존성 그래프 생성 문제를 워크리스트 알고리즘을 적용할 수 있도록 치환이 마무리 되었다. 이제 워크리스트를 설정하고 워크리스트 알고리즘을 수행하면 해당 함수에 대한 자료 의존성 그래프 생성이 완료된다.

3.2 기본 블록 단위의 자료 의존성 그래프 생성

자료 의존성은 기본 블록 내의 명령어들 사이에 존재하는 경우와 서로 다른 블록 내의 명령어들 사이에 존재하는 경우 두 가지로 나뉜다. 우선 기본 블록 내의 명령어들 사이의 자료 의존성을 찾고, 그 뒤에 서로 다른 블록 내의 명령어들 사이의 자료 의존성을 찾는다. 기본 블록 내의 명령어들 사이의 자료 의존성을 찾는 과정은 다음과 같다. 모든 기본 블록들을 독립된 노드로 간주하고, 각각의 노드에 대해서 노드 내의 명령어들을 배치된 순서로 조사하여 자료 의존성을 찾는다. 이 과정에서 in, out 값도 결정되는데, 모든 기본 블록을 독립적으로 간주했기 때문에 in 값은 NULL 이 되고, out 값은 해당 노드 내의 격자 정보만이 담기게 된다. 여기까지 완료되면 기본 블록 내의 자료 의존성 그래프가 생성되고, 각 블록의 in, out 값이 계산된 상태이다. 이 상태에서 워크리스트 알고리즘을 적용하면 서로 다른 기본 블록 사이의 자료 의존성까지 추가적으로 나타낼 수 있게 된다.

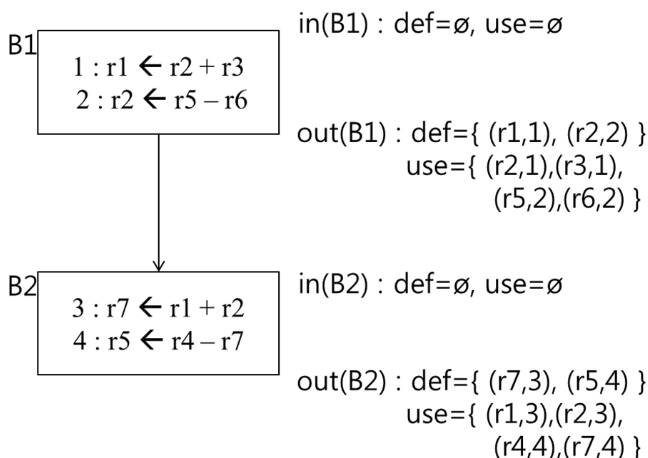

Figure 4. 기본 블록 단위의 자료 의존성 그래프 생성 예제

Figure 4 는 두 개의 기본 블록으로 이루어진 함수 내의 각각의 기본 블록에 대한 자료 의존성 그래프 생성 과정 후의 in, out 값을 나타내고, Figure 5 는 각각의 기본 블록의 자료 의존성 그래프를 나타낸다. 기본 블록 B1 과 B2 내에 각각 한 가지씩 자료 의존성이 존재함을 확인할 수 있다.

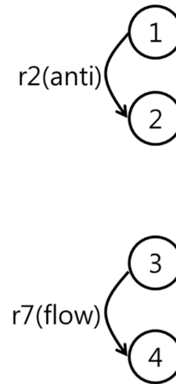

Figure 5. 기본 블록 내의 자료 의존성 그래프 생성 모습

3.3 함수 단위의 자료 의존성 그래프 생성

3.2 에서 설명한 것과 같이 각각의 기본 블록에 대해 자료 의존성 그래프를 생성하고, in, out 값을 계산한 뒤에 이 정보들을 바탕으로 함수 단위의 자료 의존성 그래프를 생성한다. 이 과정에서 워크리스트 알고리즘이 적용된다. Figure 6 은 워크리스트 알고리즘이 적용된 후의 각각의 기본 블록의 in, out 값의 변화를 보여준다.

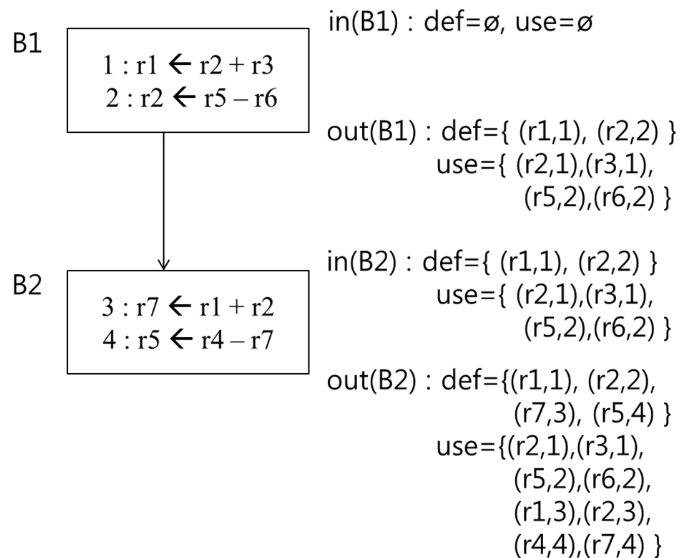

Figure 6. 함수 단위의 자료 의존성 그래프 생성 예제

기본 블록 B2 에 대한 in, out 값의 변화를 살펴볼 수 있다. in(B2)값이 B2 의 전임자 노드인 B1 의 out 값을 반영하고 있고, out(B2) 역시 in(B2) 값에 자신의 격자 정보를 합한 결과를 보여주고 있다. 이와 같이 단순히 2 개의 기본 블록으로 구성된 함수의 경우에는 한 번의 워크리스트 알고리즘의 반복으로 종료되지만 복잡한 제어 흐름 그래프를 갖는 함수의 경우는 보다 많은 반복이 이루어지게 된다.

Figure 7 에 최종적으로 완성된 함수 단위의 자료

의존성 그래프가 나타나있다. Figure 5 에 나타난 기본 블록 내의 자료 의존성 이외에 기본 블록 B1 과 B2 사이의 명령어의 자료 의존성이 추가적으로 나타나있음을 확인 할 수 있다.

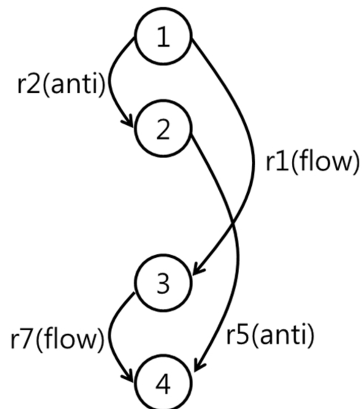

Figure 7. 함수 단위의 자료 의존성 그래프 생성 모습

4. 실험 환경 및 결과

자료 의존성 그래프 생성기는 SoarGen 재겨냥성 컴파일러 플랫폼 위에 구현되었다[2]. 대상 프로세서(target processor)로는 Synopsys[3] 사의 Processor Designer 에서 제공하는 RISC 아키텍처와 VLIW 아키텍처 두 가지 종류를 선정하였고, 각각에 대하여 Processor Designer 의 이진 코드 생성 도구(binary utility)와 시뮬레이션 환경을 사용하여 컴파일러가 생성한 코드의 정확성을 검증하였다. 실험은 명령어 스케줄링을 적용하지 않는 경우와 자료 의존성 그래프를 생성하여 명령어 스케줄링을 적용하는 경우에 각각에 대해서 결과를 얻어 서로 비교하는 방식으로 진행되었다. DSPstone[4] 과 Livermore Loops[5] 등의 벤치마크를 통해 결과에 오류가 없음을 확인하여 구현된 자료 의존성 그래프의 완성도를 점검할 수 있었다.

5. 결론

컴파일러의 성능 향상을 위해서는 효율적인 명령어 스케줄링이 필요한데, 이의 전제 조건으로 올바른 자료 의존성 그래프 생성이 필수적이다. 이에 본 논문에서는 자료 흐름 분석에 사용되는 워크리스트 알고리즘을 기반으로 한 자료 의존성 그래프 생성 알고리즘을 구현 방법에 대해 설명한 뒤, 실험을 통해 정확성을 검증하였다.

Acknowledgement

본 연구는 교육과학기술부/한국과학재단 우수연구센터 육성사업(과제번호 2012-0000470), 2011 년도 정부(교육과학기술부)의 재원으로 한국과학재단의 국가지정연구실사업(No.2011-0018609) 및 IDEC 의 지원을 받아 수행되었습니다.

참고문헌

[1] Steven S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997

[2] M. Ahn and Y. Paek, “Fast Code Generation for Embedded Processors with Aliased Heterogeneous Registers,” in Transactions on High-Performance Embedded Architectures and Compilers II, 2009, vol.5470, pp. 149–172.

[3] Synopsys inc., http://www.synonpsys.com

[4] V. uZivojnovic, J. Martinez, V. C. Schlager, and H. Meyr, “DSPSTONE: A DSP-oriented benchmarking methodology,” in Proceedings of the International Conference on Signal Processing Applications and Technology, 1994.

[5] F. H. McMahon, “The Livermore Fortran kernels: A computer test of the numerical performance range,” Lawrence Livermore National Laboratory, Tech. Rep. UCRL-53745, 1986.