EPIC 아키텍쳐를 위한 적극적 레지스터 할당 알고리듬

최 준 기† · 이 상 정††

요 약

최근 많은 명령어 수준 병렬 처리 기술들이 개발되면서 ILP 프로세서 성능이 급격히 증가하고 있다. 특히, 새로운 기술로 주목 받고 있는 EPIC(Explicitly Parallel Instruction Computing) 아키텍쳐는 조건실행(Predicated Execution)과 투기적실행(Speculative Execution)을 하드웨어와 접목하여 성능 향상을 시도하고있다.

본 논문에서는 EPIC 아키텍쳐의 특성을 최대로 활용하여 코드 스케줄 가능성을 높이는 새로운 레지스터 할당 알고리듬을 제안한다. 그리고, 제안된 레지스터 할당 알고리듬은 조건실행의 적용으로 인하여 더욱 효율을 높일 수 있음을 실험을 통하여 입증한다. 실험 결과 기존의 레지스터 할당 방법에 비하여 평균 19%의 성능 향상을 보임으로써 제안된 레지스터 할당 방법이 효과적임을 검증한다.

An Aggressive Register Allocation Algorithm for EPIC Architectures

Joon-Kee Choi† · Sang-Jeong Lee††

ABSTRACT

Recently, many parallel processing technologics were developed, ILP(Instruction Level Parallelism) processor's performance have been growed very rapidly. Especially, EPIC(Explicitly Parallel Instruction Computing) architectures attempt to enhance the performance in the predicated execution and speculative execution with the hardware.

In this paper, to improve the code scheduling possibility by applying to the characteristics of EPIC architectures, a new register allocation algorithm is proposed. And we proves that proposed register allocation algorithm is more efficient scheme than the conventional scheme when predicated execution is applied to our scheme by experiments. In experimental results, it shows much more performance enhancement, about 19% in proposed scheme than the conventional scheme. So, our scheme is verified that it is an effective register allocation method.

1. 서 론

최근 하드웨어 기술이 발전함에 따라서 더욱 많은 명령어 수준에서의 병렬성을 추출하기 위하여 컴파일러의 적극적인 개입이 보편화되고 있는 추세이다[1]. 하드웨어만으로 프로세서의 성능을 극대화하기에는 복잡한 하드웨어 구조와 비용 때문에 어렵고 이를 보완할 수 있는 방법으로 컴파일러의 역할을 증대시키고 있다[2]. 특히, VLIW를 발전시킨 EPIC 아키텍쳐는 HP와 Intel의 차세대 프로세서인 Merced 아키텍쳐의 IA-64 명령어 세트 발표시에 처음 사용된 용어로 컴파

† 준 회 원 : 신성대학 교수
†† 정 회 원 : 순천향대학교 컴퓨터학부 교수
논문접수 : 1998년 8월 27일, 심사완료 : 1998년 11월 16일

일러가 조건분기 명령을 줄임으로써 코드 스케줄의 가능성을 증가시키는 조건실행과 분기예측을 통하여 적극적으로 명령들을 이동시키는 투기적실행을 통합하여 성능을 극대화하고자 하는 새로운 방법을 시도하고 있다[3, 4, 5, 6, 16]. 또한, 컴파일러의 역할을 증대시키는 방법 중 하드웨어 자원인 레지스터를 적절히 이용하도록 하는 것은 최적화 컴파일러가 해야 할 필수적인 요소 중의 하나이다. 이러한 레지스터 할당(register allocation) 기법은 가능한 빠른 저장장치인 레지스터를 이용하고 메모리의 이용은 극소화할 수 있도록 레지스터에 변수들을 적절히 할당해 주어야 한다. 그리고 명령어를 재정렬함으로써 성능 향상을 꾀하는 코드 스케줄(code schedule) 기법도 성능 향상을 위한 중요한 최적화 기법의 하나이다.

최근의 최적화 컴파일러에서 많은 연구가 진행중인 분야중의 하나는 레지스터 할당과 코드 스케줄을 같은 단계에서 수행하도록 컴파일러를 구성하는 것이다. 예를 들면, 레지스터 할당을 먼저 수행한 다음 코드 스케줄을 수행하는 방법의 경우 레지스터 할당만을 고려하게 되면 레지스터를 되도록 적게 이용하도록 변수들을 레지스터에 배정하기 때문에 자료 종속 관계 증가로 코드 스케줄을 어렵게 만들 수 있다. 또한, 코드 스케줄을 먼저 수행하고 다음으로 레지스터 할당을 수행하는 방법의 경우는 코드 스케줄을 수행한 후 명령어의 이동으로 인하여 변수들의 생존범위가 길어져 많은 레지스터들을 사용해야 하기 때문에 레지스터 할당을 어렵게 만들 수 있다. 특히, 전자의 경우 레지스터 할당시 코드 스케줄까지 고려하여 레지스터를 할당해야 하기 때문에 레지스터 할당 알고리듬은 중요한 부분을 차지한다. 레지스터 할당은 Chaitin의 그래프 컬러링 레지스터 할당 알고리듬을 많이 응용한다. Chaitin의 레지스터 할당 알고리듬은 프로그램의 제어 흐름을 분석하여 제어 흐름 그래프(control flow graph)를 구성하고, 변수들의 간섭관계를 나타내는 간섭그래프(interference graph)를 구성한다. 그리고 마지막 단계에서 레지스터를 할당한다[7, 12].

본 논문에서는 기존 Chaitin의 레지스터 할당 알고리듬을 확장하여 EPIC 아키텍처를 위한 효율적인 레지스터 할당 알고리듬을 제안한다. 제안한 레지스터 할당 알고리듬은 코드 스케줄의 효율을 높이도록 간섭그래프 구성 후 자료 종속 그래프(data dependence graph)를 구성하여 개별 노드의 종속 거리를 구하고 노드간 종속 거리의 차를 구한다. DDG에서 구해진 정보를 토대로 하여 레지스터 할당시 가용 레지스터(available registers)를 전부 사용하도록 하고 레지스터를 골고루 배정하도록 한다. 기존 Chaitin의 레지스터 할당 알고리듬은 레지스터를 적게 사용하기 때문에 자료 종속 관계가 증가하여 코드 스케줄 시에 많은 제약이 따른다. 그러나, 제안한 레지스터 할당 알고리듬은 자료 종속 관계를 줄이도록 함으로써 코드 스케줄 가능성을 높인다. 특히, 조건실행으로 인하여 기본블록내의 명령수가 많아진 경우 더욱 효율적이도록 레지스터 할당 알고리듬을 구성한다. 실험 결과 제안한 레지스터 할당 알고리듬을 조건실행과 결합했을 때가 조건실행과 결합하지 않았을 때에 비하여 평균 15% 그리고 기존 레지스터 할당 알고리듬과 제안한 레지스터 할당 알고리듬 각각에 대하여 조건실행과 결합하여 실험했을 때 제안한 레지스터 할당 알고리듬 적용시 평균 19%의 성능 향상을 보임으로써 제안한 레지스터 할당 알고리듬이 조건실행과 결합했을 때 더욱 효과적임을 검증한다.

본 논문의 구성은 2장에서 EPIC 아키텍처, 조건실행 그리고 투기적실행에 대하여 설명하고, 3장에서는 제안하고자 하는 EPIC 아키텍처를 위한 적극적 레지스터 할당 알고리듬에 대하여 소개한다. 여기서, 기존의 레지스터 할당 알고리듬과 비교 설명하며 4장에서는 제안한 알고리듬의 타당성을 검증하기 위한 실험 방법 및 성능 측정 결과를 보이고 마지막으로 결론 및 연구 과제에 대하여 논한다.

2. EPIC 아키텍처

EPIC는 HP와 Intel 차세대 프로세서인 Merced 아키텍처의 IA-64 명령어 세트 발표시에 처음 사용된 용어이다[8]. 이는 모든 프로그램에서 컴파일러가 명령어 수준에서의 병렬처리(Instruction Level Parallelism, ILP) 능력을 향상시키는 것을 용이하게 하는 특징들을 제공하는 아키텍처를 의미한다. 특히, EPIC 아키텍처는 조건실행과 투기적실행을 같이 결합하여 코드 스케줄 능력을 향상시킴으로써 성능 향상을 꾀한다. 여기서, 조건실행은 조건문을 조건기술(predicates, guards)을 통하여 없애고 하나의 기본블록으로 통합하기 때문에 코드 스케줄 기회를 더욱 많이 갖는다. 그러나, 하드웨어 자원인 레지스터를 적절히 이용하지 못하면 자

료 종속 관계로 인한 코드 스케줄 제약으로 오히려 성능이 감소할 수 있다. 따라서, 본 논문에서는 조건실행을 새로운 레지스터 할당 방법과 결합하여 더욱 높은 성능 향상을 얻고자 한다. 다음 절에서 조건실행 적용 과정을 간단한 예를 통하여 보임으로써 제안하고자 하는 레지스터 할당 알고리듬과의 결합시 성능을 높일 수 있는 가능성이 존재함을 보인다. 또한, 컴파일 시에 분기를 예측하고 이를 통하여 분기를 넘어선 명령어 이동을 함으로써 성능을 향상시키는 투기적실행에 대하여도 간단히 언급한다.

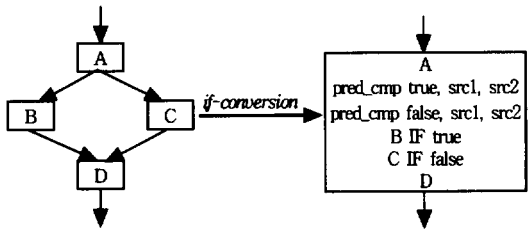
2.1 조건실행(Predicated Execution)

프로세서의 성능은 각 명령 사이의 자료 종속 관계, 분기명령 등으로 인하여 저하되고 많은 병렬성 추출의 장애 요소가 된다. 조건실행은 조건기술 레지스터(predicate register)로 구현된 불(boolean) 조건기술(predicates, guards)에 의해 각 명령들의 조건실행을 지원하여 조건분기를 제거하는 기법이다. 컴파일러는 if 변환을 통해 조건분기를 조건지정 정의(predicate defining) 명령으로 변환하고 계산된 조건기술하에서 명령을 보호한다. 조건정의 명령의 형태는 다음과 같이 정의된다.

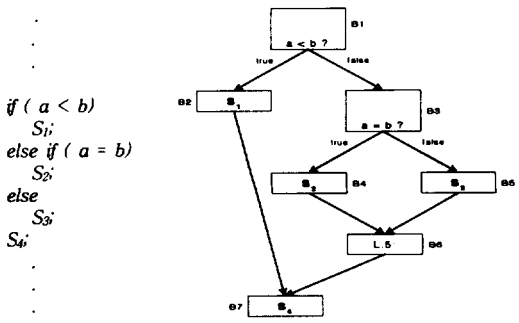
pred_<cmp> _Pout, src1, src2

여기서, cmp는 CMPGE, CMPGT, CMPLE, CMPLT 등의 명령어가 될 수 있다. cmp 명령에 의하여 src1과 src2를 비교한 후 결과가 참(true)이면 조건기술 _Pout을 가진 명령들만 수행하고, 참이 아니면 NOP(no operation)로 처리되며 수행 결과에는 아무런 영향을 미치지 않는다.

조건실행을 적용하기 위해서는 생성된 중간언어에 대한 제어의 흐름을 나타내는 제어 흐름 그래프(control flow graph)를 구성한다. 제어 흐름 그래프는 기본블록을 노드(node)로 하고 제어의 흐름을 에지(edge)로 하는 G = (N, A, s) 형식의 방향성 그래프이다. 여기서, N은 각 노드의 집합, A는 에지의 집합 그리고 s는 초기 노드를 나타낸다. (그림 1)은 if 변환의 예이다. (그림 1)의 기본블록 A에서 조건이 참이면 기본블록 B로 보호되고 거짓이면 기본블록 C로 보호된다. 따라서, 조건 값에 의하여 다음 명령을 수행하기 때문에 if 변환 후에는 기본블록이 합쳐져도 프로그램의 수행에는 잘못된 영향을 주지 않는다. 여러 개의 기본블록이 if 변환 후에 한 개의 단일블록으로 합쳐졌음을 볼 수 있다.

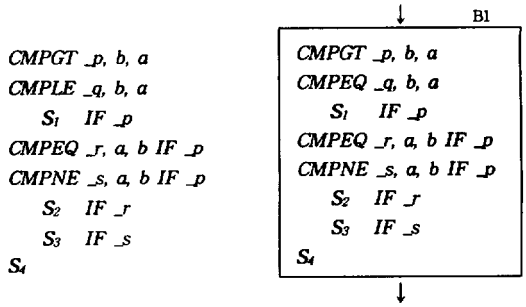


(그림 1) if 변환의 예
(Fig. 1) An example of if-conversion



(a) 코드 예
(a) An example of code

(b) 제어 흐름 그래프
(b) Control flow graph



(c) if 변환후의 코드
(c) Codes after if-conversion

(d) if 변환후의 제어 흐름 그래프
(d) Control flow graph after if-conversion

(그림 2) if 변환 과정
(Fig. 2) Processes of if-conversion

(그림 2)는 간단한 예에 대하여 조건실행을 행하는 과정을 보인다. (a)의 소스 코드에 대하여 제어 흐름 그래프를 구성한 결과는 (b)이다.

기본블록 B1에서 a < b의 결과에 따라 참이면 기

본블록 B2를 수행하고 거짓이면 기본블록 B3를 수행한다. 또한 기본블록 B3에서 다시 a = b의 결과가 참이면 기본블록 B4를 수행하고 거짓이면 기본블록 B5를 수행한다. 즉, B2와 B3는 B1에 의하여 보호되고 B4와 B5는 다시 B3에 의하여 보호된다. 따라서, 조건실행에 의하여 조건분기 명령을 조건정의 명령으로 바꾸고 if 변환을 통하여 후속 기본블록 내에 있는 명령들을 보호함으로써 조건분기가 있는 기본블록들을 한 개의 단일 기본블록으로 명령들을 정렬시킬 수 있다. if 변환된 결과는 (그림 2)의 (c)이고 (d)는 합쳐진 후의 제어 흐름 그래프이다. 결과적으로 두 개의 조건분기가 줄어들었고 각 기본블록이 한 개의 단일 블록으로 합쳐지면서 명령들이 정렬되었다.

기본블록 내의 명령수가 많아짐으로 해서 생길 수 있는 이점으로는 병렬처리의 가능성 증가이다. 명령어 병렬처리의 향상을 저해하는 요인 중의 하나인 분기문을 제거함으로써 다중 이슈(issue)를 가능하게 하고 분기문으로 인한 실행 클럭 사이클(execution clock cycles)을 줄일 수 있다. 따라서, 기본블록 내에 명령이 많아지면 코드 스케줄 등의 최적화 가능성 증가로 인하여 병렬처리될 명령의 수가 증가하게 되고 성능 향상의 기회가 증가한다.

2.2 투기적실행(Speculative Execution)

컴파일러가 제어하는 투기적실행은 컴파일 시에 미리 실행 결과를 예측하여 종속관계(dependent relations)를 제거한다. 종속관계는 크게 제어종속(control dependence)과 자료종속(data dependence)으로 나누어진다. 분기와 다른 명령들 사이에 발생하는 제어종속을 제거함으로써 컴파일러가 분기를 넘어 적극적으로 명령들을 이동시킬 수 있다. 또한, 메모리 명령간의 자료 흐름 종속 관계인 자료종속은 메모리 명령들이 서로 다른 메모리 위치를 참조한다는 가정을 컴파일러가 하도록 하여 자료종속을 제거함으로써 코드 스케줄의 가능성을 높인다. 투기적실행에서는 페이지 오류(page fault), TLB 미스, 그리고 캐쉬 미스(cache misses) 등과 같은 예외상황(exception) 발생시 이의 효과적인 복구(recovery)를 위한 과정이 필요하다. 그리고 메모리 종속 관계가 없는 것으로 가정하여 실행시 실제 자료종속 관계가 존재한 경우 효과적인 복구(repair)과정이 필요하다. 투기적실행에서 문제점 극복을 위한 방법 중 센티널 스케줄링(sentinel scheduling)은 컴파일러에 의해 스케줄된 명령들에 대한 예외상황을 정확하게 검출하고 복구할 수 있도록 한다[3, 6].

3. EPIC 아키텍처를 위한 적극적 레지스터 할당 알고리듬

컴파일러의 최적화 단계에서 하드웨어 자원인 레지스터를 적절히 이용하도록 하는 것은 최적화 컴파일러가 해야 할 필수요소 중의 하나이다[9]. 레지스터 할당의 목적은 가능한 빠른 저장장치인 레지스터를 이용하고 메모리의 이용은 극소화할 수 있도록 최적화시 변수들을 레지스터에 적절히 할당해 주어야 한다. 또한, 더욱 많은 명령어 수준에서의 병렬성을 추출하기 위해서는 원래 프로그램의 의미를 유지하면서 명령어들을 재정렬하기 위한 코드 스케줄 기법도 중요한 최적화 기법의 하나이다. 코드 스케줄을 효과적으로 수행하게 되면 병렬 수행할 수 있는 명령어의 수가 많아져 성능을 크게 개선할 수 있다. 그러나, 코드 스케줄을 효과적으로 수행하는 것도 중요하지만 프로그램의 흐름상 나누어지는 기본블록 내의 명령수가 적으면 코드 스케줄에 제약이 따를 수 있다. 따라서, 조건실행과 같이 기본블록의 수를 줄이고 대신 기본블록 내의 명령수를 많게 하면 코드 스케줄 범위가 커지기 때문에 더욱 많은 병렬성을 추출할 수 있다.

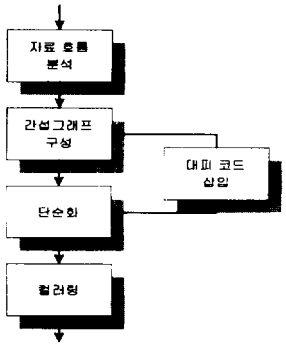
최근의 최적화 컴파일러에서 많은 연구가 진행중인 분야 중의 하나는 레지스터 할당과 코드 스케줄을 상호 고려하도록 컴파일러를 구성하는 것이다[10][11]. 이들을 적절히 이용하면 성능을 극대화시킬 수 있지만 상호 상반되는 관계에 있기 때문에 오히려 성능을 감소시킬 수 있다. 예를 들어, 레지스터 할당을 먼저 수행한 다음 코드 스케줄을 수행하는 경우 레지스터 할당은 레지스터를 되도록 적게 이용하면서 변수들을 배정하기 때문에 자료 종속 관계 증가로 인하여 코드 스케줄을 어렵게 만든다. 또한, 코드 스케줄을 먼저 수행하고 다음으로 레지스터 할당을 수행하는 경우는 코드 스케줄을 수행한 후 명령어의 이동으로 인하여 변수들의 생존범위가 길어져 많은 레지스터들을 사용해야 함으로써 레지스터 할당을 어렵게 만든다.

본 논문에서는 ILP 프로세서에서 명령어 수준 병렬성의 활용을 높이는 좀더 적극적인 코드 스케줄을 위해서 조건실행을 적용하고, 레지스터 할당시 EPIC 아키텍처를 위한 적극적인 레지스터 할당 알고리듬

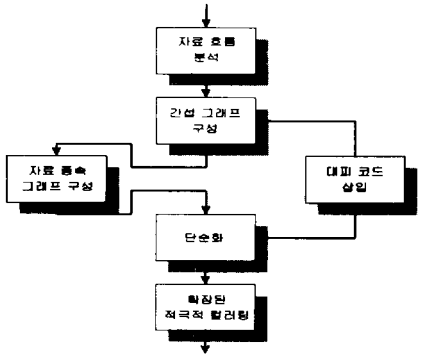
(Aggressive rEgister alloCation algorithm, AEC)을 제안한다. AEC는 기존에 많이 응용되고 있는 Chaitin의 그래프 컬러링 알고리듬을 확장한다[12]. 즉, if 변환을 거친 중간언어 상에서 간섭그래프를 구성하고 자료들의 종속 관계를 조사하기 위하여 자료 종속 그래프를 구성한다. 다음으로, 레지스터 할당시 가용 레지스터를 전부 이용하고 변수들의 자료 종속 거리가 멀리 떨어져 있는 변수들에 서로 같은 레지스터를 할당하여 레지스터가 고른 분포를 갖도록 하고 자료 종속 관계를 최대한 줄임으로써 명령어 스케줄의 효율을 높인다.

3.1 레지스터 할당 알고리듬

레지스터 할당은 그래프 컬러링 알고리듬을 최초로 적용하여 구현한 G. J. Chaitin에 의하여 골격을 갖추게 되었다[7]. 현재 진행중인 많은 연구들이 그래프 컬러링 레지스터 할당 알고리듬을 응용하는 추세이다. Chaitin의 레지스터 할당 흐름은 (그림 3)과 같고 본 논문에서 제안한 AEC의 흐름은 (그림 4)에서 보인다.



(그림 3) Chaitin의 레지스터 할당 과정
(Fig. 3) Chaitin's register allocation process



(그림 4) AEC 레지스터 할당 과정
(Fig. 4) AEC register allocation process

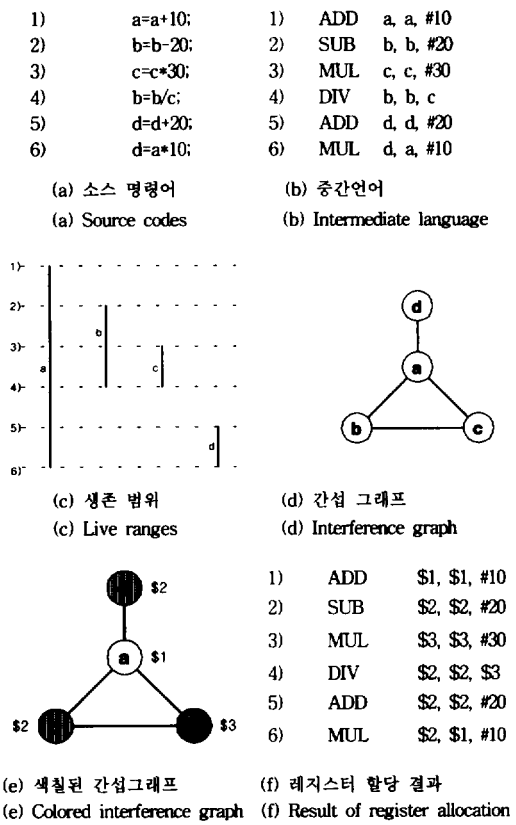
Chaitin 알고리듬의 첫 번째 단계에서는 광역 자료 흐름 분석(Data Flow Analysis)을 통하여 기본블록을 설정하고 제어 흐름 그래프를 구성한다. 그리고, 변수들의 생존 범위(live range)를 구한다. 다음 단계로 간섭그래프를 구성하는데, 변수들의 생존 범위가 서로 겹치게 되면 같은 색을 할당할 수 없기 때문에 이를 위한 정보를 얻기 위하여 간섭그래프를 구성한다. 간섭그래프에서 각 노드 즉, 변수들은 생존 범위를 나타내고 노드간 에지는 서로 간섭하고 있음을 나타낸다. 다음으로 단순화(simplification) 단계에서는 노드의 차수(degree)가 가용 레지스터 즉, 가용한 색(color)보다 적은 모든 노드들을 제거하면서 컬러링 스택(coloring stack)에 푸시한다. 만일, 노드의 차수가 가용 레지스터 수보다 크게 되면 한 노드가 대피(spill)를 위하여 선택되고 간섭그래프로부터 제거되며 대피될 노드로 표시된다. 마지막 색칠 단계에서는 컬러링 스택에 있는 노드들을 차례로 팝하면서 간섭 그래프를 재구성하고 이웃 노드들과는 다른 구별되는 레지스터를 할당한다. Chaitin의 전체적인 알고리듬은 (그림 5)와 같다.

```
Do {
  간섭 그래프 구성;
  Do {
     if (exist(degree of a node < k)) { //k : 가용 레지스터 수
       간섭 그래프로부터 노드와 노드에 연결된 모든 에지 제거;
       컬러링 스택에 노드 푸시;
       }
     else {
       노드의 차수가 k 보다 크거나 같으면 노드와 연결된;
       모든 에지를 간섭 그래프로부터 제거;
          대피될 노드로 표시;
          컬러링 스택에 노드 푸시;
          }
     }While (graph is not empty)
      대피될 노드로 표시된 노드가 있으면 대피 명령어 삽입;
  }While (spill is need)
  Do {
    컬러링 스택으로부터 노드를 팝;
    간섭 그래프에 노드와 연결된 에지 삽입;
    다른 노드와 구별되는 색 배정;
  }While (coloring stack not empty)
```

(그림 5) Chaitin의 레지스터 할당 알고리듬
(Fig. 5) Chaitin's register allocation algorithm

(그림 6)은 Chaitin의 전체 레지스터 할당 과정의 예를 보인다. 단, 가용 레지스터 수는 네 개이고 '$' 변수

를 레지스터라고 가정하며 왼쪽의 숫자는 각 명령에 해당하는 행 번호를 나타낸다. (그림 6)의 (d)를 보면 노드 b, c 그리고 d가 모두 노드 a와 간섭함을 보이고 Chaitin의 레지스터 할당 알고리듬으로 레지스터를 할당했을 때 (e)에서 세 개의 레지스터가 할당되었다. 가용 레지스터가 네 개였기 때문에 한 개의 남는 레지스터가 존재한다. 만일 가용 레지스터를 모두 사용한다면 (f)에서 자료 종속 관계를 줄일 수 있다. 많은 자료 종속 관계는 후에 명령어 스케줄시 많은 제약이 있기 때문에 중요한 문제가 된다. 본 논문에서는 이러한 점을 고려하여 기존의 레지스터 할당 알고리듬을 확장한다.



(그림 6) Chaitin의 레지스터 할당 과정 예
(Fig. 6) An example of Chaitin's register allocation processes

간섭그래프 구성 후에 자료 종속 그래프를 추가한 후 레지스터 배정(register assignment)시에 명령어 스케줄을 고려하여 배정하도록 새로운 레지스터 할당 알고리듬을 제안한다. DDG를 구성하는 이유는 각 노드들의 개별 종속 거리(individual dependence distance, IDD)와 노드 사이의 종속 거리 차를 구하기 위해서이다. 각 노드들의 개별 종속 거리는 다음의 정의에 의하여 구한다.

[정의 1] 한 노드의 종속 거리는 한 노드의 첫 정의(initial define)와 마지막 사용(last use)의 차를 말하며 다음과 같이 계산한다.

$$IDD_{nd} = lu_{nd} - id_{nd}$$

여기서, IDD_{nd}는 한 노드 $_{nd}$의 종속 거리를 나타내고 lu_{nd}는 한 노드 $_{nd}$의 마지막 사용 문장 번호를 의미하며 id_{nd}는 한 노드 $_{nd}$의 첫 정의의 문장 번호를 의미한다.

구해진 개별 종속 거리는 종속 거리가 큰 노드부터 레지스터를 배정하기 위하여 컬러링 스택에 개별 종속 거리가 가장 적은 노드부터 순서대로 푸시된다. 구성된 DDG를 토대로 노드들의 개별 종속 거리를 구한 후 노드들 사이의 종속 거리 차를 구한다. 이것은 레지스터 할당을 위해 컬러링 스택으로부터 팝된 노드와 가장 큰 종속 거리를 갖는 노드에 이미 할당된 색을 할당하기 위해서이다. 노드간 종속거리 차는 [정의 2]에 의하여 구한다.

[정의 2] 노드간 종속 거리는 레지스터 할당 대상이 되는 노드의 마지막 사용(last use) 문 번호와 다른 노드들의 마지막 사용 문 번호와의 차를 말하며 이들 중 최대값을 취하는데 아래와 같은 수식에 의하여 계산된다.

$$Diff\_lr_k = max(|(lu_k - lu_1)|, |(lu_k - lu_2)|, |(lu_k - lu_3)|, \ldots, |(lu_k - lu_n)|)$$

$Diff_lr_k$는 한 노드 k와 다른 노드들과의 종속 거리 차이값들을 취하면서 $lu_k \neq lu_n$이며 n은 정수이다. lu_k와 lu_n은 서로 간섭하지 않고 lu_k는 할당 대상이 되는 노드 k의 마지막 사용 문장 번호이며 lu_n은 레지스터의 할당 대상이 되는 노드를 제외한 다른 노드 n의 마지막 사용 문장 번호이다.

계산된 노드간 종속 거리 차이값들은 레지스터 할당시까지 유지되고 확장된 레지스터 할당 알고리듬에

의하여 레지스터 할당시 이들 중 최대값을 취하게 된다. 전반적인 AEC 알고리듬은 (그림 7)에서 보인다.

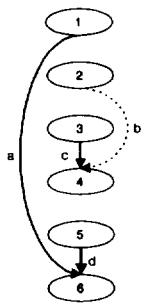
```
간섭 그래프 구성;
자료 종속 그래프 구성;
개별 종속 거리 크기순으로 컬러링 스택에 노드 푸시;
Repeat(        // 단계 1
    컬러링 스택으로부터 노드를 팝;
    간섭 그래프에 노드와 연결된 에지 삽입;
    가용한 색 배정;
    )Until(all available colors exhausted);
For(available colors are not exist and coloring stack is not empty){  // 단계 2
    assign threshold_value;
    counter++;
    컬러링 스택으로부터 노드를 팝;
    간섭 그래프에 노드와 연결된 에지 삽입;
    If(counter == threshold_value){
      Repeat{
            temp=0;
            For(counter > temp){
                  threshold_value를 갖는 노드의 색은 배정 대상에서 제외;
                  counter--;
                  }
            If(not interfere current popped node with the other node){
                  Diff_lr_k=max{|(lu_k-lu_1)|,|(lu_k-lu_2)|,...,|(lu_k-lu_n)|};
                  Diff_lr_k인 노드에 배정된 색과 같은색 배정;
                  If(assign fail){
                        exit and goto label FAIL;
                        }
                  }
            간섭 그래프에 노드와 연결된 에지 삽입;
            assign threshold_value;
            counter=counter+threshold_value;
            temp++;
      }Until(successfully assign a color);
    }
    Else if(not interfere current popped node with the other node){
      If(adj_relation){
            adj_relation을 갖는 노드에 배정된 색과 같은색 배정;
            }
      Else{
            Diff_lr_k=max{|(lu_k-lu_1)|,|(lu_k-lu_2)|,|(lu_k-lu_3)|,...,|(lu_k-lu_n)|};
            Diff_lr_k인 노드에 배정된 색과 같은색 배정;
            }
    간섭 그래프에 노드와 연결된 에지 삽입;
    }
FAIL:              // 대피 명령어 삽입
    대피 명령어 삽입;
    간섭 그래프 구성;
    자료 종속 그래프 구성;
}
```

(그림 7) AEC 알고리듬
(Fig. 7) AEC algorithm

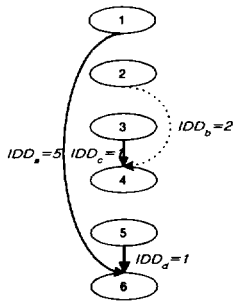
AEC 알고리듬에서 *threshold_value*는 임계값을 말하고 *Adj_relation*은 인접관계를 말한다. 'ADD $1, $2, $3'인 경우 목적지(target) 즉, $1과 오퍼런드(δperand) 즉, $2 또는 $3이 중간언어의 특성상 명령 바로 위나 아래에서 정의 또는 사용되었을 경우 이들은 서로 간섭하지 않지만 자료 종속 관계로 인하여 명령어 스케줄이 불가능하다. 따라서, $1과 $2, $3을 같은 레지스터를 배정하는 것이 좋은데 다음의 이유 때문에 임계값을 설정한다. 생성된 중간언어의 특성으로 인하여 앞서 설명한 명령어들이 연속적으로 존재하는 경우가 많이 발생하기 때문에 적정한 임계에서 같은 레지스터 배정을 중지하여야 한다. 임계값 설정의 또다른 이유는 서로 가까운 위치에 생존 범위가 짧은 임시 변수들이 존재하는 경우 (정의 2)에서 구한 최대값들이 같아지기 때문에 연속해서 같은 레지스터만 배정하게 된다. 이러한 이유로 일정한 임계에서 같은 레지스터 배정을 중지하여야 한다. 임계값 설정시 임계값이 너무 크면 레지스터 할당 후 자료 종속 관계가 오히려 증가할 가능성이 생기고, 임계값이 너무 작으면 고르게 레지스터를 배정하기 어렵게 된다. 이러한 점을 고려하여 벤치마크 프로그램들의 레지스터 할당된 중간언어 상에서 레지스터 분포와 자료 종속 관계를 실험한 결과 임계값이 5일 때 적정점을 이루었다. 따라서, 본 논문에서는 임계값을 5로 고정시켜 사용한다. 그러므로 알고리듬 상에서 임계값 5일 때 할당된 레지스터로부터 임계값 1에 도달할 때까지 할당된 레지스터는 이후 레지스터 할당 대상에서 제외한다. 인접관계는 한 명령 내에서 한 노드가 정의되고 다른 한 노드가 사용된 경우 또는, 한 노드가 사용되고 다른 한 노드가 정의된 경우를 말한다. 여기서 한 노드란 레지스터 할당을 위하여 컬러링 스택으로부터 팝된 노드를 말하고, 다른 한 노드란 이미 레지스터가 할당된 노드를 말한다. 예를 들면, 'ADD $1, $2, $3'인 문장의 경우 $1과 $2, $3은 서로 인접관계를 갖는다.

이 알고리듬은 두 단계로 구성되는데 첫 번째 단계에서는 컬러링 스택으로부터 할당 대상이 되는 노드들을 팝하면서 사용가능한 모든 레지스터를 할당한다. 이것은 기존의 방법에 비하여 남아도는 레지스터가 없도록 하고 충분히 레지스터를 이용할 수 있게 한다. 두번째 단계에서는 $Diff_lr_k$와 *threshold_value*에 의하여 고른 분포의 레지스터를 할당하도록 함으로써 자료들의 종속 관계를 최대한 줄이도록 한다.

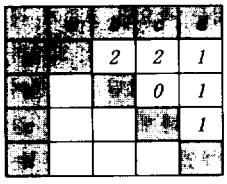
AEC 레지스터 할당 과정의 예는 간섭 그래프 구성까지 이미 (그림 6)의 (a), (b), (c), (d)에서 보였고 (그림 6)의 (a)에 대한 자료 종속 그래프, 노드들의 개별 종속 거리, 노드간 종속 거리 차 그리고 레지스터 할당 결과는 각각 (그림 8)의 (a), (b), (c), (d)와 같다.



(a) DDG 구성
(a) DDG construction



(b) DDG상의 개별 종속 거리
(b) Individual dependence distance in the DDG



(c) 노드간 종속 거리의 차
(c) Differences of dependence distance among nodes

1)	ADD	$1, $1, #10
2)	SUB	$2, $2, #20
3)	MUL	$3, $3, #30
4)	DIV	$2, $2, $3
5)	ADD	$4, $4, #20
6)	MUL	$4, $1, #10

(d) AEC 결과
(d) Results of AEC

(그림 8) (그림 6)의 예에 대한 AEC 과정
(Fig. 8) Processes of AEC for the example in (Fig. 6)

(그림 8)의 (a), (b)에서 노드는 각 명령 즉, 문장 번호를 나타내며 노드 사이의 가는 실선 에지는 참종속을, 점선 에지는 참종속, 반종속, 결과종속 세 가지 혼합 관계를 나타내고, 굵은 실선 에지는 결과종속을 나타낸다. 그리고, (b)는 DDG 상에서 개별 종속 거리를 보인다. 가용 레지스터가 네 개이기 때문에 (d)의 결과에서는 네 개의 레지스터가 모두 사용되었다. 만일, 색칠될 노드의 수가 가용 레지스터 수보다 많다면 (c)에서 구한 노드간 종속 거리의 차를 가지고 레지스터 배정의 우선순위를 정한다. 예를 들어, 노드 b가 레지스터 할당 대상이라면 a와 종속 거리 차이가 가장 크므로 a에 할당한 레지스터와 같은 레지스터를 할당한다. (그림 6)에서는 노드 d가 레지스터 $2에 할당되어 종속 관계로 인한 코드 스케줄의 제약이 있었다. 그러나, (그림 8)의 (d)에서는 노드 d가 레지스터 $4에 할당되어 종속 관계가 없기 때문에 코드 스케줄이 자유롭다. 이것은 코드 스케줄을 고려할 때 성능 향상에 중요한 요소가 된다.

3.2 적용 예

전체적인 본 논문의 흐름을 보이기 위하여 실제 소스 코드를 가지고 먼저 조건실행을 적용하고 본 논문에서 제안하는 알고리듬으로 레지스터를 할당한 후 코드 스케줄을 수행하는 과정을 논한다. 또한, 기존 Chaitin의 레지스터 할당 알고리듬과 비교하기 위하여 똑같이 조건실행을 적용하고 Chaitin의 레지스터 할당 알고리듬으로 레지스터를 할당한 후 코드 스케줄을 수행한다. 단, 중간언어는 3-주소 형태의 무한개 가상 레지스터를 가정하여 생성하며 가용 레지스터 수는 16이고 임계값은 5이다. 그리고 조건실행, 레지스터 할당, 코드 스케줄이 수행되는 프로그램 본문만을 예로 기술하고 의사 명령어(pseudo code) 등은 제외한다. 프로그램 소스는 (그림 9)이고 생성된 중간언어에 대한 제어 흐름 그래프는 (그림 10)에서 보인다.

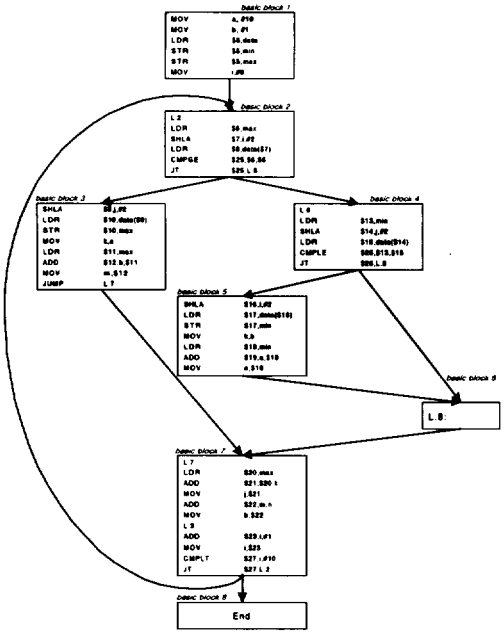
```
// find maximum, minimum value
int max, min;
int data[10] = {8, 3, 2, 4, 9, 6, 7, 1, 5, 10};
main()
{
    int i,j,k,m,n,a=10,b=1;
    max = min = data[0];
    for(i=0;i<10;i++){
        if(max < data[i]){
            max = data[i];
            k = a;
            m = b + max;
            }
        else if(min > data[i]){
            min = data[i];
            k = b;
            n = a + min;
            }
        j= max + k;
        b= m + n;
    }
}
```

(그림 9) 프로그램 소스
(Fig. 9) Program source

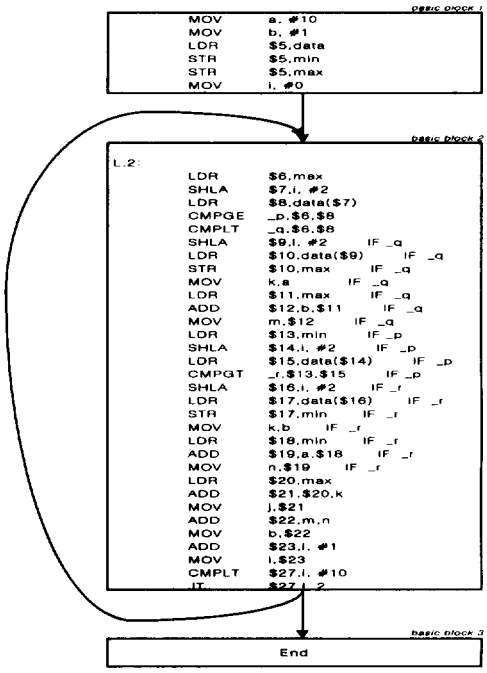
(그림 10)에서 '$' 변수는 가상 레지스터를 나타낸다. 제어 흐름 그래프에서 기본블록 2의 조건분기가 참이면 수행되는 경로는 기본블록 4, 5, 6이고 거짓이면 기본블록 3을 수행한다. 또한, 기본블록 4도 조건분기가 참이면 기본블록 6을 수행하고 거짓이면 기본블록 5를 수행한 후 기본블록 6을 수행한다. 실제 (그림 10)에 대하여 if 변환 후의 제어 흐름 그래프는 (그림 11)과 같다. (그림 11)에서 조건정의 명령 CMPGT _r, $13, $15 이후에 조건기술 _r로 보호된 명령들만 존재하는 이유는 조건이 거짓일 때 판별하면 되고 참일 때에는 조건기술이 없어도 제어 에지의 흐름상 이후 명령들은 수행을 계속하기 때문이다.

(그림 10)에 비하여 (그림 11)은 두 개의 조건분기가 줄어들었고 각 기본블록이 한 개의 단일블록으로 합쳐지면서 명령들이 정렬되었다. 기본블록 내의 명령 수가 많아지면서 코드 스케줄 기회가 증가하기 때문에 성능 향상이 기대된다.

DDG 구성, 개별 종속 거리, 노드간 종속 거리의 차를 구한 예는 이미 (그림 8)에서 보였고 (그림 12)와 (그림 13)은 각각 Chaitin의 레지스터 할당 알고리듬, AEC 알고리듬을 적용하여 레지스터 할당한 결과를 보인다.



(그림 10) (그림 9)의 예에 대한 제어 흐름 그래프
(Fig. 10) Control flow graph for the example in (Fig. 9)



(그림 11) (그림 9)의 예에 대한 if 변환후의 제어 흐름 그래프
(Fig. 11) Control flow graph after if-conversion for the example in (Fig. 9)

```
        MOV     $6,#10
        MOV     $5,#1
        LDR     $4,data
        STR     $4,min
        STR     $4,max
        MOV     $4,#0
L.2:
        LDR     $8,max
        SHLA    $7,$4,#2
        LDR     $7,data($7)
        CMPGE   _p,$8,$7
        CMPLT   _q,$8,$7
        SHLA    $2,$4,#2        IF _q
        LDR     $2,data($2)     IF _q
        STR     $2,max          IF _q
        MOV     $3,$6           IF _q
        LDR     $2,max          IF _q
        ADD     $2,$5,$2        IF _q
        MOV     $2,$2           IF _q
        LDR     $6,min          IF _p
        SHLA    $7,$4,#2        IF _p
        LDR     $7,data($7)     IF _p
        CMPGT   _r,$8,$7        IF _p
        SHLA    $1,$4,#2        IF _r
        LDR     $1,data($1)     IF _r
        STR     $1,min          IF _r
        MOV     $3,$5           IF _r
        LDR     $1,min          IF _r
        ADD     $1,$6,$1        IF _r
        MOV     $1,$1           IF _r
        LDR     $5,max
        ADD     $5,$5,$3
        MOV     j,$5
        ADD     $5,$2,$1
        MOV     $5,$5
        ADD     $4,$4,#1
        MOV     $4,$4
        CMPLT   $7,$4,#10
        JT      $7,L.2
L.1:
```

(그림 12) (그림 9) 예에 대한 Chaitin의 레지스터 할당 결과
(Fig. 12) Result of Chaitin's register allocation for the example in (Fig. 9)

```
        MOV     $2, #10
        MOV     $3, #1
        LDR     $9,data
        STR     $9,min
        STR     $9,max
        MOV     $1, #0
L.2:
        LDR     $8,max
        SHLA    $13,$1, #2
        LDR     $9,data($13)
        CMPGE   _p,$8,$9
        CMPLT   _q,$8,$9
        SHLA    $11,$1, #2      IF _q
        LDR     $5,data($11)    IF _q
        STR     $5,max      IF _q
        MOV     $5,$2       IF _q
        LDR     $10,max     IF _q
        ADD     $4,$3,$10       IF _q
        MOV     $4,$4       IF _q
        LDR     $7,min      IF _p
        SHLA    $16,$1, #2      IF _p
        LDR     $15,data($16)       IF _p
        CMPGT   _r,$7,$15       IF _p
        SHLA    $13,$1, #2      IF _r
        LDR     $12,data($13)       IF _r
        STR     $12,min     IF _r
        MOV     $5,$3       IF _r
        LDR     $11,min     IF _r
        ADD     $10,$2,$11      IF _r
        MOV     $6,$10      IF _r
        LDR     $9,max
        ADD     $9,$9,$5
        MOV     i,$9
        ADD     $3,$4,$6
        MOV     $3,$3
        ADD     $1,$1, #1
        MOV     $1,$1
        CMPLT   $9,$1, #10
        JT      $9,L.2
L.1:
```

(그림 13) (그림 9) 예에 대한 AEC의 레지스터 할당 결과

(Fig. 13) Result of AEC's register allocation for the example in (Fig. 9)

레지스터가 할당된 결과를 보면 기존 레지스터 할당을 적용했을 때는 가용 레지스터가 16개임에도 불구하고 8개의 레지스터만 사용됨으로써 남는 레지스터가 존재하였다. 그러나, 제안한 방법을 적용했을 때는 남는 레지스터는 존재하지 않고 16개의 레지스터가 모두 사용되었다. 이는 커진 기본블록 내에서 레지스터를 되도록 많이 이용하고 레지스터가 고르게 분포하도록 배정함으로써 코드 스케줄 향상을 위한 기회를 증가시키기 위해서이다.

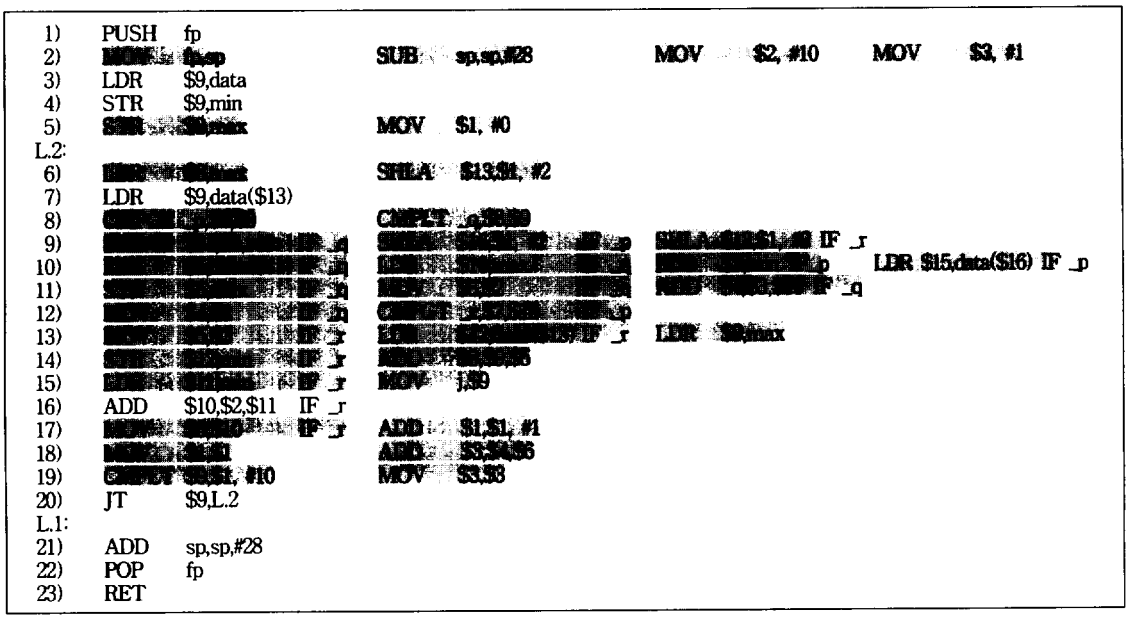
다음은 코드 스케줄을 수행하는데 코드 스케줄은 레지스터가 할당된 명령어를 가지고 트레이스(trace)를 분석하여 병렬 처리가 가능하도록 기본블록 내에서 명령어 이동을 수행한다. 그리고, 병렬 처리를 위한 명령의 이슈 수는 네 개이다. 코드 스케줄된 결과를 보면 Chaitin의 레지스터 할당된 결과는 (그림 14)와 같고 AEC의 레지스터 할당된 결과는 (그림 15)와 같다.

(그림 14) 왼쪽 열의 숫자들은 실행 클럭 사이클을 나타내며 음영으로 칠해진 명령들은 스케줄된 명령어들이고 병렬 처리가 가능한 명령들은 같은 행에 배치되었다. 예를 들면, 클럭 사이클 2에서 네 개의 명령이 동시에 실행가능하고 클럭 사이클 10에서는 세 개의 명령이 동시에 병렬 처리가 가능함을 나타낸다. 실행 클럭 사이클은 분기나 루프(loop)에 의하여 증가되지만 한번의

```
1)      PUSH    fp
2)      MOV     fp,sp                   SUB     sp,sp,#28               MOV     $5,#1
3)      LDR     $4,data
4)      STR     $4,min
5)      STR     $4,max                  MOV     $4,#0
L.2:
6)      LDR     $8,max                  SHLA    $7,$4,#2
7)      LDR     $7,data($7)
8)      CMPGE   _p,$8,$7
9)      SHLA    $13,$1,#2       IF _q                   IF _p
10)     LDR     $12,data($13)   IF _q                   IF _p   LDR     $7,data($7)     IF _p
11)     STR     $12,min     IF _q       MOV     $5,$3   IF _q
12)     LDR     $2,max      IF _q
13)     ADD     $2,$5,$2    IF _q
14)     MOV     $11,min     IF _q       CMPGT   _r,$7,$15       IF _p
15)     SHLA    $13,$1,#2   IF _r       MOV     $6,$10      IF _r
16)     LDR     $12,data($13)   IF _r   LDR     $9,max
17)     STR     $1,min      IF _r
18)     LDR     $9,max      IF _r       MOV     i,$9                    ADD     $3,$4,$6
19)     ADD     $1,$6,$1    IF _r
20)     MOV     $3,$3       IF _r       ADD     $1,$1,#1
21)     MOV     $1,$1                   CMPLT   $9,$1,#10
22)     CMPLT   $9,$1,#10               MOV     $9,$5
23)     JT      $7,L.2
L.1:
24)     ADD     sp,sp,#28
25)     POP     fp
26)     RET
```

(그림 14) Chaitin 방식의 코드 스케줄 결과

(Fig. 14) Code schedule results for the scheme of Chaitin



(그림 15) AEC 방식의 코드 스케줄 결과
(Fig. 15) Code schedule results for the scheme of AEC

실행에서 26 클럭 사이클이 소요되었음을 볼 수 있다.

코드 스케줄 결과를 보면 (그림 15)는 (그림 14)에 비하여 자료 종속 관계가 적기 때문에 보다 많은 명령어 이동의 기회가 존재해 병렬 처리가 가능하도록 많은 명령들이 스케줄되었다. 예를 들면, (그림 14)의 클럭 사이클 9~13을 보면 레지스터 $2의 자료 종속 관계로 인하여 더 이상의 코드 스케줄을 수행하지 못한다. 그러나, (그림 15)의 같은 클럭 사이클을 보면 레지스터가 골고루 할당되었기 때문에 자료 종속 관계가 적은 관계로 보다 많은 코드 스케줄이 가능하였다. 명령어 전반적으로도 자료 종속 관계가 기존의 방법으로 레지스터를 할당하였을 때 보다 적기 때문에 많은 코드 스케줄이 가능하였다. 이 결과로 볼 때 제안된 레지스터 할당 알고리듬이 조건실행을 적용하여 기본블록의 크기를 크게 하였을 때 기존의 레지스터 할당 알고리듬에 비하여 성능 향상의 가능성이 많이 존재함을 예측할 수 있다. 클럭 사이클을 보면 (그림 15)도 마찬가지로 분기문이나 루프에 의하여 실제는 여러 실행 사이클이 걸리지만 결과상으로 한 번 실행 시 23 클럭 사이클이 소요되므로 루프 등의 존재 시 (그림 14)에 비하여 더욱 많은 클럭 사이클의 이득을 얻을 수 있음을 알 수 있다.

4. 실험 방법 및 성능 측정 결과

본 논문에서는 조건실행을 적용한 중간언어에 대하여 제안한 적극적인 레지스터 할당 알고리듬이 타당성이 있는지를 검증하고자 한다. 먼저 구현한 최적화 컴파일러와 개발한 시뮬레이터(simulator)에 대하여 간단히 소개한 후 다양한 실험을 통하여 제안한 방법의 타당성을 검증한다.

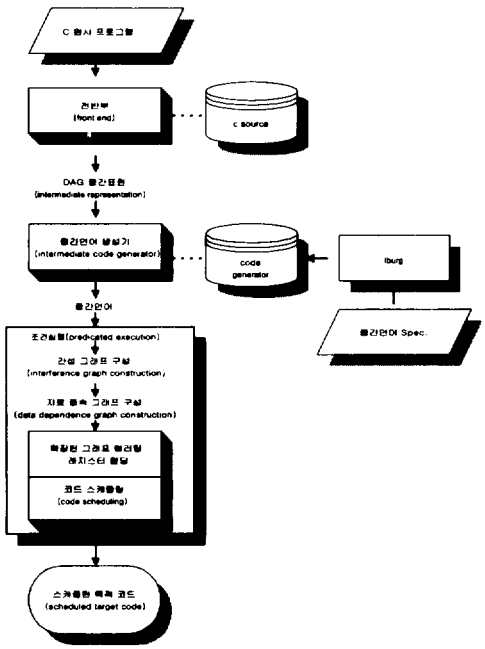
4.1 최적화 컴파일러와 시뮬레이터

본 논문에서 개발한 최적화 컴파일러는 retargetable 컴파일러인 lcc의 전반부(front-end)를 이용하여 중간언어를 생성한다[13]. Lcc는 AT&T Bell 연구소의 C. W. Fraser와 Princeton 대학의 David Hanson에 의하여 개발되었다. Lcc의 특성으로 인하여 후반부(back-end)에서 다양한 최적화 기법을 적용하기가 어렵기 때문에 후반부에서는 본 논문에서 제안하고 구현한 최적화기로 대치하여 목적 코드를 생성한다. (그림 16)은 전반적인 최적화 컴파일러의 흐름을 보인다.

Lcc의 전반부는 DAG(Directed Acyclic Graph) 형태의 중간표현을 생성한다. 이는 전반부와 후반부의 인터페이스(interface) 역할을 해 주는데 공유를 위한

자료구조, 19개의 인터페이스 함수, DAG로 이루어진 36개의 연산자들로 구성된다. 이 DAG 표현을 입력으로 받아 코드생성기 자동생성기(code generator-generator)인 lburg[14]를 이용하여 중간언어를 생성한다. 3-주소 형태의 생성된 중간언어는 무한개의 가상 레지스터와 가상 머신에 대하여 생성하며 다양한 명령어 세트를 지원한다.

생성된 중간언어상에서 최적화를 수행하는데 조건분기문을 줄이고 기본블록을 확장하기 위한 조건실행을 수행하고 간섭그래프 구성, 자료 종속 그래프 구성 단계를 거쳐 본 논문에서 제안한 AEC 알고리듬을 적용하여 레지스터를 할당하고 profiling 정보를 이용하여 트레이스를 분석한 후 코드 스케줄을 수행한다. 코드 스케줄이 레지스터 할당 후에 이루어지기 때문에 레지스터 할당시 주의깊게 레지스터를 할당하지 않으면 자료 종속 관계 증가로 인하여 스케줄을 제한할 수 있다. 본 논문에서 제안한 AEC 알고리듬은 이 점을 고려하여 설계하였다.



(그림 16) 전반적인 최적화 컴파일러 구성도
(Fig. 16) Diagram for whole optimizing compiler

시뮬레이터는 GNU의 FLEX, BISON[15]을 이용하여 구현하였고 수행 후 메모리 내용, 수행된 클럭수 및 명령 트레이스, 사용된 명령들의 정적/동적 빈도수, 분기명령의 예측률, profiling 정보 등을 생성한다. 본 논문에서 실험을 위하여 사용한 시뮬레이터의 파라미터(parameter)는 프로그램 수행 결과를 추적하기 위한 메모리 내용과 코드 스케줄을 적용하기 위해 얻는 정보인 profiling 정보, 명령 트레이스 그리고 성능 개선이 이루어졌는지를 확인하기 위한 실행 클럭수 측정이다.

4.2 성능 측정

성능 측정은 입·출력 라이브러리의 개발 중으로 인하여 SPEC 벤치마크 프로그램과 같은 큰 프로그램은 아직 제한이 있다. 따라서, 일반적으로 많이 사용하고 있는 벤치마크 프로그램을 가지고 실험에 사용하였다. 그리고, 제안된 알고리듬의 타당성을 검증하기 위하여 기존에 많이 응용되고 있는 Chaitin의 레지스터 할당 알고리듬을 적용한 후 결과를 비교하였다. 실험을 위한 벤치마크 프로그램은 <표 1>과 같다.

〈표 1〉 성능 측정 프로그램
〈Table 1〉 Benchmark programs

Pattern(patt)	패턴 비교
Fibonacci(fibo)	fibonacci 수열
Maxmin(mami)	최대, 최소값
Difference(diff)	두 배열의 차

먼저, 실험을 위하여 아래와 같은 방법으로 실험하였다. 아래에서 Not_pred는 조건실행을 하지 않은 경우, Pred를 조건 실행한 경우, Chaitin은 기존 Chaitin의 레지스터 할당 알고리듬을 적용한 경우, AEC를 제안한 레지스터 할당 알고리듬을 적용한 경우, Schedule은 코드 스케줄을 말하고 각 방법에서 '+' 기호의 의미는 같은 단계에서 실험되었음을 의미한다. 그리고 각 벤치마크 프로그램에 대하여 레지스터 할당 방법만 다르고, 조건실행과 코드 스케줄은 동일한 방법으로 실험하였다. 그리고 코드 스케줄은 기본블록 내에서 이루어지고 가용 레지스터의 수는 16이며 임계값은 5이다. 또한, 명령의 이슈 수는 4이다.

실험 방법)

I : Not_pred + Chaitin + Schedule

II : Pred + Chaitin + Schedule
III : Not_pred + AEC + Schedule
IV : Pred + AEC + Schedule

각 실험 방법으로 실험한 이유를 부연 설명하면 실험 방법 I과 III은 조건실행을 하지 않고 레지스터 할당과 코드 스케줄을 수행했을 때를 비교하고 실험 방법 II와 IV는 조건 실행을 적용하고 레지스터 할당과 코드 스케줄을 수행했을 때를 비교하기 위해서이다. 또한, 각 실험 방법들의 성능차와 I, II의 성능차 그리고 III, IV의 성능차간의 비교를 하기 위해서이다. 실행 클럭 사이클 측정 후의 실험 결과는 <표 2>와 같다.

방법 I과 III의 결과를 비교하면 벤치마크 프로그램 patt와 fibo에서 제안한 레지스터 할당 알고리듬을 적용했을 때가 약 5% 정도 기존 레지스터 할당 알고리듬을 적용했을 때보다 개선된 결과를 얻었다. 그러나 벤치마크 프로그램 mami와 diff는 자료 종속 관계가 기존 방법에 비하여 줄었지만 명령수가 적은 기본블록들이 많이 존재하기 때문에 기본블록내의 스케줄 제약으로 인하여 기존의 방법과는 차이가 없음을 볼 수 있다.

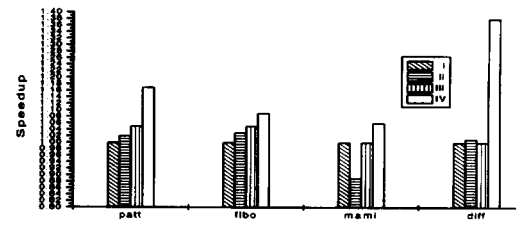
〈표 2〉 실행 클럭 사이클 측정 결과
〈Tabel 2〉 Estimated results of execution clock cycles

Experimental Methods	Benchmark			
	[illegible]	[illegible]	mami	diff
I	256	363	177	186
II	251	353	198	185
III	245	345	177	186
IV	219	335	168	135

방법 II와 IV에서는 네 개의 벤치마크 프로그램 모두 기존 방법에 비하여 성능 개선이 이루어졌고 특히, 벤치마크 프로그램 diff의 경우에는 기존 방법에 비하여 37% 정도의 성능향상이 있었다. 이는 기존의 레지스터 할당 방법이 되도록 적게 레지스터를 사용하도록 하기 때문에 합쳐진 기본블록내의 명령이 많아질수록 레지스터 할당 후 자료 종속 관계의 증가를 유발하기 때문이다. 따라서, 코드 스케줄의 제약 때문에 성능이 제안한 레지스터 할당 알고리듬에 비하여 많이 떨어졌다. 그러나, 제안한 레지스터 할당 알고리듬은 가용 레지스터를 전부 사용하고 골고루 레지스터를 배정하도록 함으로써 자료 종속 관계 감소로 인한 코드 스케줄 가능성 증가로 인하여 많은 성능 개선이 이루어졌다. 또한, 방법 III과 IV는 III에 비하여 약 15%로 IV가 성능이 우수하였다. 이것은 본 논문에서 제안한 레지스터 할당 알고리듬이 조건실행과 같이 사용될 때 더욱 효과적인 알고리듬임을 증명해 준다.

전체적으로 비교하면 제안된 레지스터 할당 알고리듬이 기존 Chaitin의 레지스터 할당 알고리듬에 비하여 전반적으로 성능 향상이 있었음을 볼 수 있고 특히, 조건실행을 적용한 후에는 더욱 높은 성능 향상을 얻었다. 전체적으로 기존의 레지스터 할당 알고리듬에 비하여 약 19%의 성능 개선을 얻음으로써 본 논문에서 제안한 레지스터 할당 알고리듬이 타당함을 알 수 있다.

다음 (그림 17)은 <표 2>의 결과에 대한 각 방법들의 성능차를 보인다. 단, 방법 I의 성능차를 1로 기준하였을 때 계산된 결과이다.

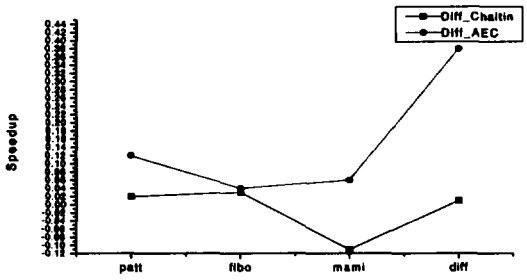


(그림 17) 각 방법들의 성능차
(Fig. 17) Speedup of each methods

(그림 17)에서 전체적으로 방법 IV가 성능이 우수함을 보임으로써 방법 I, II, III 보다 방법 IV가 타당한 방법임을 증명해 준다. 특히, 조건실행과 기존의 레지스터 할당 방법을 결합하여 성능을 측정했을 때는 벤치마크 프로그램 mami에서 오히려 조건실행을 적용하지 않았을 때보다 11% 정도 성능이 저하되었다. 이는 조건실행시 커진 기본블록 내에서 자료 종속 관계의 증가로 인한 코드 스케줄의 제약 때문이다. 따라서, 코드 스케줄의 효율을 높인 EPIC 아키텍쳐에는 기존 레지스터 할당 알고리듬이 부적합할 수 있다는 것을 단적으로 보인다.

마지막으로, 방법 I, II와 방법 III, IV의 성능차를 비교해 봄으로써 전체적으로 기존의 레지스터 할당 방법을 사용했을 때와 제안된 레지스터 할당 방법을 사용했을 때 어느 방법이 적절한 방법인지를 확인한다. 그 결과는 (그림 18)과 같다. (그림 18)에서 Diff_Chaitin은

방법 I과 II의 성능차이고 Diff_AEC는 방법 III과 IV의 성능차이다.



(그림 18) Diff_Chaitin과 Diff_AEC 성능차 비교
(Fig. 18) Comparison of speedup Diff_Chaitin and Diff_AEC

전체적으로 기존의 레지스터 할당 방법을 사용했을 때와 제안된 레지스터 할당 방법을 사용했을 때를 비교하면 후자인 Diff_AEC가 훨씬 유리함을 확인할 수 있다. 특히, 기존 레지스터 할당 방법은 오히려 성능 감소를 초래하는 경우가 발생하기 때문에 제안된 레지스터 할당 방법을 사용하는 것이 보다 안정적임을 확인해 준다.

5. 결론 및 연구 과제

본 논문에서는 제안한 EPIC 아키텍처를 위한 적극적 레지스터 할당 알고리듬을 제안하였다. 조건실행을 수행하게 되면 조건분기가 제거되고 보호된 기본블록들을 단일블록으로 묶을 수 있다. 따라서, 기본블록 내에 병렬 처리될 명령의 수는 증가하지만 레지스터를 할당할때 기존의 레지스터 할당 방법은 되도록 적은 수의 레지스터를 사용하기 때문에 자료 종속 관계 증가로 인한 코드 스케줄 제약으로 오히려 성능을 감소시킬 수 있다. 실험 결과 기존의 레지스터 할당 방법은 일부 벤치마크 프로그램에서 조건실행을 적용했을 때가 오히려 성능이 떨어짐을 알 수 있었고 전체적으로도 기존의 방법을 적용했을 때는 성능차가 그다지 크지 않았다. 그러나, 제안된 레지스터 할당 방법은 가용한 레지스터를 전부 이용하도록 하고 고른 레지스터 분포를 갖게 하기 때문에 레지스터 할당 후 자료 종속 관계가 줄어 코드 스케줄 가능성을 증가시켰고 조건실행을 적용 후 기본블록의 크기가 커질수록 더욱 병렬처리 능력이 향상됨을 확인하였다. 제안한 레지스터 할당 알고리듬을 조건실행과 결합했을 때가 조건실행과 결합하지 않았을 때에 비하여 평균 15%로 성능이 우수하였고 기존 레지스터 할당 알고리듬과 제안된 레지스터 할당 알고리듬 각각에 대하여 조건실행과 결합하여 실험했을 때, 제안된 레지스터 할당 알고리듬과 조건실행을 결합했을 때가 기존의 레지스터 할당 알고리듬과 조건실행을 결합했을 때에 비하여 평균 19%의 성능 향상을 보임으로써 제안된 레지스터 할당 알고리듬의 타당성을 검증하였다. 레지스터 할당시 최근의 ILP 프로세서들은 레지스터의 개수가 증가하는 추세에 있기 때문에 대피 명령어가 발생할 확률은 적지만 대피 명령어가 발생했을 때 이의 효과적인 처리를 고려해야 하는데 제안된 알고리듬은 대피 명령어 발생 시에 기존의 방법에 비하여 더욱 효과적일 수 있다. 즉, 생존 범위가 긴 변수를 대피시킴으로써 서로 간섭하고 있던 변수들은 제안된 알고리듬을 사용할 경우 더 많은 레지스터들을 골고루 할당할 수 있는 기회가 증가한다.

향후 연구 과제로는 보다 완벽한 최적화 컴파일러를 완성하여 SPEC 벤치마크 프로그램들을 가지고 타당성을 검증해 보고자하며 조건실행을 보다 완벽하게 구현하고 이를 적용하였을 때 레지스터의 수를 절약할 수 있다는 점을 제안된 레지스터 할당 알고리듬에서 고려해 보고자 한다.

참 고 문 헌

[1] Wen-mei W. Hwu, Richard E. Hank, David M. Gallagher, Scott A. Mahlke, Daniel M. Lavery, Grant E. Haab, John C. Gyllenhaal, David I. August, "*Compiler Technology for Future Microprocessors*," Proceedings of the IEEE, Vol.83, No.12, pp.1625-1640, Dec. 1995.

[2] J. R. Ellis, Bulldog : A Compiler for VLIW Architecture, The MIT Press, 1986.

[3] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Wen-mei W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," Proc. of the 25th International Symposium on Computer Architecture, July, 1998.

[4] Richard Johnson and Michael Schlansker, "Analysis Techniques for Predicated Code," In Proc. of the 29th Annual International Symposium on Microarchitecture, Dec. 1996.

[5] David M. Gillies, Dz-ching Roy Ju, Richard Johnson, Michael Schlansker, "Global Predicate Analysis and its Application to Register Allocation," MICRO-29, 1996.

[6] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling : A model for compiler-controlled speculative execution," ACM Transactions on Computer Systems, 11(4), Nov. 1993.

[7] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," Proc. of the ACM Symposium on Compiler Construction, pp.98-105, June, 1982.

[8] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors, and W. W. Hwu, "The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual," Technical Report IMPACT-98-04, IMPACT, Univ. of Illinois, Urbana, IL, Feb. 1998.

[9] Preston Briggs, "Register Allocation via Graph Coloring," Ph. D Thesis, RICE Univ. April, 1992.

[10] S. Pinter, "Register Allocation with Instruction Scheduling," In Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation, pp.248-257, 1993.

[11] Pohua P. Chang, Daniel M. Lavery, Scott A. Mahlke, William Y. Chen and Wen-mei W. Hwu, "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," IEEE Transactions on Computers, Vol.44, No.3, pp.353-370, March 1994.

[12] Joon-Kee Choi, Sang-Jeong Lee, "An Extended Graph Coloring Register Allocation Scheme to Enhance the Efficiency of Code Scheduling," Proc. of the International Technical Conference on Circuits/Systems, Computers and Communications, Vol.II, pp.1243-1246, 1998.

[13] C. Fraser and D. Hanson, A Retargetable C Compiler : Design and Implementation, The Benjamin/Cummings Publishing Company, 1995.

[14] C. Fraser, D. Hanson, Todd A. Proebsting, "Engineering Efficient Code Generators using Tree Matching and Dynamic Programming," Research Report CS-TR-386-92, Univ. of Princeton, Dept. of Computer Science, Aug. 1992.

[15] Free Software Foundation, GNU FLEX, BISON Manual, 1995.

[16] 유병강, 이상정, "ILP 프로세서를 위한 조건실행 지원 알고리듬", 한국정보처리학회 논문지, 제5권 제1호, pp.202~214, 1998.

최 준 기

e-mail : jkchoi@archi-cse.sch.ac.kr

1993년 2월 순천향대학교 전산학과(공학사)

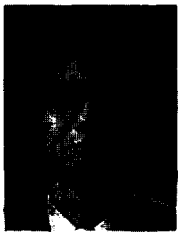
1995년 2월 순천향대학교 전산학과(공학석사)

1999년 2월 순천향대학교 전산학과(공학박사)

1998년 1월~현재 CJ 정보통신 기술고문

1998년 3월~현재 신성대학 겸임교수

관심분야 : 최적화 컴파일러, EPIC 아키텍쳐, 레지스터 할당



이 상 정

e-mail : sjlee@asan.sch.ac.kr

1983년 한양대학교 전자공학과(공학사)

1985년 2월 한양대학교 대학원 전자공학과(공학석사)

1985년 8월 한양대학교 대학원 전자공학과(공학박사)

1988년~현재 순천향대학교 컴퓨터학부 교수

관심분야 : 프로세서 설계, 최적화 컴파일러 설계, 마이크로프로세서 응용