에필로그 테일러된 프로시저를 위한 프롤로그 테일러링 기법

지 윤 찬[†] · 김 기 창[††]

요 약

본 논문은 컴파일러에 의해 생성된 목적 코드상의 프로시저의 수행 속도를 향상시키기 위한 방안으로, 프롤로그 테일러링 알고리즘을 제안한다. 레지스터의 수가 많은 기계에서 반복 수행되는 프로시저의 경우, 프롤로그와 에필로그에서 실행되는 레지스터 저장 및 복원 명령어를 줄이는 것은 프로시저 실행 속도 향상의 주요 작업이 된다. IBM XL C 컴파일러에서 제공되는 에필로그 테일러링은 실행 경로상의 레지스터 복원 명령어를 줄임으로써, 프로시저의 성능 향상에 기여해왔으나, 프롤로그 테일러링에 대한 구체적 알고리즘은 현재 제안되어 있지 않다. 본 논문이 제안하는 알고리즘에 의해 생성된 프롤로그는 각각의 실행 경로에 대해 현저히 감소된 수의 레지스터 저장 명령을 실행하도록 함으로써, 프로시저의 실행 속도를 개선시킨다. 테일러된 프롤로그를 갖는 프로시저가 바르게 실행되기 위해서는, 다이아몬드 구조나 반복 구조 내부에 레지스터 저장 명령을 생성해서는 안된다. 그러므로, 본 논문은 다이아몬드 구조나 반복 구조 내부가 아닌 최적의 위치에 레지스터 저장 명령을 생성하는 알고리즘을 제안한다.

Prolog Tailoring Technique on Epilog Tailored Procedures

Yoon-Chan Jhi[†] · Ki-Chang Kim[††]

ABSTRACT

Prolog tailoring technique, an optimization method to improve the execution speed of a procedure, is proposed in this paper. When a procedure is frequently and repeatedly called and the machine has a lot of callee-saved registers, optimizing prolog and epilog can become an important step of optimization. Epilog tailoring supported by IBM XL C Compiler has been known to improve procedure's execution speed by reducing register restore instructions on execution paths, but no algorithms for prolog tailoring has been proposed yet. The prolog generated by the prolog tailoring algorithm proposed in this paper executes considerably smaller number of register save instructions at run-time. This means the total number of instructions to be executed is decreased resulting in an improvement on the procedure's execution speed. To maintain the correctness of code, prolog code should not be inserted inside diamond structures or loop structures. This paper proposes a prolog tailoring technique which generates register save instructions at the best position in a control flow graph while not allowing the insertion of any prolog code inside diamond structures or loop structures.

1. 서 론

※ 본 연구는 한국과학재단의 '97 핵심전문연구 과제 연구비에 의해 수행되었음.
† 준 회 원 : 인하대학교 전자계산공학과
†† 정 회 원 : 인하대학교 전자계산공학과 교수
논문접수 : 1997년 11월 28일, 심사완료 : 1998년 3월 10일

프로시저의 호출에 있어서, 호출된 프로시저는 피호출자 보존 레지스터(callee-saved registers) 중, 자신에 의해 소멸되는 레지스터의 값을 프로시저의 스택 프레임에 저장하는 작업과 복원하는 작업을 수행한다 [3]. 소멸 레지스터의 저장을 수행하는 부분을 프롤로

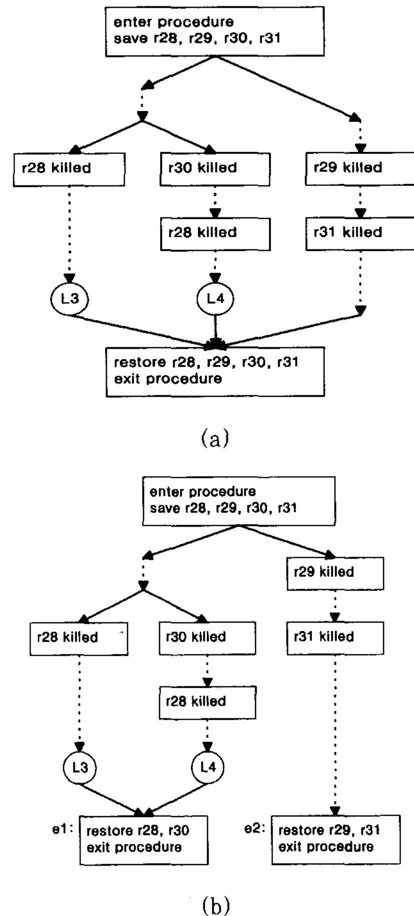
그(prolog)라고 하며 저장된 레지스터 값의 복원을 수행하는 부분을 에필로그(epilog)라고 한다. 프롤로그와 에필로그는 일반적으로 프로시저의 목적 코드의 시작과 끝부분에 위치하며, 프로시저 내부에서 소멸되는 피호출자 보존 레지스터들에 대해 일괄적인 저장과 복원을 수행한다. 프롤로그와 에필로그는 프로시저 호출에 있어서 필수적으로 실행되는 코드이므로, 빈번히 반복되어 호출되는 프로시저의 경우 프롤로그와 에필로그의 실행에 소모되는 시간은 크게 증가한다[1].

프롤로그와 에필로그에서의 작업을 필요로 하는 레지스터들은 프로시저가 어느 경로를 통해 실행되느냐에 따라 종종 소멸되지 않고 그 값이 유지되는 경우를 보인다. 이러한 레지스터들에 대해 그 내용을 저장하고 복원하는 작업은 의미가 없으며, 프로시저 실행 효율의 손실을 가져온다. 기존의 생성 기법에 의해 생성된 프롤로그와 에필로그는 프로시저의 실행 경로와 관계 없이, 내용의 보존을 필요로 하는 모든 레지스터의 값을 프로시저의 시작과 끝에서 일괄적으로 저장하고 복원하므로, 실행경로에 따른 소멸 레지스터의 특성을 반영하지 못한다. 프로시저의 실행경로에 따라 소멸 레지스터가 각기 다른 특성을 반영한 테일러링 기법으로는, 현재, IBM XL C 컴파일러에서 제공되는 에필로그 테일러링 기법(epilog tailoring)이 제안되어 사용되고 있으며[1][2], 프롤로그 테일러링(prolog tailoring)에 대한 구체적 알고리즘은 아직 제안되어져 있지 않다.

본 논문은 프로시저 실행 효율의 향상을 목적으로, 에필로그 테일러링이 이루어진 프로시저에 대해 현저히 감소된 수의 레지스터 저장 명령어를 생성하는 프롤로그 테일러링 기법을 제안한다. 2장에서는 기존의 에필로그 테일러링 기법을 다루고 있으며, 3장에서 프롤로그 테일러링의 전반적인 내용과 그 문제점에 대해 논한다. 4장에서 에필로그 테일러된 프로시저 상에서의 프롤로그 생성 알고리즘을 제안하며, 5장에서는 제안된 알고리즘에 의해 프롤로그가 생성되는 실례를 보인다. 6장에서는 실험된 결과에 대해서 논의하고자 한다.

2. 에필로그 테일러링

에필로그 테일러링은 에필로그의 실행에 소모되는 시간을 줄이기 위한 테일러링 과정이다. 에필로그 테일러링 기법은 일반적으로 하나인 프로시저의 탈출점(exit point)을 둘 이상으로 분리하면, 각 탈출점은 서로 다른 소멸 레지스터 집합을 갖게 되며, 따라서, 각 탈출점에 대해 분화된 에필로그를 생성할 수 있다는 점에 착안한 기법이다. (그림 1)은 기존의 기법에 의해 생성된 에필로그에 에필로그 테일러링을 적용하는 모습을 보이고 있다. (a)는 기존의 기법에 의해 에필로그를 생성한 프로시저의 제어 흐름 그래프(control flow graph)[3]를 나타낸다. 프로시저의 내부에서 레지스터 r28, r29, r30, r31이 소멸되고 있으며, 이에 대한 프롤로그와 에필로그가 각각 프로시저의 진입점(entrance point)과 탈출점에 생성되어 있다. (a)의 프로시저는 실제로 실행되는 경로와는 무관하게 항상 4개의 레지스터 복원 명령을 수행해야만 한다. (b)는 (a)의 탈출점을 둘로 분리하여, 각각의 탈출점에 에필

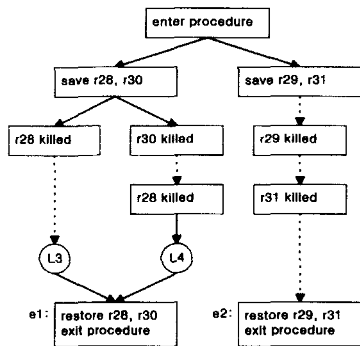


(그림 1) 에필로그 테일러링의 적용
(Fig. 1) Applying epilog tailoring technique on a general procedure.

로그를 생성한 결과를 나타낸다. Exit-1에 도달하는 경로에 대해서는 r28과 r30이, exit-2에 도달하는 경로에 대해서는 r29와 r31이 각각 소멸된다. 각 탈출점에 대해 서로 다른 소멸 레지스터 집합이 구성되므로, 각 탈출점에는 서로 다른 에필로그가 생성된다. (b)와 같은 에필로그 생성 기법은 모든 실행 경로에 대해 실제 실행되는 레지스터 저장 명령을 둘씩 감소시킨다. 에필로그 테일러링은 (b)에 나타난 바와 같이, 프로시저의 탈출점을 분화하고, 분화된 각 탈출점에 도달하는 경로에 대해 각각의 소멸 레지스터 집합에 대한 복원 명령을 생성하여, 실제 실행되는 레지스터 복원 명령의 수를 줄이는 기법이다.

3. 프롤로그 테일러링

본 논문에서는 실제 실행 경로상의 레지스터 저장 코드를 현저히 줄일 수 있는 프롤로그 테일러링에 대한 기법을 제안하고자 한다. 프롤로그 테일러링의 원리는 에필로그 테일러링과 유사하다. 기존의 기법에 의해 생성된 프롤로그의 레지스터 저장명령은 프로시저의 시작 부분에 위치한다. 이러한 레지스터 저장 명령을 프로시저의 실행 경로를 따라 실제 레지스터의 소멸 위치에 접근시키면, 프로시저는 각 실행 경로에 대해 불필요한 레지스터 저장 명령을 실행하지 않고 실행될 수 있다. (그림 2)는 (그림 1)의 (b)에 프롤로그 테일러링이 적용된 형태를 나타낸다. (그림 1)의 (b)는 진입 지점에서 4개의 레지스터 보존 명령을 실행하지만 (그림 2)의 프로시저는 모든 실행 경로에 대해 각각 2개의 레지스터 보존 명령을 수행하고 있다. 프로시저가 어느 경로를 선택해 실행되더라도 (그림 2)의 프로시저는 4개의 명령어만으로 레지스터 저장과 복원을 수행하는 반면, 같은 목적을 달성하기 위해 (그림 1)의 (b)는 6개 혹은 8개의 명령어 실행을 필요로 한다.

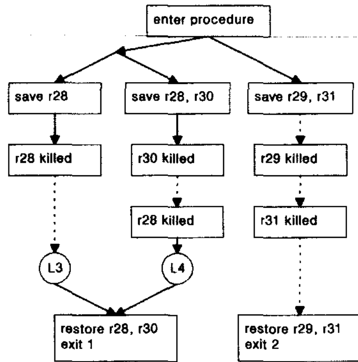


(그림 2) 프롤로그 테일러링의 적용
(Fig. 2) Applying prolog tailoring technique on an epilog tailored procedure.

프롤로그 테일러링에 있어서 제기되는 중요한 문제 중 하나는 레지스터 저장 명령을 실제 레지스터가 소멸되는 위치에 어느 정도까지 접근시켜 생성할 수 있느냐는 것이다. 만약, 어떤 기본 블록(basic block)[3]이 레지스터 r1을 소멸시킨다면, 레지스터 r1에 대한 저장 명령은 그 기본 블록의 앞까지 접근할 수 있다. 만약, 레지스터가 공통의 부모 노드를 갖는 둘 이상의 기본 블록(basic block)들에서 소멸된다면, 코드의 중복을 피하기 위해, 레지스터 저장 명령은 공통의 부모노드에 해당하는 기본 블록의 앞까지 접근할 수 있다. 또한, 레지스터가 루프 내부에서 소멸될 경우, 레지스터 저장 명령은 루프의 앞까지 접근할 수 있다. 루프 내부에 생성된 레지스터 저장 명령은 반복 실행되는 동안 매번 같은 레지스터의 다른 값을 보존한다. 또한, 이러한 실행은 CPU 시간을 소모하게 되므로, 루프의 내부로 레지스터 저장명령을 이동하는 것은 피해야 한다. 그 외의 경우, 레지스터 저장 명령은 실제 레지스터가 소멸되는 위치에 최대한 근접하는 것이 바람직 하다.

레지스터 저장 명령의 잘 못 된 이동이 일으킬 수 있는 몇 가지 문제는, 기본블록제어흐름그래프 상의 다이아몬드 구조와 반복 구조에서의 문제로 나타난다. 다이아몬드 구조는 최 외곽의 if-then-else 구문이 구성하는 것과 유사한 구조를 의미하며, 반복 구조는 최외곽의 루프에 의해 만들어지는 구조를 의미한다.

(그림 3)은 (그림 2)의 레지스터 저장 명령을 실제 레지스터가 소멸되는 위치에 더욱 근접시킨 예를 보여주고 있다. Exit 1에 이르는 두 경로 중 L3를 통과하는 경로에서는 r28만이 소멸되며 L4를 통과하는 경로에서는 r28과 r30이 소멸된다. 그러므로, L3를 통과하는 경로에는 r28에 대한 레지스터 저장 명령이 생성되었으며, L4를 통과하는 경로에는 r28과 r30에 대한 레지스터 저장 명령이 생성되었다. 레지스터 저장 명령의 이동을 통해, exit 1에 이르는 경로 중 L3를 통과하는 경로에 대해 레지스터 저장 명령어가 하나 감소되었음을 알 수 있다. 그러나, 프로시저가 실행되어 exit 1에 도달하게 되면, r28과 r30을 모두 복원하게 되는데,



(그림 3) 다이아몬드 구조 내부에 생선된 레지스터 저장 명령의 문제점
(Fig. 3) A problem with register saving codes generated inside a diamond structure.

(그림 3)에 나타난 바로는 이 시점에서 r30의 값이 반드시 저장되어 있다고 단정할 수 없다. 즉, exit 1의 레지스터 복원 명령어는 잘 못 된 값을 r30에 기록할 수 있으므로, (그림 3)에 나타난 레지스터 저장 명령의 이동에 의해 얻어진 코드는 올바르게 실행된다고 할 수 없다. 올바른 코드를 얻을 수 없게 된 원인은 다이아몬드 구조 내부에 프롤로그 코드를 생성했기 때문이며, 반복 구조 내부에 프롤로그를 생성하는 것 역시 유사한 문제를 야기한다.

본 논문은 다이아몬드 구조나 반복 구조 내부에 레지스터 저장 명령어를 생성하지 않으면서, 실제 레지스터가 소멸되는 기본 블록에 가장 근접한 위치에 레지스터 저장 명령어를 생성할 수 있는 프롤로그 테일러링 알고리즘을 제시한다.

4. 프롤로그 테일러링 알고리즘

본 논문은 프롤로그를 생성하고자 하는 프로시저에 대한 기본블록 제어흐름 그래프 (basic block control flow graph)〔3〕가 이미 생성되어 있다고 가정한다. 주어진 프로시저의 테일러된 프롤로그를 생성하기 위해서 해야 할 일은, 우선, 모든 다이아몬드 구조와 반복 구조를 하나의 노드로 대체함으로써 흐름 그래프 상에서 제거하는 것이다. 다이아몬드 구조와 반복 구조를 제거한 흐름 그래프는 트리 구조를 형성하게 된다. 이 흐름 그래프 상의 각 노드마다 소멸되는 레지스터를 계산하고, 계산된 소멸 레지스터 집합에 의해 레지스터 저장 명령어가 생성될 지점이 결정된다. 전체적인 알고리즘은 (그림 4)와 같다. 본 장의 각 절은 알고리즘의 세부 단계를 다루도록 한다.

```
basicblockfg_t generate_prolog(basicblockfg_t bbfg)
{
    sccfg기본블록 제어흐름그래프에서 반복 구조를 제거한
        그래프. ;
    bccfgsccfg의 다이아몬드 구조를 제거한 그래프. ;
    dkrbccfg의 각 노드에 대해 DKR(Definitely Killed
        Registers)을 계산한다. ;
    tailored_bbfgdkr과 bccfg에 의해 정해지는 bbfg상의 최적의 위
        치에 레지스터 저장 명령을 생성 ;
    return tailored_bbfg ;
}
```

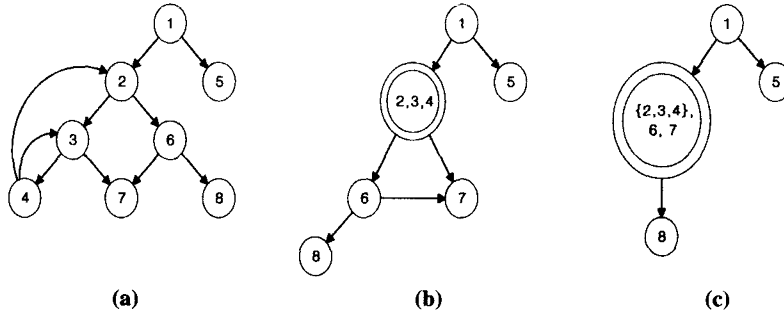
(그림 4) 프롤로그 최적화 알고리즘의 전반적인 단계
(Fig. 4) Basic steps of prolog tailoring

4.1 반복 구조의 제거

프롤로그 테일러링의 첫번째 단계는 반복 구조를 제거하는 것이다. 반복 구조는 SCC(Strongly Connected Component)〔4〕로 인식될 수 있으며 중첩 반복 구조 역시 하나의 SCC로 나타난다. SCC로 나타나는 반복 구조는 흐름 그래프 상에 하나의 노드로 치환된다. 제어흐름 그래프 상의 SCC를 찾아내기 위해 *Tarjan*의 알고리즘〔4〕을 사용할 수 있다. (그림 5)의 (a)와 (b)는 반복 구조가 제거되는 예를 보이고 있다. (a)의 노드 2, 3, 4는 하나의 SCC를 구성하고 있으며, (b)에서 나타난 바와 같이 하나의 노드로 치환된다. (a)의 노드 2, 3, 4에 도달하는 모든 간선은 (b)에서 새로이 생성된 노드에 도달하도록 구성되며 (a)의 노드 2, 3, 4에서 출발하는 모든 간선 역시 (b)에 새로이 생성된 노드에서 출발하도록 구성된다. 이와 같은 방법으로 흐름 그래프 상의 모든 반복 구조를 하나의 노드로 치환하여 얻어진 그래프를 SCC 흐름 그래프(SCC flow graph)라고 한다.

4.2 다이아몬드 구조의 제거

프롤로그 테일러링의 두번째 단계는 SCC 흐름 그래프 상의 모든 다이아몬드 구조를 제거하여 BCC 흐름 그래프(BCC flow graph)를 구성하는 것이다. 다이아몬드 구조는 그래프 상의 BCC(Bi-Connected Component)〔4〕를 찾아냄으로써 검출할 수 있다.



(그림 5) SCC 흐름 그래프와 BCC 흐름 그래프의 구성
(Fig. 5) Constructing SCC flow graph and BCC flow graph

BCC를 정의하기 위해서는 이중연결 그래프(bi-connencted graph)와 접합점(articulation point)을 정의할 필요가 있다. 접합점은 삭제될 경우, 하나의 그래프를 두 개 이상의 부그래프(subgraph)로 나누어지게 하는 그래프 상의 노드로 정의 되며, 이중연결 그래프는 접합점을 가지지 않는 그래프로 정의 된다[4]. BCC는 전체 그래프 중 이중연결 그래프의 특성을 갖는 부그래프를 뜻한다[4]. (그림 5)의 (c)는 (b)의 SCC 흐름 그래프로부터 구성한 BCC 흐름 그래프를 나타낸다. 그림에 나타나 있듯이, (b)의 노드 {2, 3, 4}, 6, 7이 형성하는 다이아몬드 구조는 BCC 흐름 그래프의 노드로 치환된다.

SCC 흐름 그래프로부터 구해진 BCC들을 그대로 연결하여 BCC 흐름 그래프를 작성할 경우 두 가지 문제가 발생한다. 첫째, BCC 흐름 그래프의 루트에 해당하는 BCC가 다수 존재할 수 있다. 기본블록 제어 흐름 그래프를 BCC 흐름 그래프로 변환하는 목적은 트리 형태의 그래프를 생성하는 것이므로, 하나의 루트 노드를 정할 필요가 있다. 둘째, BCC들 사이에는 공유된 SCC 노드들이 존재한다. 공유 노드에 대한 가공이 없었을 경우, 문제를 일으킬 수 있는 예는 (그림 5)에서 찾아 볼 수 있다.

〈표 1〉 (그림 5)의 (b)에서 검출되는 BCC
〈Table 1〉 BCC set found in (Fig. 5)(b)

BCC	1	2	3	4
SCC	1, {2, 3, 4}	1, 5	{2, 3, 4}, 6, 7	6, 8

(그림 5)의 (b)에서 찾을 수 있는 BCC는 〈표 1〉과 같다. SCC 노드 6은 BCC 3과 4 사이의 공유 노드이다. 예를 들어 SCC 노드 8에서 레지스터가 소멸된다면, 프롤로그 생성 알고리즘에 의해 만들어지는 레지스터 저장 명령어는 SCC 노드 6에 삽입된다. 이 때, 생성된 레지스터 저장 명령어는 SCC 노드 7에 의도하지 않은 영향을 끼치게 되므로 바람직한 결과를 기대하기 어렵다. 따라서, BCC 흐름 그래프를 구성하기에 앞서 BCC 집합에 대한 공유 노드 제거 과정이 이루어질 필요가 있다. 공유 노드 제거를 위한 알고리즘은 (그림 6)과 같다.

```
bccset_t   remove_shared_node(bccset_t bccset)
{
    for (bccset의 모든 bcc들에 대해)
        bcc 내부로의 간선을 가지는 지역 루트 노드 제거. ;
    for (bccset의 모든 bcc들에 대해)
    {
        공유노드가 존재한다면, 위상순서상 선행하는 bcc의 공
        유노드를 제거. ;
        모든 노드가 삭제된 bcc가 존재한다면, 이를 bccset에서
        제거. ;
    }
    if (sccfg의 루트노드가 삭제되었다면)
        sccfg의 루트노드로부터 bcc생성하여 bccset에 추가. ;
    return bccset ;
}
```

(그림 6) BCC 집합의 공유 노드 제거 알고리즘
(Fig. 6) Algorithm for shared node removal in a given BCC set

공유 노드를 제거한 BCC 집합으로부터 BCC 흐름 그래프를 생성하면 (그림 5)의 (c)와 같은 결과가 얻어진다. BCC를 검출하기 위한 알고리즘은 *Tarjan*에 의해 제시되었으므로[4][5], 본 논문에서는 SCC 흐름 그래프 상의 모든 BCC를 검출하기 위해 *Tarjan*의 알고리즘을 사용했다. SCC 흐름 그래프를 트리 형태의 BCC 흐름 그래프로 변환하기 위한 전체 알고리즘은 (그림 7)과 같다.

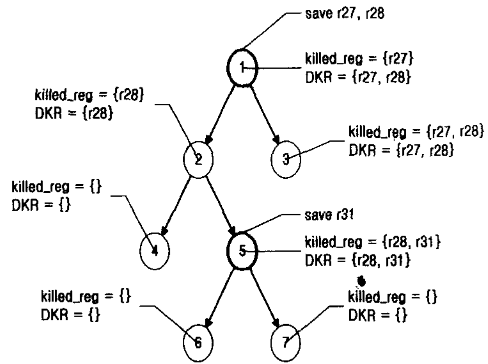
```
bccfg_t  scc_to_bcc(sccfg_t  sccfg)
{
    bccsetsccfg 내부의 bcc들을 검출. ;
    bccsetremove_shared_node(bccset) ;
    bccfg기억 장소 할당. ;
    bccset을 구성하는 bcc들로부터 bccfg의 노드 생성. ;
    sccfg의 간선들로부터 bccfg의 간선을 생성. ;
    return  bccfg ;
}
```

(그림 7) BCC 흐름 그래프 생성 알고리즘
(Fig. 7) Algorithm for constructing a BCC flow graph from a given SCC flow graph

두 BCC가 각각 포함하는 SCC들 사이에 간선이 존재한다면, 두 BCC사이에는 자식 노드와 부모 노드의 관계가 성립한다. 각각의 BCC는 하나 이상의 부모를 가질 수 없으므로, 다이아몬드 구조 제거로 얻어지는 BCC 흐름 그래프는 트리와 같은 모습을 하게 된다. 만약, 어떤 BCC가 하나 이상의 부모노드를 갖는다면, 이 BCC는 다이아몬드 구조를 제거하는 과정에서 부모 노드들을 포함하는 하나의 BCC로 통합되어 졌을 것이다.

4.3 DKR 계산

다음 단계는 각각의 BCC에 대해 소멸 레지스터 집합을 계산하고, 계산된 소멸 레지스터 집합에 기초하여 각 BCC의 DKR(Definitely Killed Registers)을 구하는 것이다. 어떤 BCC의 DKR이라 함은 그 BCC로부터 출발하는 경로에서 반드시 소멸되는 레지스터를 뜻한다. (그림 8)은 각각의 BCC에 대해 소멸 레지스터와 DKR을 계산한 예를 보이고 있다. R27은 노드 1에서 소멸되고, r28은 노드 1의 어느 자식으로 경로가 결정되든 상관 없이 소멸되므로, 노드 1의 DKR은 r27과 r28임을 알 수 있다. 반면, r31의 경우 노드 1-2-5-7을 거치는 경로에서만 소멸되므로 노드 1의 DKR에 포함되지 않는다. 같은 방법으로, 노드 2의 DKR은 r28임을 알 수 있으며, 다른 노드들의 DKR역시 마찬가지로 구해진다. 일단, 각 BCC에서의 소멸 레지스터 집합이 구해지면, BCC n의 DKR은 다음과 같이 재귀적으로 정의되어질 수 있다.

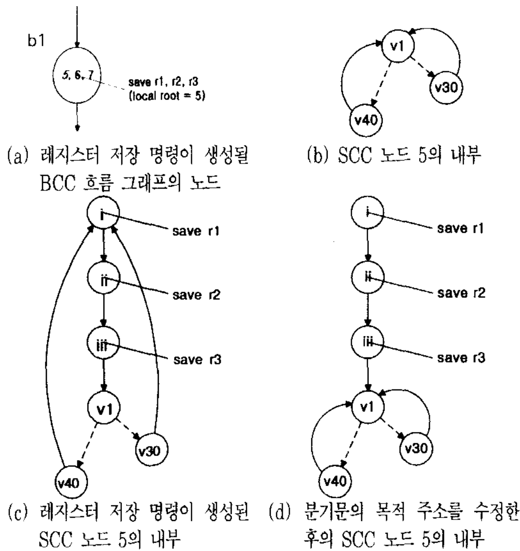


(그림 8) DKR의 계산과 레지스터 저장 명령의 생성
(Fig. 8) Computing DKR of each node and generating register save instructions

4.4 프롤로그의 생성

BCC 흐름 그래프의 루트 BCC로부터 위상 순서에 따라 프롤로그 코드를 생성한다. BCC 흐름 그래프가 트리 형태를 취하므로, 루트 BCC에서 BCC n에 도달하기 위한 경로는 오직 하나만이 존재하는데, BCC n의 프롤로그는 DKR(n) 중 이 경로에서 이미 저장이 끝난 레지스터를 제외한 나머지 레지스터에 대해 레지스터 저장 명령어를 실행한다. BCC에 생성되는 프롤로그 코드는 BCC의 지역 루트 노드에 해당하는 기본 블록의 선두에 삽입된다. DKR에 의해 레지스터 저장 명령이 생성되는 과정은 (그림 8)에 함께 나타나 있다. 루트 BCC의 DKR이 r27, r28이며, 이들에 대해서는 아직 저장 명령이 생성되어 있지 않으므로, 루트 BCC에서는 r27과 r28이 저장되어야만 한다. BCC 2와 3은 BCC 1에서 이미 r27과 r28에 대한 저장이 이루어졌으므로, 레지스터 저장 명령을 생성할 필요가 없다. BCC 4, 6, 7은 DKR이 없으므로, 역시, 레지스터 저장 명령을 생성할 필요가 없는 노드이다. BCC 5는 DKR 중 r28이 이미 저장 명령의 생성을 마쳤으므로, r31을 저장하는 명령어만을 생성한다. BCC의 지역 루트가 둘 이상인 경우, 모든 지역 루트에 동일한 프롤로

그 코드를 각각 생성한다.



(a) 레지스터 저장 명령이 생성될 BCC 흐름 그래프의 노드

(b) SCC 노드 5의 내부

(c) 레지스터 저장 명령이 생성된 SCC 노드 5의 내부

(d) 분기문의 목적 주소를 수정한 후의 SCC 노드 5의 내부

(그림 9) SCC 노드 내부에 레지스터 저장 명령이 생성되는 과정
(Fig. 9) Generating register save instructions inside a SCC node

```
insert_regsave_code(node_t *n, regset_t tosave)
{
    n의 첫머리에 tosave에 대한 레지스터 저장 명령을 생성. ;
    if( n이 SCC 노드이면 ) {
        old_start  삽입한 저장 명령과 이전의 선두 명령어의 사이
        에 레이블 생성. ;
        for( n에 포함된 명령어 중, old_start이후의 모든 명령어
        들에 대해 )
            if( SCC 노드의 예전 선두 명령어로 분기한다면 )
                old_start로 분기하도록 수정. ;
    }
}

insert_prolog(bfgnode_t *n)
{
    if(DKR(n) 중 레지스터 저장 명령을 생성하지 않은 레지스터가
    존재한다면)
    {
        v  DKR(n) 중 아직 저장되지 않은 레지스터. ;
        for( n의 모든 지역 루트 노드 k에 대해 )
            insert_regsave_code(k, v) ;
    }
    for( n의 모든 자식 노드 j에 대해 )
        insert_prolog(j) ;
}
```

(그림 10) 레지스터 저장 명령 생성 알고리즘
(Fig. 10) Algorithm for generating register save instruc-tions

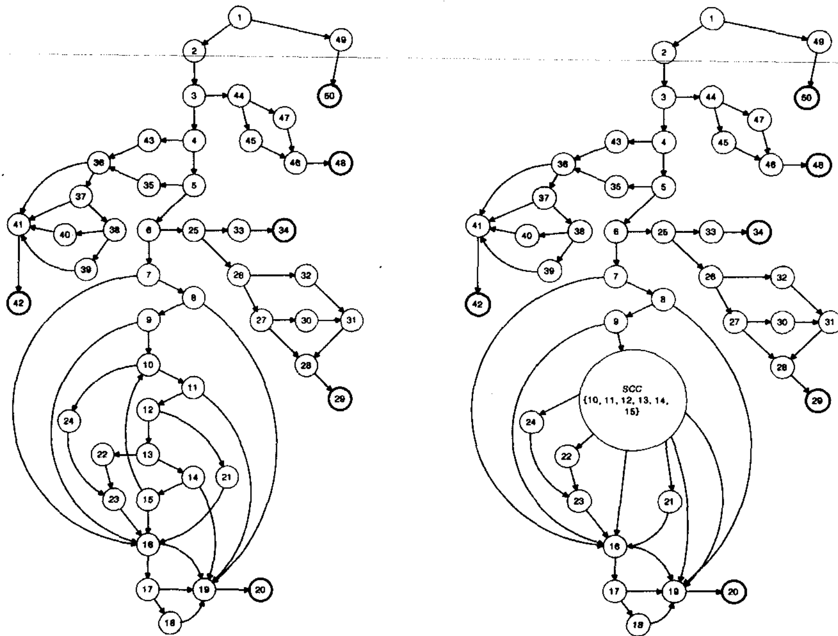
레지스터 저장 명령이 삽입되는 노드가 SCC 구조를 취할 경우, (그림 9)와 같은 처리를 필요로 한다. (그림 9)의 (a)는 지역 루트 노드가 SCC 노드인 BCC를 포함하는 BCC 흐름 그래프의 일부를 나타내고 있다. 알고리즘에 의해 프롤로그를 생성할 위치가 b1으로 결정되었을 경우, b1의 지역 루트 노드인 노드 5의 선두 블록에 레지스터 저장 명령이 삽입된다. (b)에서 나타내는 바와 같이, 노드 5의 선두 블록인 v1은 루프의 진입점 이므로, 결국, v1에 삽입된 레지스터 저장 명령은 반복 구문의 내부에 삽입되는 결과를 가져오게 된다. 레지스터 저장 명령이 삽입된 이후의 상태는 (c)과 같다. 레지스터 저장 명령이 삽입되기 전에 v1이 차지하던 위치에 r1, r2, r3를 저장하는 명령이 삽입되므로, v40과 v30에 의해 분기하는 목적지가 수정되어야만 한다. SCC 노드 5의 내부 명령어들에 대해 분기 목적 주소를 수정한 결과는 (d)와 같다. BCC 흐름 그래프와 DKR에 의해 레지스터 저장 명령을 생성하는 알고리즘은 (그림 10)과 같다.

5. 프롤로그 테일러링 적용 실례

프롤로그 테일러링 알고리즘이 적용되는 실례를 보이기 위해, *xlisp* 2.1의 소스 코드 중, *xlcont.c*의 placeform 프로시저를 선택했다. *placeform* 프로시저의 기본 블록 제어 흐름 그래프는 (그림 11)의 (a)와 같다. IBM RS/6000 XL C 컴파일러를 이용해 얻은 어셈블리 코드로부터 구성된 그래프이며, 에필로그 테일러링이 적용된 직후의 상태를 나타낸다. 노드 1이 프로시저의 진입점이며, 노드 20, 29, 34, 42, 48, 50은 에필로그 테일러링에 의해 분화된 프로시저의 탈출점이다.

5.1 반복 구조의 제거 및 SCC 흐름 그래프의 생성

제어 흐름 그래프 상의 반복 구조를 제거하기 위해, (a) 프로시저에 *Tarjan*의 SCC 검출 알고리즘[5]을 적용한다. 노드 10, 11, 12, 13, 14, 15가 SCC를 구성하므로, 이 단계에서 SCC 노드로 치환되며, SCC 노드의 지역 루트 노드는 노드 10으로 정해진다. 노드 9로부터 노드 10으로 향하는 간선은 노드 10, 11, 12, 13, 14, 15로 구성된 SCC를 향하도록 재구성된다. 노드 10, 11, 12, 13, 14 등에서 SCC 외부로 출발하는 간선들은 이들이 속하는 SCC에서 출발하는 간선으로 재구성된다. SCC 검출 및 치환, 간선 연결을 통해 얻



(a) 기본 블록 제어 흐름 그래프　　　(b) SCC 흐름 그래프

(그림 11) placeform 프로시저의 제어 흐름 그래프
(Fig. 11) Control flow graph of placeform

어지는 SCC 흐름 그래프는 (그림 11)의 (b)와 같다.

5.2 다이아몬드 구조의 제거 및 BCC 흐름 그래프의 생성

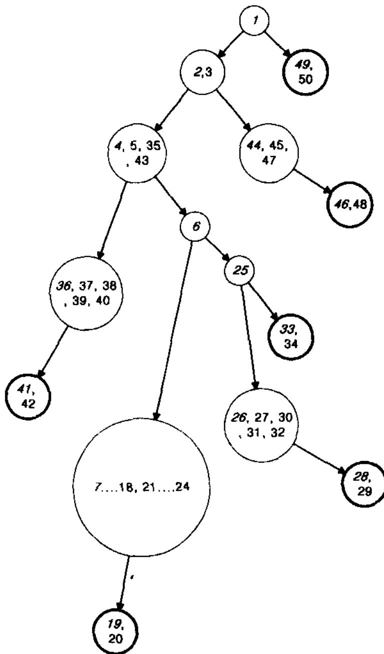
다음은 (그림 11)의 (b)에서 다이아몬드 구조를 제거하기 위한 단계이다. *Tarjan*의 BCC 검출 알고리즘 [5]을 적용하여 BCC 집합을 구성하고, 둘 이상의 BCC에 중복되어 나타나는 공유 노드를 제거한다. <표 2>는 공유 노드 제거 과정이 진행됨에 따라, BCC 집합이 변화하는 모습을 나타낸다. 첫 열에 표시된 b*n*은 검출된 각 BCC 노드를 의미하며, 둘째 열은 SCC 흐름 그래프 상의 노드 중 해당 BCC에 포함되는 노드들을 나타낸다. 강조체로 표기된 노드는 각 BCC의 지역 루트 노드에 해당한다.

검출된 BCC 집합에 대해 공유 노드를 제거하기 위해, 먼저, 자신이 속한 BCC의 외부를 향하는 간선을 갖는 모든 지역 루트 노드를 BCC에서 삭제한다. 노드 1은 b2와 b3의 지역 루트 노드이고, b2에서 b3로, b3에서 b2로의 간선을 가지므로, 두 BCC에서 모두 삭제된다. b2, b3, b6, b7, b13, b14, b15, b16, b18 등의 지역 루트 노드 역시, 자신이 속한 BCC의 외부를 향하는 간선을 가지므로, 제거된다. 이 단계를 마친 이후에 남아있는 공유 노드에 대해서는, 공유 노드를 갖는 두 BCC 중 위상 순서가 선행하는 BCC에 속한 노드를 삭제한다. 이 단계에서, b2의 2, b3의 49, b6의 4, b7의 44, b8의 46, b10의 36, b11의 41, b14의 7, b16의 33, b18의 26, b19의 28, b21의 19 등의 노드가 삭제된다. 모든 노드가 삭제된 b2, b3, b6, b7, b14, b16, b18 등의 BCC는 BCC 집합에서 제외된다. 마지막으로, SCC 흐름 그래프 상의 루트 노드인 노드 1이 BCC 집합에 존재하지 않으므로, 노드 1을 포함하는 BCC를 생성하여 BCC 흐름 그래프의 루트 노드로 삼는다. 공유 노드 제거 및 루트 노드 생성을 마친 BCC 집합은 <표 2>의 세째 열과 같다.

공유 노드 제거 과정을 거쳐 얻어진 BCC 집합과, SCC 흐름 그래프의 간선들의 관계로 생성되는 BCC 흐름 그래프는 (그림 12)와 같다.

〈표 2〉 공유 노드 제거에 의한 BCC 집합의 변화
〈Table 2〉 Differences between before and after processing shared node removal

BCC #	공유 노드 제거 이전	공유 노드 제거 이후
b1	-	1
b2	1, 2	-
b3	1, 49	-
b4	49, 50	49, 50
b5	2, 3	2, 3
b6	3, 4	-
b7	3, 44	-
b8	44, 45, 46, 47	44, 45, 47
b9	46, 48	46, 48
b10	4, 5, 43, 35, 36	4, 5, 35, 43
b11	36, 37, 38, 39, 40, 41	36, 37, 38, 39, 40
b12	41, 42	41, 42
b13	5, 6	6
b14	6, 7	-
b15	6, 25	25
b16	25, 33	-
b17	33, 34	33, 34
b18	25, 26	-
b19	26, 27, 28, 30, 31, 32	26, 27, 30, 31, 32
b20	28, 29	28, 29
b21	7…19, 21…24	7…18, 21…24
b22	19, 20	19, 20



(그림 12) BCC 흐름 그래프
(Fig. 12) A BBC flow graph of *placefrom*

5.3 DKR의 계산 및 레지스터 저장 명령어 생성

(그림 12)와 같은 BCC 흐름 그래프가 구해지면, 레지스터 저장 명령어를 생성할 위치를 결정하기 위해, BCC 흐름 그래프의 모든 노드에 대해 DKR을 계산한다. BCC 흐름 그래프 상의 한 노드에서의 DKR은 그 정의에 의해, 자식 노드들의 DKR에 공통 포함된 레지스터들과 자신에게서 소멸되는 레지스터들의 집합으로 정의된다. DKR이 구해지면, BCC 흐름 그래프의 각 노드에 레지스터 저장 명령을 생성한다. 레지스터 저장 명령어의 생성은 BCC 흐름 그래프의 루트 노드로부터 시작하여 자식 노드로 작업 노드를 이동하며 진행된다. 작업 노드에서는 노드의 DKR에 대한 저장 명령을 생성하는데, DKR 중, 루트 노드로부터의 경로에서 이미 저장 명령을 생성한 레지스터가 있다면, 이 레지스터에 대해서는 저장 명령을 생성하지 않는다. (그림 12)의 각 노드에서의 소멸 레지스터 집합, DKR, 그리고, 레지스터 저장 명령을 생성해야 할 레지스터들은 〈표 3〉과 같다. 레지스터 저장 명령어가 삽입되는 블록이 SCC를 형성하는지 여부를 조사하면서 작업을 진행해야 하지만, (그림 12)의 어떠한 노드도 지역 루트 노드가 SCC가 아니므로, 이 예제에서는 SCC 노드에 대한 처리 과정이 적용되지 않는다.

〈표 3〉 (그림 12)의 소멸 레지스터 및 DKR 계산
〈Table 3〉 Killed registers and DKR of (Fig. 12)

BCC #	소멸 레지스터 집합	DKR	저장할 레지스터
b1	r28, r29, r31, lr	r28, r29, r31, lr	r28, r29, r31, lr
b4	r30, r31, lr	r30, r31, lr	r30
b5	r31	r31, lr	-
b8	r30, r31, lr	r30, r31, lr	r30
b9	-	-	-
b10	-	lr	-
b11	r30, lr	r30, lr	r30
b12	-	-	-
b13	-	lr	-
b15	-	lr	-
b17	lr	lr	-
b19	r27, r28, r29, r30, r31, cr4, lr	r27, r28, r29, r30, r31, cr4, lr	r27, r30, cr4
b20	-	-	-
b21	lr	lr	-
b22	-	-	-

6. 실험 및 결과

테일러링에 의해 얻어지는 성능 향상을 측정하기 위해, 두 가지 방법에 의한 실험을 실시했다. 첫 번째 실험에서는 테일러링 유형에 따른 실행 속도 향상을 측정하였다. 실험에 사용한 프로그램은 1과 40 사이의 정수에 대해 피보나치 수열을 구하는 작업을 수행한다. C언어로 작성한 프로그램에 대해 SGI IRIX 6.2 cc가 생성한 MIPS R5000 어셈블리 프로그램을 테일러링 한 다음, 목적 코드를 생성하고, 링크하는 방법으로 실행 파일을 생성했다. 테일러링 하기 전과 에필로그 및 프롤로그 테일러링을 적용한 후의 어셈블리 프로그램은 (그림 13)과 같다. 실험 결과, 테일러링 하지 않은 경우, 실행에 14255μsec가 소요되었으며 에필로그 테일러링, 프롤로그 테일러링을 적용한 경우의 실행 시간은 각각 13047μsec, 12564μsec이 소요되었다.

〈표 4〉 피보나치 수열 계산 문제의 테일러링 유형에 따른 상대 성능 평가
〈Table 4〉 Relative performance evaluation between tailoring techniques

최적화 유형	없음	에필로그	프롤로그
없음	1	1.0926	1.1346
에필로그	-	1	1.0384
프롤로그	-	-	1

〈표 4〉는 테일러링 유형에 따른 상대 성능 평가를 나타낸다. 프로그램 A, B가 있고, A의 실행 시간이 T_a, B의 실행 시간이 T_b라면, 프로그램 A에 대한 프로

```
        .ent        fib 2
fib:    .option     O0
        .set        noreorder
        .cpload     $25
        .set        reorder
        subu        $sp, 40
        sw          $31, 28($sp)
        .cprestore  24
        sw          $4, 40($sp)
        sw          $16, 20($sp)
        .mask       0x90010000, -12
        .frame      $sp, 40, $31
        .loc        2 4
        .loc        2 5
        lw          $14, 40($sp)
        bgt         $14, 1, $32
        .loc        2 6
        li          $2, 1
        b           $33
$32:    .loc        2 7
        lw          $4, 40($sp)
        addu        $4, $4, -1
        .livereg
        0x0800000E,0x00000000
        jal         fib
        move        $16, $2
        lw          $4, 40($sp)
        addu        $4, $4, -2
        .livereg
        0x0800000E,0x00000000
        jal         fib
        addu        $2, $2, $16
        b           $33
        .loc        2 8
        b           $33
$33:    .livereg
0x2000FF0E,0x00000FFF
        lw          $16, 20($sp)
        lw          $31, 28($sp)
        addu        $sp, 40
        j           $31
        .end        fib
```

(a) 테일러링 되지 않은 상태

```
        .ent        fib 2
fib:    .option     O0
        .set        noreorder
        .cpload     $25
        .set        reorder
        subu        $sp, 40
        .cprestore  24
        sw          $4, 40($sp)
        .mask       0x90010000, -12
        .frame      $sp, 40, $31
        .loc        2 8
        .loc        2 9
        lw          $14, 40($sp)
        bgt         $14, 1, $32
        .loc        2 10
        li          $2, 1
addu    $sp, 40
        j     $31
$32:    .loc        2 11
        sw     $31, 28($sp)
        lw          $4, 40($sp)
        addu        $4, $4, -1
        .livereg
        0x0800000E,0x00000000
        jal         fib
        sw     $16, 20($sp)
        move        $16, $2
        lw          $4, 40($sp)
        addu        $4, $4, -2
        .livereg
        0x0800000E,0x00000000
        jal         fib
        addu        $2, $2, $16
        livereg
0x2000FF0E,0x00000FFF
        lw     $16, 20($sp)
        lw     $31, 28($sp)
        addu   $sp, 40
        j      $31
        .loc        2 12
        .end        fib
```

(b) 에필로그 및 프롤로그 테일러링을 마친 상태

(그림 13) 실험에 사용된 피보나치 수열 문제의 어셈블리 코드
(Fig. 13) A portion of MIPS R5000 assembly code Fibonacci series calculation

그램 B의 성능 향상 P_{ba}는 다음과 같다.

$$P_{ba} = \frac{T_A}{T_b}$$

두 번째 실험은 프롤로그 테일러링에 의한 레지스터 저장 명령어의 감소율을 보이기 위한 것으로, xlisp 2.1에서 임의 추출한 프로시저 8개에 대해, 각각 테일러링을 수행하고, 프로시저가 실행하는 레지스터 저장 명령어 수의 변동을 측정하였다. 프롤로그 테일러된 프로시저는 어느 경로를 통하여 실행되느냐에 따라 실제 실행되는 레지스터 저장 명령의 수가 달라진다. 따라서, 이 경우, 레지스터 저장 명령어의 수는 프로시저가 실행될 수 있는 모든 경로에 대한 선택의 확률이 동일하다는 가정하에 평균 실행되는 명령어 수를 구하였다. SCC 흐름 그래프에서의 가능한 실행 경로의 수를 PT, 탈출점의 개수를 NE, 탈출점 i까지의 경로에서 실행되는 레지스터 저장 명령어의 수를 NS_i, 탈출점 i에 도달하는 가능한 경로의 수를 PE_i이라고 하면, 프로시저의 실행 시간에 실행되는 레지스터 저장 명령어의 평균 개수 AS는 다음과 같이 구해질 수 있다.

$$AS = \sum_{i=1}^{NE} \frac{PE_i \cdot NS_i}{PT}$$

실험 결과, 평균 17.47%의 레지스터 저장 명령어 감소율을 보였으며, 가장 많이 감소된 경우와 가장 적게 감소된 경우를 제외한 평균 감소율은 12.82%를 나타내었다. 결과는 〈표 5〉와 같다.

〈표 5〉 프롤로그 테일러된 프로시저에서 실행되는 레지스터 저장 명령어 수의 변동

〈Table 5〉 The decreased number of register save instructions by prolog tailoring

프로시저	최적화 전	최적화 후	차	감소율(%)
placeform	7	4.51	2.49	35.57
mark	8	5.50	2.50	31.25
sweep	9	6.50	2.50	27.78
xlpatprop	5	4.00	1.00	20.00
evlist	9	7.50	1.50	16.67
xlenter	6	5.79	0.21	3.50
evalh	9	8.70	0.30	3.33
cons	9	8.85	0.15	1.67
평균				17.47
극한 제외 평균				12.82

7. 결론 및 향후 연구 과제

본 논문을 통하여, 프로시저의 실행 속도를 향상시키기 위한 방안으로, 프로시저 실행 경로상의 레지스터 저장 명령어의 개수를 줄일 수 있는 기법을 제안하였으며, 실험에 의해 실제 실행되는 레지스터 저장 명령어가 평균 12.82% 감소됨을 보였다. 이는 프로시저의 실행 속도를 종전보다 향상시킬 수 있으며, 피호출자 저장 레지스터가 많고, 프로시저 호출이 빈번하게 일어나는 환경에서는 더욱 효과를 나타낼 것으로 기대된다. 실험에 사용한 피보나치 수열을 구하는 프로그램은 8억회 이상의 프로시저 호출이 일어나므로, 프롤로그 테일러링의 효과를 잘 나타낼 수 있는 환경이라 할 수 있다. 실험 결과, 프롤로그 테일러링은 에필로그 테일러링만을 수행한 프로시저에 대해 3.84%의 속도 향상을 보였다.

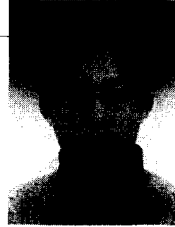
본 논문에서 제안한 프롤로그 테일러링 알고리즘은 에필로그의 테일러링이 이루어져 있는 프로시저에 적용될 수 있다. 에필로그 테일러링은, 저장된 모든 레지스터를 항상 복원하지는 않으므로, 레지스터 저장 및 복원을 위해 *push*, *pop* 명령어를 사용할 수 없다는 단점을 가지고 있으나, 프롤로그 테일러링과 함께 사용된다면, 이러한 제약이 극복된다.

프롤로그 테일러링 알고리즘은 다이아몬드 구조와 반복 구조 내부에 레지스터 저장 명령을 생성하지 않기 위해, BCC 흐름 그래프 및 SCC 흐름 그래프를 구성한다. 하지만, 이러한 준비만으로는 반복 구조 내부에 레지스터 저장 명령어를 생성하는 것을 완전히 막을 수는 없었다. 이 때문에, 레지스터 저장 명령어를 생성할 단계에서 SCC 노드 내부에 삽입되는 저장 명령어에 대한 처리가 필요하다. 구조적으로 좀 더 개선된 알고리즘이 존재한다면, 테일러링의 비용을 줄일 수 있으리라 예상된다. 또한, BCC 검출 과정과 공유 노드 제거 과정을 통합하는 알고리즘이 제안된다면, 테일러링에 드는 비용을 더욱 줄일 수 있을 것이다.

참 고 문 헌

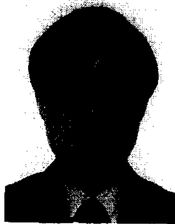
[1] K. Ebcioglu, R. D. Groves, K. C. Kim, G. M. Silberman, and I. Ziv, "VLIW Compilation Techniques in a Superscalar Environment,"

ACM SIGPLAN Not., Vol.29, No.6, pp.36-48, 1994
[2] K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced Compiler Technology for the RISC System/6000 Architecture," IBM RISC System/6000 Technology, SA23-2619, IBM Corporation, 1990
[3] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers, Addison Wesley, 1988
[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974
[5] R. E. Tarjan, "Depth first search and linear graph algorithms," SIAM J. Computing, Vol.1, No.2, pp.146-160, 1972
[6] F. E. Allen, and J. Cocke, "A Catalogue of Optimizing Transformations," In Design and Optimization of Compilers, Prentice-Hall, 1972
[7] K. Ebcioglu, F. Allen, K. Zadeck, and B. BkRosen, A Book on Compiler Optimizations, pp.3-21, North Holland, 1988
[8] R. Sedgewick, Algorithms in C, Addison Wesley, 1990
[9] S. Graham, and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," ACM 23, 1, pp.172-202, 1976
[10] E. Farquhar, and P. Bunce , The MIPS Programmer's Handbook, Morgan Kaufmann, 1993.
[11] J. Heinrich, R4000 Microprocessor User's Manual, MIPS Technologies, Inc., 1996



지 윤 찬

1993년 인하대학교 전자계산공학과 (학사)
1993년~현재 인하대학교 대학원 전자계산공학과 석사과정
관심분야 : 분산 HTTP 프락시 서버, 병렬처리(특히, 네트워크 상의 분산 병렬처리 시스템)



김 기 창

1986년 California State Polytechnique University, Pomona (학사)
1992년 University of California, Irvine (공학 석사, 박사)
1992년~94년 IBM T.J. Watson 연구소 연구원
1994년~현재 인하대학교 전자계산공학과 조교수
관심분야 : 병렬화 컴파일러