ILP 프로세서를 위한 개선된 레지스터 할당 기법
(An Improved Register Allocation Technique for ILP Processors)

신 화 정 † 이 기 호 ††
(Hwajung Shin) (Kiho Lee)

요 약 고성능 마이크로 프로세서들은 성능 향상을 위해 ILP를 지원한다. 병렬성을 극대화시키기 위해서는 많은 성능 저해 요인들을 제거해야 한다. 최근에는 컴파일러의 역할을 증대시켜 이러한 요인들을 줄이기 위한 노력들이 활발히 진행되고 있다.

본 논문에서는 성능 저해 요인인 조건 분기 처리를 위하여 조건 실행과 레지스터 할당을 결합함으로써 메모리로의 대피를 최소화하고 병렬성을 향상시킬 수 있는 개선된 레지스터 할당 알고리즘을 제안한다. 제안한 방법을 적용하여 실험한 결과 간섭 그래프의 에지수가 4.47% 감소되었고 그 결과 요구되는 대피 변수의 수도 21.35% 감소되었다. 그리고 기존의 방법에 비해 19.38%의 성능 향상 결과를 얻었다. 결국 본 레지스터 할당 기법은 조건 실행을 통해 조건 분기 명령을 제거하여 기본 블록 내의 명령어 수를 증가시켜 병렬처리의 기회를 증진시키고 조건 분석을 통해 간섭 그래프의 불필요한 에지를 제거시켜 보다 효율적인 레지스터 할당을 실현함으로써 제안한 방법의 타당성을 검증하였다.

Abstract High performance microprocessors support ILP for improving the performances. In order to maximize the parallelism, it is necessary to remove the factors that constrain the improvement in performances. Recently, there have been active researches to reduce these hindrance factors by utilizing the role of compiler.

This paper proposes an improved register allocation algorithm that minimizes the number of spilled codes and maximizes the parallelism. The proposed technique combines the predicated execution and register allocation in order to deal with conditional branches, a hindrance factor of improving the performance. The result of experiments with the proposed algorithm shows the reduction of 4.47% in the number of edges of the interference graph, and the reduction of 21.35% in the number of spilled variables. The overall performance improves by 19.38%, comparing to the conventional algorithm. In the proposed algorithm, the predicated execution removes the conditional branches, the increases in the number of instructions inside a basic block gives the higher chance of parallel execution, and predicate analysis removes the unnecessary edges in the interference graph. The experiment results from the proposed algorithm verifies the effectiveness of our improved algorithm.

1. 서 론

프로세서의 성능은 크게 하드웨어와 소프트웨어의 병렬성 처리 기법에 의해 좌우된다. 하드웨어에 의한 클럭 싸이클 시간은 개발 당시에 정해지는 것으로 시간을 단축시키기가 어렵지만 소프트웨어에 의한 컴파일 기법을 효과적으로 활용하여 최적화를 이룸으로써 실행 시간을 단축시키는 것이 가능하다. 실행 시간을 줄이기 위해 명령들을 병렬로 처리하는 마이크로 프로세서의 성능은 컴파일러 기술에 많은 영향을 받는다. 최근 마이크로 프로세서는 10년 전 프로세서보다 100배 이상 빠른 연산 속도를 가지게 되었는데 이런 혁신적인 성능 향상 기술 중의 하나가 명령어 수준 병렬 처리(ILP : Instruction Level Parallelism)이다[1].

최근 고성능 마이크로 프로세서들은 성능 향상을 위해 ILP를 지원한다. 즉 주어진 순차 코드에서 상호 의존성이 없는 독립적인 명령어를 찾아내어 병렬로 수행하여 실행 결과에는 변화를 주지 않으면서 실행 시간을

† 종신회원 : 한양여자대학 전산정보처리과 교수
hjshin@hywoman.ac.kr
†† 종신회원 : 이화여자대학교 컴퓨터학과 교수
khlee@mm.ewha.ac.kr
논문접수 : 2000년 5월 12일
심사완료 2000년 11월 18일

단축시킨다. 대표적인 프로세서에는 순차 코드에서 병렬 수행 가능한 독립적인 명령을 찾아내는 스케쥴링을 수행 시간에 하드웨어가 담당하는 수퍼스칼라(super-scalar)와 컴파일 시간에 컴파일러가 담당하는 VLIW (Very Long Instruction Word) 그리고 VLIW를 발전시킨 EPIC(Explicitly Parallel Instruction Computing) 등이 있다. EPIC는 64비트 명령어 세트 아키텍처 디자인 기술로 분기 예측, 예측 실행, 명시적 병렬 처리 기술들을 독특한 방식으로 결합하여 마이크로 프로세서의 대폭적인 성능 향상을 실현한다[2].

순차 처리만이 가능했던 프로세서에서는 성능 향상을 위해 명령 수를 줄이는 것이 최대 목표였지만 병렬 처리가 가능한 프로세서에서는 시스템에서 제공하는 병렬성을 높일 수 있도록 하는 것이 더욱 중요하다. 프로세서의 성능을 저하시키는 대표적 요인 중 하나는 분기 명령이다. 하나의 프로그램에서 제어 흐름을 변화시키는 조건 분기는 분기 명령의 실행과 타겟 명령어의 인출간에 종속 관계를 일으키기 때문에 기본 블록 내에 명령 수가 감소하여 병렬 처리 효율이 저하된다. 이것을 완화하는 방법으로 조건 실행(predicated execution)을 이용할 수 있다[3]. 이렇게 하면 분기 명령을 제거할 수 있고 분기 방향에 관계없이 IF 변환된 부분을 따라 빠르게 순차적인 명령어 인출이 가능하고 분기 명령으로 인한 파이프라인의 부작용을 제거할 수 있다.

성능 향상을 위해 컴파일러가 해야할 일 중에는 명령어 수준 병렬성을 보다 많이 추출하도록 하는 것과 레지스터를 효율적으로 사용하는 것이 중요하다. 실행 변수의 값이 메모리에 존재하는 경우와 레지스터에 존재하는 경우 이를 접근하는데 걸리는 시간이 차이가 크다. 그러므로 실행 효율을 높이려면 빠른 접근이 가능한 레지스터 할당이 효율적으로 이루어져야만 한다. 본 논문에서는 ILP 프로세서의 성능 향상을 저해하는 중요 요인 중에 하나인 조건 분기를 처리하는 조건 실행과 프로그램 처리 속도를 향상시킬 수 있는 레지스터 할당을 결합한 조건 실행 지원 레지스터 할당 알고리즘을 제안한다. 벤치마크 프로그램으로 제안한 레지스터 할당을 실시하여 기존의 방법을 적용했을 경우와 대피 변수 수와 실행 클럭 수를 측정하여 타당성을 검토한다.

본 논문의 전체적인 구성은 다음과 같다. 2장에서는 조건 실행에 대해 설명하고 3장에서 조건 실행 지원 레지스터 할당 알고리즘을 제안한다. 4장에서는 제안한 알고리즘의 타당성을 검증하기 위한 실험 방법과 성능 측정 결과를 보이고 5장에서 결론 및 향후 연구 과제에 대해 논한다.

2. 조건실행

프로세서의 성능을 저해하는 요인으로는 자료 종속 관계, 제어 종속 관계, 프로세서의 구조적인 문제 등을 들 수 있다. 제어 종속 관계는 분기 명령에 의해 주로 발생하는데 이들 분기 명령의 문제점은 프로그램에서 제어 종속을 일으켜 기본 블록 내에 명령어 수를 감소시킬 뿐만 아니라 조건 분기 명령의 경우 조건 판정이 지연되기 때문에 지속적인 파이프라인 처리를 방해하여 실행 시간의 증가를 가져온다. 조건 분기문은 SPEC 벤치마크 프로그램에서 두번째로 많이 사용되는 명령으로 20%이상을 차지한다[1][4]. 이런 분기 명령을 처리하는 방법에는 분기 예측(branch prediction)과 조건 실행이 있다[5]. 분기 예측은 미리 분기를 예측하여 지속적인 파이프라인 처리를 보장하는 방법으로 어느 한 쪽의 경로가 우세할 경우에는 효과적이지만 여러 경로가 유사한 분기율을 가질 경우는 예측이 어려워져 잘못된 예측에 대한 처리 부담이 증가하게 되므로 비효율적이다.

조건 실행은 조건 분기를 취급하기 위한 효과적인 방법으로 보호 실행(guarded execution)이라고도 한다[5]. 이 기술은 몇 개의 경로를 통합하여 하나의 블록을 만들고 이 경로에 연계된 분기 명령들을 제거하는 것이다. 조건 실행은 조건 기술자(predicated specifier)라고 하는 원시 오퍼랜드의 부울 값에 근거한 명령어의 조건적 실행이다. 조건 기술자의 값이 참이면 명령은 정상적으로 실행되고 그 값이 거짓이면 실행되지 않는다. 조건 실행은 컴파일러로 하여금 다중 실행 경로에서 하드웨어가 병렬성을 최대로 활용할 수 있도록 하기 위해 조건 기술자 값에 무관하게 명령어를 인출하여 효율적으로 처리하는 것이 가능하다.

조건 분기를 소거하기 위해 코드 변형 기법인 IF 변환(IF conversion)을 실시한다[3][5]. IF 변환은 비순환 제어 흐름 영역 전체를 하나의 분기없는 블록의 조건화된 코드로 변환할 수 있다. 즉 조건 분기를 적절한 조건 정의와 조건 실행 명령으로 교체하는 처리 기법을 IF 변환이라고 한다. IF 변환은 우선 코드 내에 조건 분기들에 조건 레지스터를 할당하여 조건 기술자가 설정된 비교 명령, 즉 다음 형식과 같은 조건 정의 명령으로 대치한다.

cmpop cond_reg, operand1, operand2

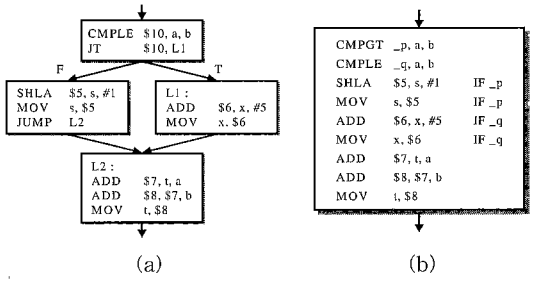
여기서 cmpop는 비교연산자로 operand1과 operand2를 비교하여 참, 거짓을 판별하고 참이면 이후 조건 레지스터인 cond_reg를 가진 명령들만 수행한다. 다음으로 분

기 후에 나오는 선택적인 경로 상의 명령들에 조건 기술자를 삽입하여 다음과 같은 형식의 조건 실행 명령으로 변환한다.

opcode operand3, operand4, operand5 IF cond_reg

일반 명령 뒤에 IF절을 추가하는 형태로 조건 레지스터의 값이 참이면 명령을 실행하고 만약 거짓이면 실행하지 않도록 한다. 이렇게 IF 변환을 실시하면 분기와 연계된 제어 종속 관계가 조건 정의 명령과 연계된 자료 종속 관계로 변환된다. 즉 조건 실행은 조건 분기를 사용하지 않고도 조건적으로 실행되는 프로그램 구조를 표현할 수 있다.

IF 변환의 예를 살펴보면 그림 1과 같다. 그림 1(a)에서 IF 변환을 위해 조건 정의 명령을 쓰고 선택 경로상에 명령을 조건 실행 명령으로 변환한다. 그림 1(b)는 IF 변환한 후의 결과이다. 여기서 _p, _q는 조건 레지스터로서 1비트로 이루어진 특수 목적 레지스터이다. IF 변환 전에 사용되었던 2개의 분기문이 제거되고, 제어 흐름 그래프에서 4개이던 기본 블록이 하나의 기본 블록으로 통합되었음을 확인할 수 있다.



(a) (b)

그림 1 (a) IF 변환 전의 제어 흐름 그래프 (b) IF 변환 후의 제어 흐름 그래프

3. 조건실행 지원 레지스터 할당

전체 컴파일 시간 중 많은 시간이 소요되는 레지스터 할당은 역사가 오래되었지만 여전히 많은 연구가 이루어지는 분야이다. 대부분의 최적화 컴파일러에서 채용하고 있는 레지스터 할당 알고리즘은 그래프 컬러링에 기초한 것으로 Chaitin에 의해 최초로 시도되었고[6] 현재 진행 중인 연구들도 이를 많이 응용하고 있다[4][7][8]. 레지스터를 할당하는 과정에서 모든 레지스터가 사용 중이라면 이미 사용중인 레지스터 중에 하나를 선택하여 그 내용을 메모리에 저장(store)하고 그 레지스터를 사용해야 한다. 후에 다시 사용하려면 적재(load) 명령이 필요하게 된다. 이때 선택되는 변수를 대피(spill) 변수라 하고 저장 명령과 적재 명령을 대피 코드라 한다. 레지스터 할당의 목적은 이 대피 코드를 최소화하는 것이다.

본 논문은 Chaitin의 레지스터 할당 알고리즘을 확장하여 ILP 프로세서를 위한 알고리즘을 개발하였다. Chaitin 이후에도 이를 개선하기 위한 연구가 계속 되었다. Chow[9]는 변수에 우선 순위를 부여하기 위해 새로운 휴리스틱을 적용하여 지역 레지스터 할당과 전역 레지스터 할당으로 나누어 레지스터 할당을 수행하고 생존 범위 분할을 통하여 대피 코드의 수를 줄였다. 하지만 우선 순위를 결정할 때 코드가 고정된 것으로 가정하였기 때문에 코드 이동 후의 레지스터 할당은 전혀 고려되지 않았다. Briggs[7]는 Chaitin의 단점을 보완하기 위해 컬러링 단계 후에 대피 결정을 한다. 이 방법은 Chaitin의 방법보다 더 효율적인 할당 결과를 얻는다. 그러나 명령어 수준의 병렬 처리와 분기 명령에 대한 고려가 없다. 이와 같이 Chaitin의 레지스터 할당 알고리즘을 확장한 기존의 알고리즘들은 대피 코드의 수는 줄여주지만 ILP에 대한 고려가 없으므로 본 논문의 대상이 되는 ILP 프로세서를 위한 레지스터 할당에는 큰 효과를 기대할 수 없다. 따라서 본 논문에서는 가장 보편적으로 많이 사용되고 있는 Chaitin의 레지스터 할당 기법을 근간으로 선택하여 명령어 수준 병렬 처리를 고려하여 특히 분기 명령을 효율적으로 처리할 수 있는 조건 실행을 지원하는 레지스터 할당 알고리즘을 제안하고자 한다. 제안하는 조건 실행 지원 레지스터 할당 알고리즘(Register Allocation algorithm Supporting Predicated Execution, RASPE)의 전체 흐름도는 그림 2와 같다.

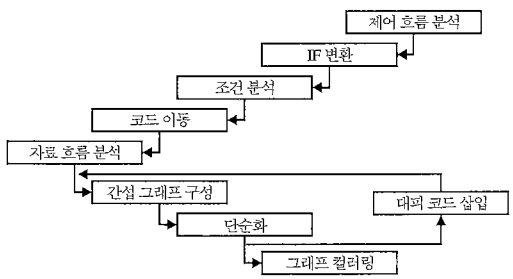


그림 2 RASPE 전체 흐름도

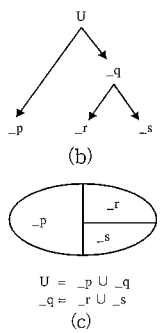
RASPE는 컴파일러가 생성한 중간 코드를 입력으로 받아 제어 흐름 분석을 통해 제거할 수 있는 조건 분기 명령을 선정하고 IF 변환을 수행하여 조건 분기 명령을

제거한다. 다음 단계로 IF 변환된 중간 코드의 조건을 표현하는 조건 기술자, 즉 조건(predicate) 간의 관계를 분석하는 조건 분석을 실시한다. 예를 들어 조건 p하에서 IF 변환으로 q와 r이라는 조건이 생성되었다면 다음과 같은 관계가 성립한다. q나 r이 참이면 p는 반드시 참이다($q \cup r \subseteq p$). q가 참이면 r은 반드시 거짓이고 그 역도 성립한다($q \cap r = \phi$). p가 참이면 q나 r 중에 하나는 참이다($p \subseteq q \cup r$). 이와 같은 관계를 p는 q와 r로 분할된다고 하고 q와 r은 서로 배타적이라고 하며, 여기서 조건이 분할된다고 하는 것은 집합에서의 포함 관계로 표현할 수 있다. 조건 분석을 위해 조건을 노드로 하고 포함 관계를 에지로 표시하는 분할 그래프를 구성한다. 이때 포함 관계가 있는 조건들은 동시에 참이 되어 함께 처리될 가능성이 있지만 포함 관계가 없는 조건들, 즉 서로간의 에지가 없는 조건들은 동시에 참이 될 가능성이 전혀 없고 그 중 어느 하나만이 참이 되어 실행된다. 분할 그래프가 구성되면 조건 관계에 대한 조사는 그래프 트래버설(traversal) 알고리즘을 이용하여 두 조건이 같은 분할의 서로 다른 에지를 통해 공통의 조상(ancestor) 노드에 도달할 수 있으면 두 조건은 서로 배타적이라고 판별한다. 이렇게 서로 배타적인 조건들을 찾아내어 각 조건하에서만 사용되는 변수들의 집합을 구하면 이들이 서로 배타적으로 실행되는 변수 집합이 된다.

조건 분석을 실시하는 과정을 예를 들어 살펴보면 그림 3과 같다. 그림 3(b)는 IF 변환된 예제 코드에 대한 분할 그래프이고, 그림 3(c)는 조건간의 포함 관계를 나타낸 것이다. 전체 집합을 U라 하고 첫 번째 조건에서 _p와 _q로 분할되고, _q를 다시 _r과 _s로 분할한다. 즉

```
CMPLT   _p,a,b
CMPGE   _q,a,b
MUL     $5,k,m      IF _p
MOV     x,$5        IF _p
CMPGE   _s,c,#0     IF _q
CMPLT   _r,c,#0     IF _q
ADD     $6,x,#5     IF _s
MOV     x,$6        IF _s
ADD     $7,m,#1     IF _s
MOV     m,$7        IF _s
ADD     $8,k,m      IF _s
MOV     y,$8        IF _s
SHLA    $9,x,#1     IF _r
MOV     x,$9        IF _r
SUB     $10,k,m     IF _r
MOV     y,$10       IF _r
ADD     $11,k,#1    IF _r
MOV     k,$11       IF _r
```

(a)



(b) (c)

```
CMPLT   _p,a,b
CMPGE   _q,a,b
MUL     $5,k,m      IF _p
CMPGE   _s,c,#0     IF _q
CMPLT   _r,c,#0     IF _q
MOV     x,$5        IF _p
ADD     $6,x,#5     IF _s
SHLA    $9,x,#1     IF _r
MOV     x,$6        IF _s
ADD     $7,m,#1     IF _s
MOV     x,$9        IF _r
SUB     $10,k,m     IF _r
MOV     m,$7        IF _s
MOV     y,$10       IF _r
ADD     $8,k,m      IF _s
ADD     $11,k,#1    IF _r
MOV     y,$8        IF _s
MOV     k,$11       IF _r
```

(d)

그림 3 (a) IF 변환 후 코드 (b) 분할 그래프 (c) 포함 관계 (d) 코드 이동 후 코드

_p, _r, _s는 서로 포함 관계가 없으므로 서로 배타적으로 실행됨을 의미한다. 조건 _p하에서만 사용되는 {$5}, _s하에서만 사용되는 {$6, $7, $8}, _r하에서만 사용되는 {$9, $10, $11}은 서로 배타적 변수 집합이다.

명령들의 병렬성을 최대한 추출하여 병렬 처리하기 위해 실행 결과를 변경시키지 않으면서 서로 독립적인 명령들을 찾아내어 코드 이동을 실시한 후 자료 흐름 분석을 통해 변수의 생존 범위를 구한다. 코드 이동은 병렬 처리를 가능하게 하여 실행 클럭 수는 줄여주지만 생존 범위가 길어져서 보다 많은 레지스터를 요구하게 된다. 그림 3(a)에서 $7을 예로 들어보면 정의 후 바로 사용되므로 생존 범위가 겹치는 변수는 m 하나이다. 그

```
제어 흐름 분석;
if 변환;
  제어 흐름 그래프 내의 제어 종속 정보를 근거로 if 변환될
    기본 블록 선택;
  원래의 비교 명령을 조건 정의 명령으로 대치하고 불필요한
    분기문을 삭제;
  분기 후에 나오는 선택 경로 상의 명령들을 조건 실행 명령으로
    변환;
조건 분석;
  분할 그래프 작성;
  그래프 트래버설을 통해 서로 배타적 조건 결정;
  배타적 변수 집합 결정;
코드 이동;
  독립적인 명령을 찾기 위한 자료 흐름 분석;
  if ( 자료 종속으로 인해 병렬 처리 할 코드가 없는 경우 )
    자료 종속 관계가 없는 명령을 이동;
자료 흐름 분석;
  코드 이동 후 코드에 대해 자료 흐름 분석;
간섭 그래프 구성;
  조건 분석에서 서로 다른 배타적 변수 집합에 속한 변수간의
    에지 제거;
k = 가용 레지스터 수;
repeat
{
   repeat
   {
     if ( 노드의 차수가 k보다 작은 노드가 존재 )
     {
       간섭 그래프로부터 노드와 노드에 연결된 모든 에지 제거;
       컬러링 스택에 노드 push;
     }
     else
     {
       노드 차수가 k보다 크거나 같은 노드 중 대피 변수 선정;
       선택된 노드와 이에 연결된 에지를 간섭 그래프에서 제거;
       대피될 노드로 표시;
       컬러링 스택에 push;
     }
   } until ( 그래프가 빌 때까지 )
   대피될 노드로 표시된 것이 있으면 대피 코드 삽입;
   간섭 그래프 재구성;
} until ( 더 이상의 대피 코드가 없을 때까지 )
repeat
{
   컬러링 스택으로부터 노드를 pop;
   간섭 그래프에 노드와 연결된 에지 삽입;
   다른 노드와 구별되는 색 배정;
} until ( 컬러링 스택이 빌 때까지 )
```

그림 4 RASPE 알고리즘

러나 그림 3(d) 코드 이동 후의 코드에서 보면 m, k, x, $9, $10 총 5개의 변수와 생존 범위가 겹치게 된다. 이렇게 간섭 그래프 상의 에지 수가 증가하게 되어 레지스터 할당을 어렵게 만든다. 이를 보안하기 위해 조건 분석 실시 후 얻어낸 배타적 변수 집합을 사용한다. 서로 다른 배타적 변수 집합에 속한 변수들 간에 생존 범위가 겹치는 것은 이들이 동시에 실행되지 않으므로 간섭 그래프 상에 에지로 포함시키지 않는다. 즉 $7과 $9, $7과 $10 간의 에지는 간섭 그래프에서 제거시킬 수 있다. 결국 간섭 그래프가 단순해져 레지스터 할당이 용이해지고 대피 코드 발생 가능성도 줄어든다.

전체적인 RASPE 알고리즘은 그림 4와 같다.

3.1 RASPE 알고리즘의 적용예

C로 작성한 퀵 소트 프로그램을 예로 들어 RASPE 알고리즘을 적용해 레지스터 할당 과정을 살펴보면 다음과 같다.

우선 그림 5의 C 프로그램을 중간 언어로 변환하여 그림 6(a)와 같은 3주소 중간 코드를 얻는다. 이 코드의 제어 흐름을 분석하여 IF 변환할 조건 분기문을 선택한다. 조건 분기는 $9와 a를 비교하여 클 때와 작거나 같을 때로 나누어 실행이 된다. 이를 위해 사용된 비교 명령이 CMPLE $20, $9, a이다. $9가 a보다 클 때를 조건 레지스터 $50, $9가 a보다 작거나 같을 때를 $51로 설정하여 조건 정의 명령으로 대치하고 분기 후에 나오는 선택 경로 상의 명령들을 조건 실행 명령으로 변환한 후 불필요해진 분기문들을 제거한다. 이렇게 IF 변환을 완결하면 그림 6(b)의 코드가 된다. 조건 분석을 실시하기 위해 조건 실행 그래프를 작성하여 포함 관계를 분석하여 배타적인 변수 집합을 구한다. 그리고 나서 결과에는 영향을 주지 않으면서 서로 독립적인 명령들을 동시에 실행할 수 있

```
#define n 20
int key[20] = {32,1,19,16,13,10,6,7,17,9,5,12,11,4,14,15,3,8,18,2};
void main()
{
  int a, j, I, cnt=0, k=10000, flag;
  for (j=1; j<n; j++) {
    a=key[j];       flag=1;
    for (i=j-1;i>=0 && flag;i--) {
      if(key[i]>a) {
        key[i+1]=key[i];   cnt=cnt+1;   k=k+2;
      }
      else {
        k=k-5;   flag=0;   i++;
      }
    }
    key[i+1]=a;
  }
}
```

그림 5 퀵 소트 프로그램

```
MOV   cnt ,#0
MOV   k, #10000
MOV   j, #1
L.2:
SHLA  $5, j, #2
LDR   $6, key($5)
MOV   a, $6
MOV   flag , #1
SUB   $7, j, #1
MOV   i, $7
JUMP  L.9
L.6
SHLA  $8, i, #2
LDR   $9, key($8)
CMPLE $20, $9, a
JT    $20, L.10
SHLA  $10, i, #2
LDR   $11, key($10)
STR   $11, key+4($10)
ADD   $12, cnt, #1
MOV   cnt, $12
ADD   $13, k, #2
MOV   k, $13
JUMP  L.11
L.10:
SUB   $14, k, #5
MOV   k, $14
MOV   flag, #0
ADD   $15, i, #1
MOV   i, $15
SUB   $16, i, #1
MOV   i, $16
L.9:
CMPLT $21, i, #0
JT    $21, L.13
CMPNE $22, flag, #0
JT    $22, L.6
L.13:
SHLA  $17, i, #2
STR   a, key+4($17)
ADD   $18, j, #1
MOV   j, $18
CMPLT $23, j, #20
JT    $23, L.2
(a)

MOV   cnt, #0
MOV   k, #10000
MOV   j, #1
L.2:
SHLA  $5, j, #2
LDR   $6, key($5)
MOV   a, $6
MOV   flag, #1
SUB   $7, j, #1
MOV   i, $7
JUMP  L.9
L.6:
SHLA  $8, i, #2
LDR   $9, key($8)
CMPGT $50, $9,a
CMPLE $51, $9,a
SHLA  $10,i,#2      IF $50
LDR   $11,key($10)  IF $50
STR   $11,key+4($10)IF$50
ADD   $12,cnt,#1    IF $50
MOV   cnt, $12      IF $50
ADD   $13, k, #2    IF $50
MOV   k, $13        IF $50
SUB   $14, k, #5    IF $51
MOV   k, $14        IF $51
MOV   flag, #0      IF $51
ADD   $15, i, #1    IF $51
MOV   i, $15        IF $51
SUB   $16, i, #1
MOV   i, $16
L.9:
CMPLT $21, i, #0
JT    $21, L.13
CMPNE $22, flag, #0
JT    $22, L.6
L.13:
SHLA  $17, i, #2
STR   a, key+4($17)
ADD   $18, j, #1
MOV   j, $18
CMPLT $23, j, #20
JT    $23, L.2
(b)

MOV   cnt, #0
MOV   k, #10000
MOV   j, #1
L.2:
SHLA  $5, j, #2
LDR   $6, key($5)
MOV   a, $6
MOV   flag, #1
SUB   $7, j, #1
MOV   i, $7
JUMP  L.9
L.6:
SHLA  $8, i, #2
LDR   $9, key($8)
CMPGT $50, $9, a
CMPLE $51, $9, a
SHLA  $10, i, #2    IF
nk$50
SUB   $14, k, #5    IF $51
LDR   $11,key($10)IF $50
MOV   k, $14        IF $51
MOV   flag, #0      IF $51
ADD   $15, i, #1    IF $51
STR   $11,key+4($10)IF$50
ADD   $12, cnt, #1 IF $50
MOV   i, $15        IF $51
MOV   cnt, $12      IF $50
ADD   $13, k, #2    IF $50
SUB   $16, i, #1
MOV   k, $13        IF $50
MOV   i, $16
L.9:
CMPLT $21, i, #0
JT    $21, L.13
CMPNE $22, flag, #0
JT    $22, L.6
L.13:
SHLA  $17, i, #2
STR   a, key+4($17)
ADD   $18, j, #1
MOV   j, $18
CMPLT $23, j, #20
JT    $23, L.2
(c)
```

그림 6 (a) 예제의 중간코드 (b) IF 변환 코드 (c) 코드 이동후 코드

도록 코드 이동을 실시하여 그림 6(c) 코드를 얻는다.

표1은 자료 흐름 분석을 통해 구축한 간섭그래프의 비트 벡터를 나타낸다. 두 노드, 즉 레지스터를 할당 받고자하는 프로그램 변수나 임시 변수의 생존 범위가 겹치면 에지가 존재하는 것으로 비트 값은 1로 표시하고 그렇지 않은 경우는 0으로 표시한다. 조건 분석으로 얻은 배타적 변수 집합은 {$10, $11, $12, $13}과 {$14, $15}이다. 서로 다른 배타적인 집합에 속하는 원소간의 에지는 의미가 없는 것으로 간섭 그래프에서 제외된다. 이렇게 제거될 에지를 표시하기 위해 표 1에서 비트 값 1에 괄호를 사용하여 표현했다. 이들을 제거한 후 시스템에서 사용 가능한 레지스터들을 가지고 단순화 과정을 거치며 대피가 필요한 경우 적절한 대피 코드를 삽입하고 간섭 그래프를 재구성한 후 단순화 과정이 완결될 때까지 단계를 반복한다. 마지막으로 실제 레지스터를 배정한다.

가용 레지스터 수를 5라고 가정한 단순화 과정을 보면 다음과 같다.

($18, 0), ($17, 0), ($22, 0), ($6, 0), ($5, 0), ($21, 0), (flag, 1), ($16, 0), ($13, 0) , ($12, 0) , ($7, 0) , (count, 1), ($15, 0), ($14, 0), ($9, 0), ($8, 0), ($11, 1), ($10, 0), (a, 0), (i, 0), (j, 0), (k, 0)

표 1 자료 흐름 분석을 통해 구축한 간섭 그래프의 비트 벡터

	cnt	k	j	i	flag	a	$5	$6	$7	$8	$9	$10	$11	$12	$13	$14	$15	$16	$17	$18	$21	$22
cnt	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	0	0	0
k	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0
i	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
j	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1
flag	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0
a	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1
$5	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$6	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$7	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$8	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$9	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$10	1	1	1	1	1	1	0	0	0	0	0	0	1	0	0	(1)	(1)	0	0	0	0	0
$11	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	(1)	(1)	0	0	0	0	0
$12	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	(1)	0	0	0	0	0
$13	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
$14	1	1	1	1	1	1	0	0	0	0	0	(1)	(1)	0	0	0	0	0	0	0	0	0
$15	1	1	1	1	1	1	0	0	0	0	0	(1)	(1)	(1)	0	0	0	0	0	0	0	0
$16	0	0	1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$17	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$18	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$21	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$22	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

여기서 쌍으로 표시한 자료 중 첫 번째 인자는 레지스터를 할당 받아야하는 변수이고, 두 번째 인자가 0이면 컬러링 스택에 푸시된 것, 1이면 대피 변수로 선정된 것이다. 결과적으로 그림 8의 예제에서는 3개의 대피 변수가 발생하였다. 만일 조건 분석을 하지 않으면 다음과 같은 단순화 과정을 거치며 6개의 대피 변수가 발생한다.
($18, 0), ($17, 0), ($22, 0), ($6, 0), ($5, 0), ($21, 0), (flag, 1), ($16, 0), ($13, 0), ($7, 0), ($15, 1), ($12, 0), ($14, 1), ($11, 1), ($8, 1), (count, 1), ($10, 0), ($9, 0), (a, 0), (i, 0), (j, 0), (k, 0)

4. 실험 결과

본 논문에서 제안한 RASPE 알고리즘의 타당성을 검증하기 위해 11개의 벤치마크 프로그램을 선정하여 실험하였다. Chaitin의 레지스터 할당 기법과 RASPE 레지스터 할당 기법의 성능 비교를 위해 실험기(simulator)를 구현하였다. 실험기의 구조는 그림 7과 같다. 첫 단계는 실험기에서 사용하는 토큰을 찾아내기 위해 어휘 분석을 해주는 파일과 구문 분석을 해주는 파일로 구성되어 있다. 중간 언어의 각 행을 읽어들여 해당 행의 명령 및 심볼에 대한 자료구조를 구축한다. 프로그램의 시작에서 전역 해시 테이블(hash table)을 생성하고 각 함수의 시작에서 지역 해시 테이블을 생성한다. 심볼은 사용 영역(scope)에 따라 전역 혹은 지역 해시 테이블에 입력하여 운영한다. 각 명령은 분석하여 명령어 형(type), 행 번호, 주소 값, 레이블 이름, 연산 코드, 오퍼랜드, 조건 레지스터 등의 정보를 연결 리스트(linked list)로 구성한다. 두 번째 단계에서는 첫 단계에서 구축된 각 행의 자료구조를 읽어들여 파이프라인 스테이지별로 구분하여 모의실험을 실시한다. 4개의 명령을 동시에 처리하기 위해 명령간의 자료 종속 관계를 조사하여 자료 종속 관계가 없는 명령에 대해서만 동시에 실행하도록 스케줄한다. 조건 실행의 경우는 각 파이프라인 단계에서 조건 기술자, 즉 조건 레지스터의 값으로 명령의 실행 여부를 결정한다. 실험기는 수행 후 메모리의 내용, 실행 클럭 수, 명령 실행 과정을 보여주는 실행 추적 파일 등을 생성한다.

본 논문에서 제시한 레지스터 할당 기법의 타당성을

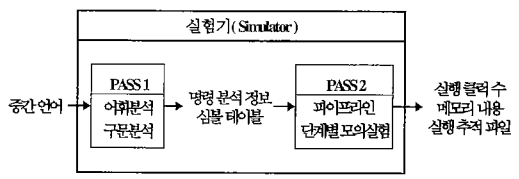


그림 7 실험기의 구성도

입증하기 위해서는 레지스터 할당의 목적인 대피 코드 수의 감소와 실행 시간의 감소를 보여야한다. 이를 위해 실험기를 이용하여 기존의 레지스터 할당 알고리즘과 실행 시간을 비교하였다. 기존의 알고리즘으로 레지스터 할당의 대표격인 Chaitin의 레지스터 할당 알고리즘을 적용한 이유는 레지스터 할당 알고리즘에 관한 연구는 Chaitin 이후에도 계속 되었지만 이들은 본 논문의 대상이 되는 ILP 프로세서를 위한 개선책이 아니므로 큰 의미가 없기 때문이다. Chaitin의 레지스터 할당 알고리즘을 적용한 경우와 RASPE 알고리즘을 적용한 경우에 대해 실행 클럭 수를 비교하여 평균 19.38% 실행 클럭 수가 감소함을 확인하였고 그 결과는 표 2와 같다.

표 2 레지스터 할당 알고리즘의 실행 클럭 수 비교표

	Chaitin	RASPE	감소율(%)
단어수세기	1766	1567	11.27
배열의차	217	136	37.33
버블소트	2947	2121	28.03
삽입소트	515	444	13.79
선택소트	2488	2020	18.81
쉘 소트	2858	2852	0.21
이진탐색	1633	1318	19.29
최대최소찾기	207	168	18.84
패턴매칭	265	198	25.28
피보나치수열	367	299	18.53
퀵 소트	2799	2188	21.83

표 3 간섭 그래프 구성상의 에지 수 비교표

	방법1	방법2	감소율(%)
단어수세기	320	318	0.63
배열의차	321	283	11.83
버블소트	153	145	5.23
삽입소트	245	237	3.27
선택소트	173	169	2.31
쉘 소트	357	346	3.08
이진탐색	274	260	5.10
최대최소찾기	232	224	5.17
패턴매칭	120	114	5.00
피보나치수열	186	182	2.15
퀵 소트	184	174	5.43

RASPE에서 조건 실행을 적용하는 것이 타당하다는 것을 보이기 위해 조건 분석을 실시하지 않은 경우 (방법1)과 조건 분석을 실시한 경우 (방법2)로 나누어 실험하였다. 우선 레지스터 할당의 용이성을 보이기 위해 간섭그래프 구성상의 에지 수를 비교하였고 대피 코드 수의 감소를 보이기 위해 대피 변수 개수를 비교하였다. 간섭그래프 구성상의 에지 수를 비교해보면 조건 분석을 실시하여 배타적 변수 집합을 구해 불필요한 에지를 삭제한 결과 평균 4.47% 에지가 감소하였고 각 결과는 표 3과 같다.

실제 프로그램에 RASPE를 적용할 경우 가용 레지스터 수와 실제 필요로 하는 레지스터 수간의 차이에 따라 어떤 효과가 있는지를 알아보기 위해 가용 레지스터 수를 1개에서 10개까지로 변화시켜 가면서 레지스터 할당 알고리즘을 적용하여 대피 변수 개수를 비교한 결과가 표 4이고 대피 변수의 상대적 감소율을 그래프로 표현한 것이 그림 8이다.

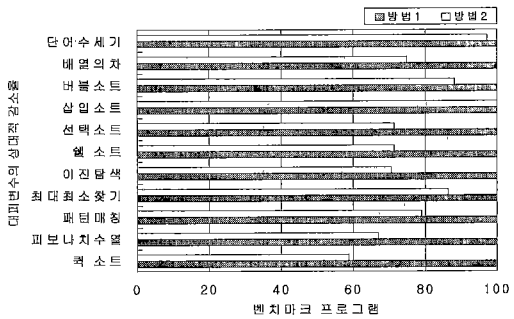


그림 8 대피 변수의 상대적 감소율 그래프

표 4 대피 변수 개수 비교 표

벤치마크 프로그램		가용 레지스터 수에 따른 대피 변수 개수									
		1	2	3	4	5	6	7	8	9	10
단어 수세기	방법1	15	10	5	4	2	1	0	0	0	0
	방법2	14	10	5	4	2	1	0	0	0	0
배열 의차	방법1	19	13	11	9	7	7	4	3	2	1
	방법2	16	11	9	7	5	6	2	1	0	0
버블 소트	방법1	12	11	8	7	3	1	0	0	0	0
	방법2	11	10	7	6	2	1	0	0	0	0
삽입 소트	방법1	13	9	6	5	4	3	1	0	0	0
	방법2	10	10	6	5	4	3	1	0	0	0
선택 소트	방법1	9	12	6	5	4	3	2	1	0	0
	방법2	8	7	5	4	3	2	1	0	0	0
쉘 소트	방법1	11	10	9	8	7	6	5	4	2	1
	방법2	9	8	7	6	5	4	3	2	1	0
이진 탐색	방법1	15	13	12	10	8	5	4	1	0	0
	방법2	13	11	10	8	3	2	1	0	0	0
최대 최소찾기	방법1	15	8	7	5	4	3	2	0	0	0
	방법2	12	7	7	6	3	2	1	0	0	0
패턴 매칭	방법1	15	12	9	6	1	0	0	0	0	0
	방법2	13	10	7	4	0	0	0	0	0	0
피보나치수 열	방법1	16	15	14	13	11	10	6	2	1	0
	방법2	9	7	6	12	11	7	6	1	0	0
퀵 소트	방법1	10	9	11	6	6	5	3	1	0	0
	방법2	8	7	5	4	3	2	1	0	0	0

이상의 실험을 통해서 다음과 같은 결과를 확인하였다. 본 논문에서 제안한 RASPE 알고리즘을 적용하여 레지스터 할당을 실시하면 Chaitin 알고리즘을 적용하여 레지스터 할당을 실시하는 경우와 비교해서 평균적으로 19.38% 실행 클럭 수를 감소시켰다. 또한 조건 분석을 실시하였기 때문에 간섭 그래프 상의 에지 수가 4.47% 감소하였으며 대피 변수 수도 21.35% 감소함으로써 제안한 방법의 타당성을 검증하였다.

5. 결 론

본 논문에서는 조건 분기를 처리하는 조건 실행과 기존의 그래프 컬러링 레지스터 할당을 결합한 새로운 레지스터 할당 알고리즘을 제안했다. 컴파일러가 생성한 중간 코드를 입력으로 받아 IF 변환으로 조건 분기 명령을 제거한 후 병렬 처리할 명령 수를 증가시키기 위해 코드 이동을 실시한다. 자료 흐름 분석을 통해 레지스터 할당의 기초 정보를 구하고 조건 분석을 실시하여 불필요한 간섭 그래프 에지를 제거한 후 간섭 그래프를 구축한다. 단순화 단계에서 필요한 경우 대피 변수를 선정하여 대피 코드를 삽입하고 간섭 그래프를 재구성하여 단순화가 완결될 때까지 반복 처리한다. 끝으로 실제 가용 레지스터를 배정하여 레지스터 할당을 마친다.

본 알고리즘의 타당성을 검증하기 위해 실험한 결과 조건 분석을 실시하여 간섭 그래프의 에지수를 4.47% 감소시켰고 그 결과 요구되는 대피 변수의 수가 21.35% 감소되었다. 그리고 기존의 Chaitin 알고리즘 보다 19.38%의 성능 향상을 이루었다. 제안한 기법은 조건 실행을 통해 조건 분기 명령을 제거하여 기본 블록 내의 명령어 수를 증가시켜 병렬처리의 기회를 증진시켰으며 조건 분석을 통해 간섭 그래프의 불필요한 에지를 제거시켜 보다 효율적인 레지스터 할당을 실현하였다. 조건 분기 명령이 많이 포함되며 이들 분기 명령이 다량의 반복을 처리하는 응용 프로그램의 경우 더욱 효과적일 수 있음을 보였다.

향후 연구 과제는 SPEC 벤치마크 프로그램을 가지고 실험할 수 있도록 제안한 레지스터 할당 알고리즘을 더욱 확장하고자 한다. 그리고 다양한 최적화 기법들을 추가하여 더욱 높은 병렬성을 꾀할 수 있는 최적화 컴파일러를 개발하고자 한다.

참 고 문 헌

[1] D. Patterson and J. Hennessy, Computer Architecture a Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc.

[2] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton and W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture,' Proceedings of the 25th International Symposium on Computer Architecture, July 1998.

[3] R. Johmson and M. Schlansker, "Analysis Techniques for Predicated Code," In Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 100-113, Dec. 1996.

[4] G. Lu, "Issues in Register Allocation by Graph Coloring," CMU-CS-96-171, Nov. 1996.

[5] D. Pnevmatikatos and G. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processor," International Symposium on Computer, pp. 120-129, 1994.

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein,"Register Allocation via Graph Coloring," Computer Languages, Vol. 6, No. 1, pp. 47-57, 1981.

[7] P. Briggs, "Register Allocation via Graph Coloring," Ph. D Thesis, RICE Univ., 1992.

[8] S. Mahlke, "Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches," Ph. D. Thesis, Univ. of illinois, 1996.

[9] F. Chow and J. Hennessy, "The priority-based coloring approach to register allocation," ACM Transactions on Programming Languages and Systems, pp. 501-536, 1990.

[10] D. Gillies, D. Ju, R. Johnson and M. Schlansker, "Global Predicate Analysis and its Application to Register Allocation," MICRO-29, pp. 114-125, 1996.

[11] T. Kong and K. Wilken, "Precise Register Allocation for Irregular Architectures," Micro-31, pp. 297-307, 1998.

[12] D. Lin, "Compiler Support for Predicated Execution in Superscalar Processors," Master Thesis, Univ. of illinois, 1990.

[13] W. Hwu, R. Hank, D. Gallagher, S. Mahlke, D. Lavery, G. Haab, J. Gyllenhaal and D. August, "Compiler Technology for Future Microprocessors," Proceedings of the IEEE, Vol. 83, No. 12, pp. 1625-1640, 1995.

[14] S. Mahlke, D. Lin, W. Chen, R. Hank and R. Bringmann, "Effective Compiler Support for Predicates Execution Using the Hyperblock," Micro-25, 1992.

[15] J. McCormick, Jr., "Supporting Predicated Execution: Techniques and tradeoffs," Master Thesis, Univ. of illinois, 1996.

[16] D. August, J. Puiatti, S. Mahlke, D. Connors, K. Crozier and W. Hwu, "The Program Decision

Logic Approach to Predicated Execution," ISCA99, 1999.
[17] 이기호, 신화정, "조건 실행을 고려한 레지스터 할당 알고리즘", 한국정보과학회 춘계 학술발표논문집(A) 제 26권 1호, pp. 57-59, 1999.

신 화 정
1985년 2월 이화여자대학교 자연과학대학 전자계산학과 이학사. 1987년 2월 이화여자대학교 대학원 전자계산학과 이학석사. 1992년 3월 ~ 현재 이화여자대학교 과학기술대학원 컴퓨터학과 박사과정. 관심분야는 프로그래밍 언어, 컴파일러, 병렬처리

이 기 호
1961년 이화여자대학교 수학과(이학사, 석사). 1968년 ~ 1972년 텍사스 주립대(전산학 석사, 박사 수료). 1981년 서울대학교 대학원(전산학 박사). 1987년 이화여자대학교 전산연구소장. 1988년 캘리포니아 대학교 연구교수. 1973년 ~ 현재 이화여자대학교 컴퓨터학과 교수. 관심분야는 프로그래밍 언어론, 컴파일러, S/W 공학, ICAI 등임.