論文96-33B-4-16

RISC 컴파일러 상에서의 루프 합치기에 의한 코드 최적화 알고리듬

(A Code Optimization Algorithm by the Loop Fusion on RISC Compilers)

李 喆 源 *, 林 寅 七 **

(Chul-Won Lee and In-Chil Lim)

요 약

본 논문에서는 RISC 컴파일러의 코드 최적화 부에서 루프에 대한 효율적인 목적 코드를 생성하는 루프 구조 최적화 알고리듬을 제안한다. 프로그램 내에서의 반복 구조인 루프는 순차 프로그래밍 언어로 작성된 프로그램에서 수행 시간의 많은 시간을 차지하게 되며, 또한 이러한 반복 구조를 갖는 루프 외부의 코드를 최적화하는 것보다 루프의 수를 줄일 수 있으면, 사용되는 레지스터의 할당 문제를 보다 쉽게 해결함으로써 프로그램의 효율성을 향상시킬 수 있다. 제안된 알고리듬은 여러 개의 루프들을 각각 수행시키는 대신 각 루프의 종속성 및 범위를 조사하여 루프 합치기(loop fusion)가 가능한 경우 루프를 재 구성하여 하나의 루프로 합성함으로써 순차 프로그래밍 언어로 작성된 프로그램의 수행 시간의 감소를 기대할 수 있다. 제안된 알고리듬은 SUN SPARC-10 Workstation 상에서 C 언어로 구현하여 기계 독립적인 중간 코드에 대해 효율적인 출력 코드를 생성한다.

Abstract

A loop structure optimization algorithm is proposed for generating a set of efficient codes for loop structure in order to optimize RISC compiler codes. Since there are so many loop structure in the program, most of the execution time is used to process looping codes. Thus, reduction of loop instructions is more effective than optimizing codes outside the loop. The proposed algorithm presents a method to combine several different loops into a simple loop. Therefore, rather than executing each loop independently, loops in the program are serached, analyzed, and finally created some relative information such as dependency and range. In doing so, the loops in the program can efficiently be recombined and restructured. As a result, the overall execution time for the program of the sequential programming language is reduced.

I. 서 론

컴퓨터 산업이 발전해 온 이래로 컴퓨터 구조 영역에는 몇 가지 기술이 등장하여 사용되어 왔으며, 그중에서 매우 중요하고 가능성있는 기술 혁신 중의 하나가 RISC(Reduced Instruction Set Computer) 구조이다. 이러한 RISC 구조는 과거의 많은 구조들 보다 훨씬 낮은 가격으로 더욱 개선된 처리력과 개방성에 의해 점차로 산업 표준이 되어가고 있다[1-3].

이러한 RISC 명령어들은 간단하며 보통은 동일한 길이로 되어 있어 대부분의 명령어들을 단일 명령 사이클내에 처리할 수 있으며, 또한 대규모의 레지스터 화일을 이용하여 메모리 액세스 명령을 제외한 모든 명령어들을 레지스터 상에서 수행함으로서 보다 효율적인 생산성을 얻을 수 있다[1,4].

* 正會員, 斗源工業專門大學 電子科

(Dept. of Elec. Eng., DooWon Technical College)

** 正會員, 漢陽大學校 電子工學科

(Dept. of Elec. Eng., Hanyang University)

※ 본 연구는 '95년도 산학협동재단 지원으로 수행된 연구임.

接受日字 : 1995年5月10日, 수정완료일 : 1996年2月16日

그러나, RISC를 위한 컴파일러에서는 제한적이고 단순한 구조를 갖는 명령어로 코드를 생성하여야 되기 때문에 CISC(Complex Instruction Set Computer)에 비해서 코드 사이즈가 방대하여 지며, 또한 분기가 많은 작업에서는 효율이 매우 저하될 수 있으므로, RISC 구조를 위한 컴파일러 설계 시에는 최적화된 코드를 만들 수 있는 기법이 요구되고 있다[4-6].

최적화 기법에 대해서 많은 연구가 되어 오고 있으나, 수행 시간의 대부분은 루프 안의 명령들을 수행하는 데 사용한다는 사실로 미루어 루프를 최적화하는 작업은 프로그램의 효율성을 향상시킬 수 있어 컴파일러의 코드 최적화에 있어서 매우 중요한 부분이라고 할 수 있다. 루프를 최적화시키기 위한 방법은 루프의 명령들을 각각 다른 처리기에 할당하여 병렬 수행시키는 방법과 순차 프로그램 상에서 명령들을 재구성함으로써 최적화 하는 방법 등이 있다[7-8].

순차 프로그램이 많이 사용되는 점으로 미루어 순차 루프가 효과적으로 수행될 수 있도록 하는 방법의 연구가 필요하며, 루프의 구조를 분석하고 종속성에 대한 정보를 추출하여 이를 수행 과정에서 효율적으로 사용하는 것이 바람직하다. 또한, 순차 프로그래밍 언어로 작성된 프로그램에서 루프의 종속성에 대한 구조를 그래프로 표현하며, 컴파일러가 이 그래프로 이용하여 루프를 최적화시키고 코드의 생성을 위한 중간 언어로 사용하는 것이 효과적인 방법이다[9-10].

본 논문에서는 일련의 연속적인 명령어 들로 이루어진 순차적 프로그램 상에서 여러 개의 루프들을 하나의 루프로 단일화할 수 있는 경우 각각의 루프들을 수행시키는 대신 하나의 루프만 수행하게 하는 루프 최적화 알고리듬을 제안한다.

제안한 루프 최적화 알고리듬에서는 3 번지 형식을 사용한 중간 코드를 입력으로 받아 기본 블럭으로 분할하여 제어 흐름 그래프를 구성하고 각 문장에 대한 데이타 흐름 분석을 수행한다. 데이타 흐름 분석을 수행하여 구하여진 데이타 흐름 정보에 의해서 제어 흐름 그래프에서의 루프내의 문장들 간에 종속성을 찾아내고 종속성이 존재하지 않는 경우 각 루프들을 하나의 루프로 재 구성하여 수행시킴으로써 코드의 최적화를 얻을 수 있다. 또한, 여러 개의 루프를 수행시키는 대신 단일화된 루프만 수행시킴으로써 순차 언어로 작성된 프로그램의 수행 시간을 줄일 수 있으며, 각 루프에 제한된 최적화의 기회도 더 많이 제공될 수 있다.

제안한 알고리듬은 SUN SPARC-10 Workstation 상에서 C 언어로 구현하여 기계 독립적인 중간 코드에 대해 효율적인 출력 코드를 생성한다.

Ⅱ. RISC 컴파일러의 코드 최적화

최적화 기법은 최적화되어 지는 과정에 따라 기계 의존 최적화와 기계 독립 최적화로 나누어 볼 수 있으며, 또한 기본 블럭 안에 있는 문장들을 대상으로 수행하는 지역적 최적화와 그 이상을 기준으로 하는 전역적 최적화로 나눌 수 있고, 단일문 최적화와 루프 최적화로 나눌 수 있다[4].

최적화는 원시 언어 단계에서 중간 코드 생성 단계, 목적 코드 생성 단계까지 각 단계 별로 코드 개선 변환을 통하여 수행 시간을 줄이는 최적 효과를 얻을 수 있다. 원시 언어 단계에서는 사용자들이 프로그램 작성 시 주어진 알고리듬에 대해 변경을 하여 개선할 수 있는 명시적(explicit) 최적화를 구현할 수 있으나 사실상 효율적 코드 개선 변환을 구현하기는 매우 한정적이다. 또한, 목적 코드 생성 단계는 레지스터들을 유효하게 이용한다든지 명령어들을 기계어에 맞게 선택해서 기계 종속적인 최적화를 하기 때문에 코드 최적화 부 보다는 코드 생성기 부분에서 더 강조된다[5-6].

반면에 중간 코드 생성 단계에서는 코드 개선의 기회가 많아지며, 언어를 구현하는 데 있어 분석 과정을 거쳐 암시적(implicit) 부분에 대해 비효율적인 부분을 제거함으로서 개선의 효과를 기대할 수 있다. 최적화의 효율성 향상을 위하여 컴파일러가 가져야 할 특성은 다음과 같다. 첫째, 최적화는 프로그램의 의미를 보존함으로서 최적화가 끝난 후 원래 내용이 없어지거나 변형되어서는 안되며, 둘째 최적화 시 프로그램의 속도 향상이 이루어져야 하고, 셋째 최적화에 필요한 노력이 값어치가 있어야 한다[6].

최적화에 있어서는 실행 시간을 짧게하기 위한 최적화와 코드 크기를 작게하는 최적화 중 한가지에 목적을 정하게 되며, 이러한 두가지 최적화 기법은 일반적으로 동시에는 수행될 수 없으며, 또한 실제로 여러 컴파일러에서는 두가지 조건 중에서 한가지를 선택하게 되면 다른 조건들은 무시하게 된다. 그러나, 코드 크기가 작은 프로그램이 큰 프로그램보다 실행 시간이 빠르게 될 가능성이 높기 때문에 최적화의 기법에 따라서는 이 두가지 조건을 모두 만족 시킬 수 있으며, 코

드의 간결성을 가지고 있는 RISC에서는 코드 크기를 줄임으로 해서 실행 시간도 단축시킬 수 있다[5].

최적화부는 제어 흐름과 데이타 흐름 분석을 수행하는 부분과 최적화 작업을 수행하는 세가지 형태의 구성 요소로 이루어진다. 제어 흐름 분석은 기본 블럭을 단위로 수행하여 프로그램을 흐름 그래프로 표현하는 과정이며, 데이타 흐름 분석은 프로그램의 전체적인 정보를 모아서 이 정보를 흐름 그래프의 각 블럭에 제공하는 과정이다. 또한, 제어 흐름과 데이타 흐름 분석 요소에 의해 수집되는 정보들은 최적화부에서 효과적으로 이용될 수 있다[9-10].

Ⅲ. 제어 흐름 분석 및 데이타 흐름 분석

1. 제어 흐름 분석

제어 흐름도를 구성하는 각 노드는 기본 블럭(basic block)으로 구성되며, 기본 블럭이란 제어가 시작점으로 들어가서 바깥점으로 나올 때까지 중지나 분기가 없는 연속적인 문장들의 집합으로 이루어 진다. 기본 블럭으로 나누기 위해서는 기본 블럭의 첫번째 문장인 리더를 구하여야 하며, 리더들의 집합은 첫번째 문장이나 분기 명령의 대상이 되는 문장, 분기 명령어 다음에 오는 문장들로 이루어 진다. 각 리더들에 대한 기본 블럭은 리더 자신과 다음 리더나 프로그램 끝을 만나기 바로 전까지의 모든 문장들로 구성된다.

기본 블럭 집합의 제어 흐름 정보를 나타내는 흐름 그래프(flow graph)에서 각 노드는 기본 블럭을 의미하며, 연결선(edge)은 노드들의 수행 순서를 의미한다. 수행 순서가 블럭 B1 다음에 블럭 B2라면 B1에서 B2로 향하는 연결선이 존재하며, 블럭 B1은 블럭 B2의 predecessor node라 하고, 블럭 B2는 블럭 B1의 successor node라고 한다. B1에서 B2로 향하는 연결선이 존재하기 위해서는 다음과 같은 조건을 만족하여야 한다.

① B1의 마지막 문장이 B2의 첫번째 문장으로 분기

② 프로그램 순서가 B1 바로 다음이 B2이며, B1의 마지막 문장에 무조건 분기문이 존재하지 않는 경우

그림 2는 그림 1의 프로그램에 대한 중간 코드 및 제어 흐름도를 나타낸 것이다.

이러한 제어 흐름도에서의 루프 구조는 루프의 노드를 지배하는 루프의 헤더(loop header)와 루프의 몸체(loop body), 루프의 꼬리(loop tail)로 이루어 진다.

```
main()
{
        register int top, i, tmp, list[70];
        top = 70;
        while( top > 1 ) {
            top = top - 1;
            i = 0;
            while( i < top ) {
                if( list[i] > list[i+1] ) {
                    tmp = list[i];
                  list[i] = list[i+1];
                  list[i+1] = tmp
                }
              i = i + 1;
            }
        }
}
```

그림 1. 예제 프로그램 - Bubble 정렬
Fig. 1. Example Program - Bubble Sort.

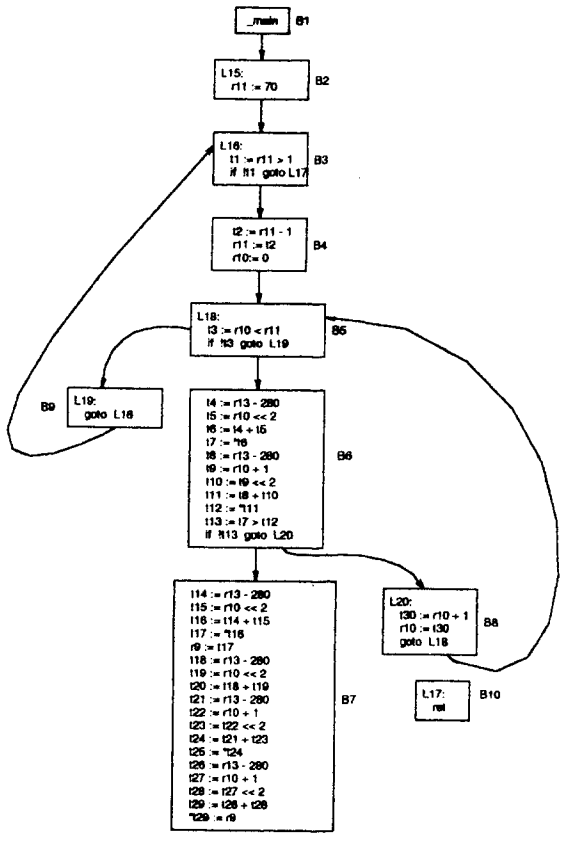

그림 2. 그림 1에 대한 제어 흐름도
Fig. 2. Control Flow Graph of Fig. 1.

또한, 제어 흐름도에서 초기 노드로 부터 노드 n까지 도달할 때 모든 경로는 노드 d를 거쳐간다면 노드

d는 노드 n을 지배한다고 하며(dominate), 'd dom n'으로 표시한다. 모든 노드는 자기 자신을 지배하며, 한 루프의 입구 노드는 그래프의 모든 노드를 지배한다. 루프는 이러한 지배자(dominator) 정보를 이용하여 발견할 수 있으며 다음과 같은 성질을 갖는다.

① 루프는 헤더라고 하는 단일의 엔트리를 가져야 한다. 이 엔트리는 루프내의 모든 노드를 지배한다.

② 루프 내의 모든 노드는 최소한 한번은 반복 실행할 수 있는 경로가 있어야 한다. 루프는 이러한 경로를 빠져나와 헤더에 다시 연결되는 경로를 의미한다.

제어 흐름도에서의 모든 루프는 꼬리를 지배하는 헤더가 속해 있는 연결선이며, 이러한 연결선을 역 연결선(back edge)이라고 한다. 그림 3은 제어 흐름도에서 루프를 찾는 알고리듬을 나타낸 것이다.

```
Procedure Loop_Search()
{
    while( each node N of flow graph ) {
        SUCC = successor node of node N;
        PRED = predecessor node of node N;
        while( SUCC != NULL ) {
            num = number_of_node of SUCC;
            if( Back_Edge(N,num) == TRUE )
                generate loop;
            SUCC = next node;
        }
    }
}
```

그림 3. 루프 발견 알고리듬
Fig. 3. Loop Search Algorithm.

2. 데이타 흐름 분석

기본 블럭의 집합으로 구성된 루프의 최적화를 수행하기 위해서는 변수들의 정의(definition) 및 사용(use)에 대한 정보를 알아야 하며, 각 문장에서 수집된 정보들은 집합의 방정식을 사용하여 서로 관련되어 지며, 루프의 상호 종속성을 구분하는 데 이용된다.

변수 x에 대한 정의는 변수 x에 값을 대입하는 할당문이고, 사용은 계산식에서 현재의 변수 x의 값을 읽어 내는 것이다. 문장 d에서 p 지점까지 경로가 존재하고 이 경로 중에 정의 d가 소멸되지 않으면 정의 d는 정의 p에 도달(reach)된다고 한다. 다음 식은 초기 노드에서 p 지점 까지의 변수의 생성 및 제거에 관한 식을 나타낸 것이다.

- e_gen [B] : 블럭 B에 의해서 생성된 사용 가능 식
- e_kill [B] : 블럭 B에 의해서 제거된 사용 가능 식

또한, in [B]는 블럭 B의 시작 직전에서 사용 가능한 식의 집합을 나타내며, out [B]는 블럭 B의 종료 후에 사용 가능한 식의 집합을 나타낸다. 임의의 식이 한 블럭의 처음에서 사용 가능하기 위해서는 그 블럭의 노드 끝 지점에서 사용 가능하여야 한다. 다음 식은 in [B] 및 out [B]에 대한 정의식이며, 초기 노드 값 in [B1]은 사용 가능 식이 없는 공집합으로 가정한다.

- out [B] = (in [B] - e_kill [B]) ∪ e_gen [B]
- in [B] = out [B] ; B가 초기 블럭이 아닐 때
- in [B1] = ϕ ; B1이 초기 블럭일 때

3. live 변수 분석

live 변수 분석은 변수 x와 포인터 p에 대해 p에서의 변수 x의 값이 p에서 시작하는 흐름 그래프에서 임의의 경로를 따라 사용될 수 있는 지에 대한 정보를 구하는 과정이다. 만일 x의 값이 p 지점에서 사용될 수 있다면 x는 p 지점에서 live하다고 말하며, 또한 p 지점에서 사용될 수 없다면 변수 x는 p 지점에서 dead 되었다고 한다.

in [B]를 블럭 B 바로 직전의 점에서 live한 변수 집합이 되도록 하고, out [B]를 블럭 B 바로 직후의 점에서 live한 변수의 집합이 되도록 정의한다. def [B]를 블럭 B 안에서 그 변수의 사용 이전에 블럭 B 안에서 값이 명확히 할당된 변수 집합이라 하고, use [B]를 그 값이 그 변수의 임의의 정의 이전에 블럭 B 안에서 사용될 수 있는 변수들의 집합이라고 하면, in [B] 및 out [B]를 계산하는 식은 다음과 같다.

- in [B] = use [B] ∪ (out [B] - def [B])
- out [B] = ∪ in [S] ; S는 B의 successor

이 방정식은 임의의 변수가 블럭 안에서 재 정의되기 이전에 사용되거나, 또는 그 블럭으로 부터 나오면서 live하고 그 블럭 안에서 재 정의되지 않는다면 그 변수는 블럭으로 들어가면서 live하고, 임의의 변수가 그 변수의 successor 중 하나의 시작에서 live할 때

그 변수가 현재의 기본 블럭 끝에서 live하다는 것을 의미한다.

4. 사용-정의 고리

사용 정의 고리(use-definition chain)는 임의의 변수 사용마다 그 변수에 도달하는 모든 정의를 리스트로 표현한 것으로서, 변수가 이전의 어느 정의에 의한 것인지 알 수 있도록 정의한다. 블럭 B 내에서 변수 a가 사용될 때 그 전에 a의 정의가 존재하지 않는다면 a의 사용에 대한 사용 정의 고리는 in [B] 안에서 a의 정의 집합이 되며, 또한 만약 블럭 B 안에서 a의 사용 전에 a의 정의가 존재한다면 a의 사용 정의 고리는 a의 마지막 정의가 된다. 또한, in [B] 는 사용 정의 고리에 포함되지 않는다. 그림 4는 사용 정의 고리를 구하는 알고리듬을 나타낸 것이다.

```
Procedure Make_UD_Chain(Use, Definition)
{
    if( ud_chain[Use].i == NULL )
        ud_chain[Use].i = Definition;
    else {
        temp_node = ud_chain[Use].next_node;
        ud_chain[Use].i of next_node = Definition;
        ud_chain[Use].next_node of next_node = temp_node;
    }
}
```

그림 4. 사용 정의 고리 알고리듬
Fig. 4. Use Definition Chain Algorithm.

Ⅳ. 코드 최적화 알고리듬

본 논문에서 제안한 최적화부는 컴파일러의 전반부에서 C 언어 프로그램을 입력으로 받아 출력한 중간 코드를 입력으로 하여 최적화 알고리듬을 수행하여 최적화된 코드를 출력한다. 언어 종속적인 전반부와 기계 종속적인 후반부를 연결하는 역할을 담당하는 중간 언어의 선택은 최적화부에서 효율성에 많은 영향을 줄 수 있으므로 이식성 및 독립성이 용이한 3번지 코드를 중간 언어로 사용한다.

루프 합치기 기법은 순차 프로그램 내의 루프들을 중간 코드에 대하여 제어 흐름 분석을 수행하여 기본 블럭으로 나누고, 이러한 기본 블럭을 단위로 하여 작성된 제어 흐름도 상에서 발견된 루프에 대해 데이타 흐름 분석을 수행하는 동안 모여진 정보들을 바탕으로 구현된다. 종속성을 이용한 본 알고리듬은 그림 5에 나타낸 흐름도와 같이 수행된다.

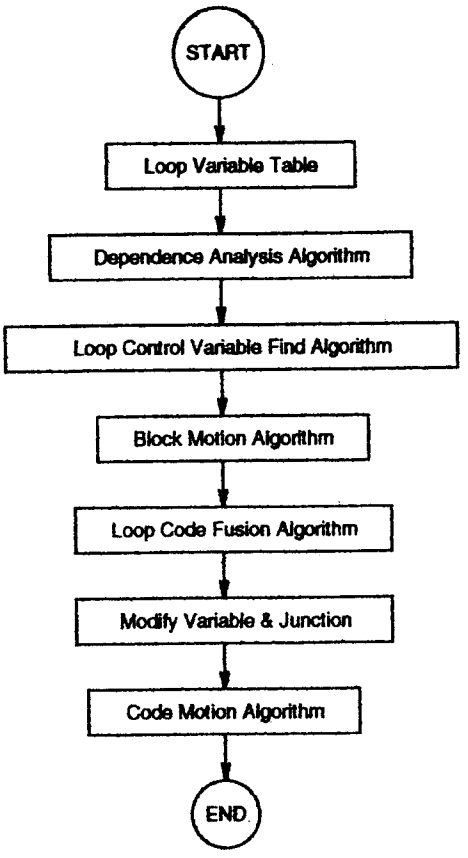

그림 5. 제안된 코드 최적화 알고리듬의 흐름도
Fig. 5. Flow Chart of the Proposed Code Optimization Algorithm.

제어 흐름도 상에서 발견된 루프들의 변수 테이블을 비교하여 루프 합치기를 할수 있는 종속성을 검사하고 루프 제어 변수 알고리듬에 의해 루프의 범위를 비교한다. 루프가 인접한 형태로 구성되도록 블럭 옮기기를 수행한 후 루프 합치기를 실행하고 변수의 수정 및 분기점의 변경을 행한다. 합쳐진 루프 내에서 불변식을 발견하면 루프의 프리 헤더로 불변식을 이동시킨다.

두 개의 루프가 합쳐지기 위해서는 루프 내의 기본 블럭들의 변수들이 상호 독립적이어야 하며, 종속성은 크게 일련의 제어 흐름인 제어 종속과 자료 흐름인 자료 종속의 서로 상이한 형태로 나타난다. 두 개의 식에서 한 식의 실행이 다른 식의 실행 결과에 조건적이면 제어 종속적이며, 이와 같은 형태의 종속성을 제어 종속성이라고 한다.

또한, 2 개의 문장을 교환하였을 때 한 문장에 사용

된 변수가 다른 값을 갖게 되면 자료 종속적이며, 이와 같은 형태의 종속성을 자료 종속성이라고 한다. 이러한 종속성의 여부는 데이타 흐름 분석을 통해서 결정되어 진다.

루프 제어 변수(loop control variable)는 루프가 매번 수행될 때마다 임의의 변수 x의 값이 일정한 양만큼 증가하거나 감소하는 루프의 귀납 변수(induction variable) 중에서 반복 수행을 제어하는 곳에서 사용되는 변수이고, 루프 제어 변수 값이 참인 동안에만 반복 수행을 계속 할 수 있으며, 루프 합치기를 하기 위해서는 각 루프의 제어 변수가 가지고 있는 범위가 동일하여야 한다. 그림 6은 루프 제어 변수를 찾기 위한 알고리듬을 나타낸 것이다.

```
Procedure Find_Loop_Cont_Var()
{
   while( ∀ loop ) {
      for( ∀ statement i for loop ) {
         if( op_code of i = ':=' ) {
            cl = left for i;
            cr = right for i;
            tmp = right for cr;
            if( ( cl == left for cr ) &&
                ( op_code of cr == binary_op_code ) ) {
               op_code of tmp= 'constant';
               set_of_induction_var = cl;
               set_of_induction_eq_basic = i;
            }
         }
      }
   }
   loop_cnt_var = header_induction_var on set_of_induction_var;
}
```

그림 6. 루프 제어 변수 발견 알고리듬
Fig. 6. Loop Control Variable Find Algorithm.

```
Procedure Block_Motion()
{
    for( ∀ statement for basic block ) {
       switch( operand ) {
          case constant : write invariant;
          case Def_on_loop1 : write variant1;
          case Def_on_loop2 : write variant2;
       }
    }
    if( ∀ statement == invariant && variant1 )
       independence on Loop2;
    else if( ∀ statement == invariant && variant2 )
       independence on Loop1;
    else
       dependence Loop;
    if(independence on Loop1 )
       Up_Block_Motion for Loop1;
    else if(independence on Loop2 )
       Down_Block_Motion for Loop2;
}
```

그림 7. 블럭 옮기기 알고리듬
Fig. 7. Block Motion Algorithm.

그림 7은 루프 사이의 기본 블럭에 대해서 루프와 관련없는 계산식을 찾기 위한 알고리듬이며, 비록 블럭 옮기기를 수행했다 하더라도 포인터로 표현하는 사용 정의 고리 정보는 변화되지 않고 블럭을 옮기기 전의 정보를 유지할 수가 있다.

블럭 옮기기에 의해서 흐름 그래프 상의 두 루프가 서로 인접한 형태로 구성되었으면 두 루프 간의 합치기를 수행하며, 이때 각 루프의 루프 제어 변수 및 귀납 변수를 하나의 공통된 변수로 만들어 주며, 또한 분기점 및 변수 이름을 대치한다. 그림 8은 루프 코드 합치기 알고리듬을 나타낸 것이다.

```
Procedure Code_Fusion()
{
    for( loop statement ) {
          header = header1;
       body = body1 ∪ body2;
       tail = tail1;
    }
    header_Junction = header2_Junction;
    for( body statement i ) {
       if( i.operand == loop_cnt_var2 )
          i.operand = loop_cnt_var1;
    }
}
```

그림 8. 루프 코드 합치기 알고리듬
Fig. 8. Loop Code Fusion Algorithm.

루프 내에서 루프와 관련없는 식(loop invariant statement)들을 발견하면 코드 위치 옮기기 알고리듬을 수행하여 루프내의 계산 횟수를 줄인다. 코드 위치 옮기기를 수행하기 위해서는 데이타 흐름 분석에 의해 사용되고 있는 변수들이 어느 정의에 의한 것인지를 나타내고 있는 사용 정의 고리를 계산해서 불변식(invariant statement)들을 결정한다. 그림 9는 코드 위치 옮기기 알고리듬을 나타낸 것이다.

```
Procedure Code_Motion()
{
    for( loop statement i )
       if( i == invariant_statement )
          Move i to Free_Header_of_loop;
}
```

그림 9. 코드 위치 옮기기 알고리듬
Fig. 9. Code Motion Algorithm.

V. 수행 결과 및 분석

본 논문에서 제안한 코드 최적화 기법은 SUN

SPARC-10 상에서 C 언어로 구현하여 그림 10에서의 예제 프로그램에 대한 중간 코드에 적용한다.

```
main()
{
    register int i,j,s;
    register int k[10];
    i = 0;
    while( i < 10 ) {
        k[i] = 0;
        i = i + 1;
    }
    s = 0;
    j = 0;
    while( j < 10 ) {
        s = s + 1;
        k[j] = j + 1;
    }
}
```

그림 10. 예제 프로그램
Fig. 10. Example Program.

기존의 루프 최적화 알고리듬[6]에서는 루프 내에서 불변식을 발견하면 코드 옮기기를 이용한 최적화와 공통 부분식 제거, 값싼 연산 만들기, dead 코드 제거와 같은 지역적 최적화를 수행하거나 또는 병렬 처리를 위한 루프 구조 분석을 주로 수행하였으나, 아직까지는 순차적 프로그램을 많이 사용하는 중요성으로 미루어 전역적 최적화 기법인 루프 코드 최적화는 프로그램의 수행에 있어 매우 향상된 효과를 얻을 수 있다.

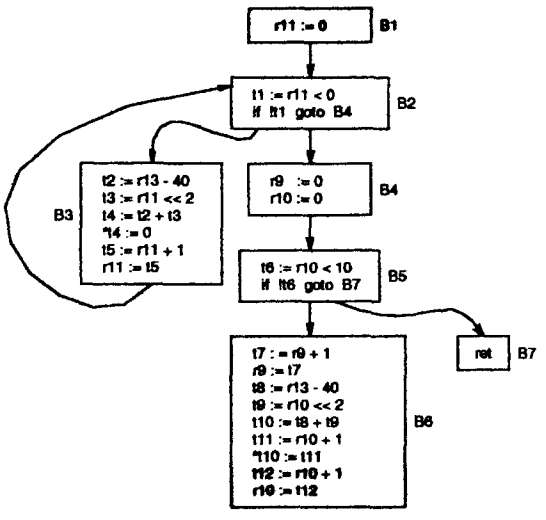

그림 11. 제어 흐름도에서의 변수 정의
Fig. 11. Variable Definition of Control Flow Graph.

그림 11은 그림 10의 예제 프로그램에 대한 제어 흐름도와 변수의 정의를 나타낸 것이다.

제어 흐름도 상에서 데이타 흐름 분석을 통해 변수들의 종속성을 알아 낼 수 있으며, 그림 12는 블럭 옮기기를 수행한 것이다.

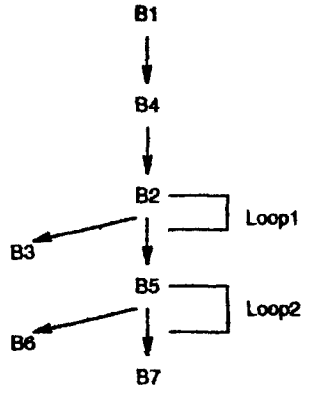

그림 12. 블럭 옮기기 수행
Fig. 12. Performance of Block Motion.

그림 13은 루프 합치기를 수행한 후의 최적화 코드를 나타낸 것이다. 최적화를 하기 전에는 제어 흐름도 상에 2개의 루프가 존재하였으나, 최적화를 통해 루프를 1개로 줄임으로써 코드 크기 및 실행 횟수의 감소를 기대할 수 있다.

```
L15:
  r11 := 0
  r9 := 0
  r10 := 0
L16:
  t1 := r11 < 10
  if !t1 goto L19
  t2 := r13 - 40
  t3 := r11 << 2
  t4 := t2 + t3
  *t4 := 0
  t7 := r9 + 1
  r9 := t7
  t8 := r13 - 40
  t9 := r11 << 2
  t10 := t8 + t9
  t11 := r11 + 1
  *t10 := t11
  t5 := r11 + 1
  r11 := t5
  goto L16
L19:
  ret
```

그림 13. 최적화 코드
Fig. 13. Optimization Code.

Ⅵ. 결 론

본 논문에서는 RISC 컴파일러의 최적화 효율을 향상시킬수 있는 흐름 그래프 상에서의 루프 합치기 알고리듬을 제안하였다.

제안한 알고리듬은 원시 프로그램에 대해 중간 코드를 생성하여 입력으로 사용하며 중간 코드는 3 번지 형식으로 표현하였다. 중간 코드에 대해 제어 흐름도를 작성하고 기본 블럭을 바탕으로 데이타 흐름 분석 알고리듬을 적용하여 루프 합치기에 필요한 정보들을 구한 후, 이 정보들을 이용하여 각 루프간의 종속성 여부를 확인한 후 블럭 옮기기를 통한 루프 인접 상태에서 루프 합치기를 수행하였다. 프로그램 상에서 루프들을 각각 수행하는 것보다 하나의 단일된 루프로 실행함으로써 코드의 크기가 감소하였고, 또한 반복 회수를 줄임으로서 실행 시간의 감소도 기대할 수 있었다.

앞으로의 연구 과제는 최적화에 있어 실현 효과가 크지 않은 지역적인 부분보다는 루프와 같은 전역적 최적화를 구현할 수 있도록 연구가 계속 이루어 져야 하며, 또한 여러가지의 형태를 갖는 중첩 루프에 있어서의 최적화를 구현하는 것이다.

참 고 문 헌

[1] W.Stallings, "Reduced Instruction Set Computer Architecture," Proc. IEEE, Vol. 76, No. 1, pp. 38-55, Jan. 1988.

[2] C.G.Bell, "RISC : Back To The Future," DATAMATION, Jun. 1, 1987.

[3] C.E.Gimarc and V.M.Milutinovic, "A Survey of RISC Processors and Computers of the Mid-1980s," Computer, pp. 59-69, Sep. 1987.

[4] S.S.Muchnick, "Optimizing Compilers for SPARC," Sun Tech., pp. 64-77, Summer 1988.

[5] J.W.Davidson and R.A.Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," Proc. 2nd Inter. Conf. on Architectural Support for Program Language and Operating System, pp. 60-64, Oct. 1987.

[6] J.M.Bishop, "The Effect of Data Abstraction on Loop Programming Techniques," IEEE Trans. on Software Eng., Vol. 16, No. 4, pp. 389-402, Apr. 1990.

[7] R.Leveugle and G.Saucier, "Optimized Synthesis of Concurrently Checked Controllers," IEEE Trans. on Computers, Vol. 39, No. 4, pp. 419-425, Apr. 1990.

[8] C.D.Polychronopoulos, "Compiler Optimization for Enhancing Parallelism and Their Impact on Architecture Design," IEEE, Vol. 37, No. 8, pp. 991-1004, Aug. 1988.

[9] C.Wilson and L.J.Osterwell, "Omega-A Data Flow Analysis Tool for the C Programming Language," IEEE Trans. on Software Eng., Vol. 11 No. 9, pp. 832-838, Sep. 1985.

[10] B.G.Ryder and M.C.Paull, "Elimination Algorithms for Data Flow Analysis," ACM Computer Survey, Vol. 18, No. 3, pp. 277-316, Sep. 1986.

[11] 박종득, "RISC 시스템의 기계독립적인 컴파일러를 위한 코드 최적화 기법에 관한 연구," 한양대학교 박사학위 논문, 1990

저 자 소 개

李 喆 源(正會員) 第 28卷 B編 第 6號 參照.
현재 두원공업전문대학 전자과 전임강사.

林 寅 七(正會員) 第 28卷 第 4號 參照.
현재 한양대학교 전자공학과 교수.