韓國 OA學會 論文誌
第5卷 第3號, 2000. 9.

2000-5-3-1-1

선코드 스케줄링의 최적화를 위한 연구

최 준 기*

A Study for an Optimization of Prepass Code Scheduling

Joon-Kee Choi*

요 약

선코드 스케줄링은 코드 스케줄링을 먼저 수행함으로써 자료 종속 관계가 복잡해지고, 레지스터를 할당할 때 간섭그래프가 복잡해져 레지스터 할당을 어렵게 만들 수 있다.

본 논문에서는 이를 개선하기 위하여 2-단계 컬러링 기법을 제안한다. 단계 1에서 생존 거리가 큰 변수들에 레지스터 배정, 단계 2에서 나머지 변수들에 레지스터를 할당함으로써 레지스터 할당 소요 비용을 최소화한다. 실험 결과 기존의 방법에 비해 제안한 방법이 효율적임을 검증하였다.

Abstract

Prepass code scheduling(code scheduling before register allocation), the register lifetimes may be lengthened, which may increase the amount of data dependence relations. So, it makes difficult to allocate the registers because of complex interference graph.

In this paper, to improve that defect, propose an 2-phase coloring method. At first phase-1 assign the registers to variables which have long live ranges. Secondly, phase-2 allocate the registers to remained variables to minimize the register allocation cost. Experimental results shown that proposed method is more efficient scheme than Chaitin's scheme when prepass code scheduling.

* 인덕대학 여성정보행정과 전임강사

I. 서 론

최근 컴파일러의 최적화 기법들이 많이 연구되고 있다[1,2]. 이 중 코드 스케줄링(code scheduling) 기법과 레지스터 할당(register allocation)을 결합한 연구들이 최적화의 중요한 부분을 차지하기 때문에 특히 집중적인 연구가 필요한 실정이다.

코드 스케줄링은 프로그램의 실행 시간을 최소화하기 위하여 명령어들을 재정렬하기 위한 기법이다. 최근의 마이크로프로세서들이 이슈(issue)율을 향상시키기 위한 노력들을 하고 있는 것으로 볼 때 최적화 부분에서 매우 중요한 비중을 차지한다. 코드 스케줄링 방법으로는 국소적으로 수행하는 방법(local scheduling)과 광역적으로 수행하는 방법(global scheduling)이 있다. 기본블록내에서 스케줄을 수행하는 국소적 스케줄링 방법에는 리스트 스케줄링(list scheduling)[3]이 있고 프로그램의 전체 단위로 스케줄을 수행하는 광역적 스케줄링 방법은 추적 스케줄링(trace scheduling)[3], 여과 스케줄링(percolation scheduling)[4] 기법 등이 있다.

레지스터 할당도 또한 최적화 과정에서 꼭 필요한 단계이다. 프로세서 설계 기술이 발전하면서 물리적 레지스터(physical register)의 개수도 증가하는 추세이나 프로그램상에 존재하는 많은 변수들을 다 수용하기에는 부족한 상태이다. 많은 변수들이 레지스터에 존재하는 것은 빠른 기억장치를 이용하기 위함이며 만일 메모리로의 대피(spill)나 메모리로부터 변수들을 가져오게 되면 많은 실행 속도의 손실을 가져온다. 따라서, 최근의 로드(load)/스토어(store) 아키텍쳐에서는 중요한 문제로 대두되고 있다. 레지스터 할당 방법은 Chaitin, Briggs 그리고 Chow 등이 제안을 했지만 모두 Chaitin이 제안한 방법을 기초로 하고 있다[6,7,8]. Chaitin은 NP-완전(complete) 문제로 알려진 그래프 컬러링 알고리듬을 레지스터 할당에 적용하였다. 최적화 컴파일러에서의 레지스터 할당은 꼭 필요한 경우 변수들을 메모리로 대피시키기 때문에 NP-완전 문제에 해당되지는 않는다.

중요한 과정들인 코드 스케줄링과 레지스터 할당을 결합하게되면 몇 가지 문제가 발생한다[9]. 만일 코드 스케줄링 다음으로 레지스터 할당을 수행하게(선코드 스케줄링, prepass code scheduling)되면 코드 재정렬로 인한 변수들의 생존범위(live range)가 길어지기 때문에 간섭 그래프(interference graph)가 복잡해지고 따라서 레지스터 할당을 제약한다. 그리고, 레지스터 할당 다음으로 코드 스케줄링을 수행하게(후코드 스케줄링, postpass code scheduling)되면 자료 종속 그래프(data dependence graph)가 복잡해짐으로써 코드 스케줄링을 제약한다. 이것은 레지스터 할당 알고리즘이 최소의 레지스터로 많은 변수들을 수용하기 때문에 발생하는 문제이다.

본 논문에서는 이러한 문제점들을 개선하고자 선코드 스케줄링의 최적화를 위한 새로운 방법을 제안한다. 즉, 레지스터 할당 전에 코드 스케줄링으로 인한 길어진 생존범위를 갖는 변수들을 미리 레지스터에 배정(assign)한 다음 나머지 변수들을 가지고 레지스터 할당을 수행한다. 긴 생존범위를 갖는 변수들을 갖는 변수집합과 나머지 변수집합을 구하여 레지스터 배정과 레지스터 할당을 하는 2-단계 컬러링(2-phase coloring) 방법을 적용한다.

제안한 방법은 선코드 스케줄링에서의 레지스터 할당시 많은 시간상의 비용을 필요로 하는 간섭 그래프를 단순화시킴으로써 효율적인 레지스터 할당을 수행하도록 한다.

II. 선코드 스케줄링

선코드 스케줄링은 코드 스케줄링을 레지스터 할당 전에 수행함으로써 레지스터 할당보다는 코드 스케줄링에 더 비중을 둔 방법이다. 명령어들을 재정렬하기 위한 코드 스케줄링은 효율정도에 따라 동시에 실행할 수 있는 명령수가 증가한다. 즉, 8-이슈인 ILP(Instruction Level Parallelism) 프로세서라면 명령어들의 실행을 위해 동시에 8개의 명령을 이슈할 수 있다는 의미로 코드 스케줄링의 최대 효과는 8개의 명령을 동시에 이슈

할 수 있도록 재정렬하는 것이다. 코드 스케줄링의 이 목적을 실현하기 위해서는 다음 단계 즉, 레지스터 할당에서 손해를 보아야한다. 이것은 상호 상충(conflicts) 관계에 있기 때문이다. 다음 예제 그림으로 확인할 수 있다.

```
1) LDR   $12, wordct      1
2) ADD   $13, $12, #1     2
3) STR   $13, wordct      3
4) ADD   $14, token, #1   3
5) MOV   token, $14       4
```

그림 1. 중간코드의 일부
Fig 1. A part of an intermediate code

그림 1은 3-주소 형태(3-address format)의 중간 코드 일부를 보인다. $12, $13 등은 무한개의 레지스터를 가정하여 생성된 임시변수이다. 그림에서 좌측의 숫자는 문장 번호이고 우측의 번호는 4 이슈로 실행됨을 가정했을 때의 실행 클럭수를 나타낸다. 4 이슈이지만 3번, 4번 문장만 동시에 실행되어 총 4 클럭 사이클이 소요되었음을 볼 수 있다.

그림 2는 그림 1의 DDG를 나타낸다.

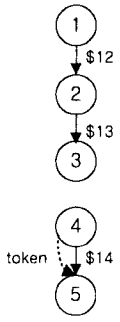


그림 2. 자료 종속 그래프
Fig 2. Data Dependence Graph

DDG는 코드 스케줄링을 수행하기 전에 자료들의 종속관계를 파악하기 위하여 구성한다. 자료 종속은 일반적으로 참종속(true dependence), 반종속(anti dependence), 결과종속(output dependence)의 세 가지 자료 종속 관계를 갖는데 생성된 중간코드의 특성상 참종속과 반종속이 가장 많이 발생하고 이 종속 관계들로 인하여 코드 스케줄링을 방해한다[10]. 참종속은 그림 1에서 1)과 2)의 $12와 같이 1)에서 정의(define)되고 2)에서 사용(use)된 경우이다. 반종속은 그림 1의 token 변수와 같이 4)에서 사용되고 5)에서 정의된 경우이다. 결과종속은 문장에서 모두 정의된 경우이다.

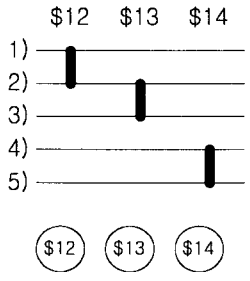
그림 2에서 원은 문번호를, 실선 화살표는 참종속을, 점선 화살표는 반종속을 각각 나타낸다. DDG 상에서 보면 종속관계로 인하여 1)과 4), 2)와 4)나 3)과 4)가 동시에 이슈될 수 있다. 따라서, 이들 쌍중에 4)를 1) 다음으로 코드 이동하면 그림 3과 같이 재정렬될 수 있고 이에 대한 실행 클럭수를 측정할 수 있다.

```
1) LDR   $12, wordct      1
4) ADD   $14, token, #1   1
2) ADD   $13, $12, #1     2
3) STR   $13, wordct      3
5) MOV   token, $14       3
```

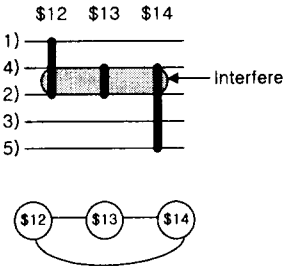
그림 3. 재정렬된 중간코드
Fig 3. Reordered intermediate code

코드 이동으로 인해 실행 클럭수가 총 3 클럭이 소요됨을 볼 수 있으며 이는 그림 1의 총 4 클럭에 비하면 1클럭의 이득을 얻었다. 이것은 프로그램상의 일부를 보여주며 만일 이 중간코드가 루프(loop)를 구성하고 있다면 상당히 많은 실행 클럭수를 줄일 수 있다.

다음으로 레지스터 할당을 위해서 생존 범위를 구하고 이를 토대로 간섭 그래프를 구성하면 다음 그림 4와 같다. 그림 4(a)는 그림 1에 대한 생존 범위(위)와 간섭 그래프(아래)이고 그림 4(b)는 그림 3에 대한 생존 범위(위)와 간섭 그래프(아래)이다. 각 그림의 간섭그래프에서 굵은 실선이 다른 변수들과의 간섭관계를 나타낸다.



(a) 코드 스케줄링 전
(a) Before code scheduling



(b) 코드 스케줄링 후
(b) After code scheduling

그림 4. 간섭 그래프 비교
Fig 4. Comparison of interference graph

그림 4(a)에서 보는 것처럼 코드 스케줄링 전에는 생존 범위가 간섭하는 경우는 발생하지 않았지만 코드 스케줄링 후에는 그림 4(b) 처럼 모든 임시변수들이 서로 간섭한다. 이것은 코드 스케줄링으로 인한 레지스터 할당의 제약을 의미한다. 따라서, 코드 스케줄링과 레지스터 할당의 적절한 조화가 필요하다.

Ⅲ. 선코드 스케줄링의 효율을 높이기 위한 2-단계 컬러링

예를 통해 보여준 선코드 스케줄링의 단점을 보완하기 위하여 본 논문에서는 2-단계 컬러링 기법을 제안한다. 선코드 스케줄링을 위한 전체적인 시스템 흐름은 그림 5와 같다.

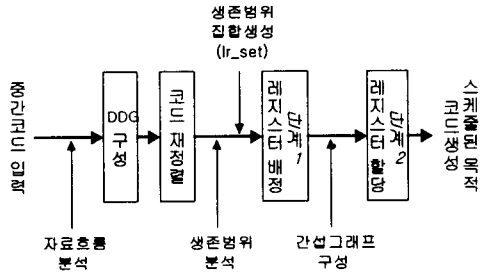


그림 5. 전체적인 시스템 흐름도
Fig 5. Overall system configuration

무한개의 레지스터를 가정하여 생성한 3-주소 형태의 중간코드를 입력받아 자료 흐름 분석을 거쳐 DDG를 구성한다.

다음으로 코드 재정렬을 수행하는데 본 논문에서는 리스트 스케줄링 기법을 사용하였다. 리스트 스케줄링 기법은 기본블록 단위에서 프로파일링 정보(profiling information)를 토대로 트레이스(trace)를 분석한 후 명령어를 이동한다. 이 기법은 대부분의 경우에 최적화된 결과를 생성하며 초기 코드 스케줄링 결정을 다시 고려할 필요가 없으므로 속도가 빠르다는 장점을 갖는다.

코드 재정렬 단계가 끝나면 변수들의 생존 범위를 측정하는 생존 범위 분석 단계를 거친 후 생존 범위 집합(live range set, lr_set)을 구한다. 이것은 레지스터 할당을 위한 초기 과정이다. 각 변수들의 마지막 사용($last_{use}$)에서 첫 번째 정의($define_{first}$)된 문장 번호를 빼면 해당 변수의 생존거리가($live_{dist}$)되는데 생존거리가 10 이상인 변수 집합(=lr_set_{long})을 스택(stack)에 푸시(push)한다.

다음의 알고리즘에 의하여 구현된다.

Live range analysis;
Do{// Lr_set;
A variable;
Livedist =$last_{use}$-$define_{first}$;
If ($Live_{dist}$)= 10)
Lr_set_{long} = variable;
Push lr_set_{long} onto stack;
}While(each variables);

그림 6. 생존 범위 집합 생성 알고리즘
Fig.6. Live range set generation algorithm

생존거리 10을 기준으로 한 이유는 10 이상인 변수들이 사용자 정의 변수들로 프로그램 전반에 걸쳐 영향을 미치기 때문에 긴 생존거리를 갖는다. 그리고, 여러 프로그램들을 가지고 생존거리를 측정한 결과 작은 프로그램의 경우에도 10 정도에서 대부분 나누어지고 중간코드에서 생성된 임시변수들은 대부분 거리가 1인 특성이 있었다.

레지스터 할당 단계에서는 간섭 그래프를 구성하는 비용이 상당히 많이 소요[11] 되므로 이를 줄이기 위한 방안을 제시한다. 전 단계에서 구한 lr_set_{long}의 변수들을 간섭 그래프 구성 전에 미리 레지스터에 배정한다.

단계 1의 레지스터 배정은 단순히 변수들을 레지스터에 배정시키는 과정으로 생존거리가 긴 변수들을 차례로 실제 레지스터에 배정한다.

단계 2에서는 레지스터 할당을 수행하는데 이를 위해 간섭 그래프를 먼저 구성한다. 레지스터 할당은 변수들 중에서 실제 레지스터에 어떤 변수를 배정해야하는지 우선 선택하고 이를 배정하는 과정이다. 나머지 변수들을 가지고 간섭 그래프를 구성하며 레지스터를 할당한다. 마지막으로 선코드 스케줄된 목적코드를 생성한다.

전체적인 알고리즘은 그림 7과 같다.

```
DDG 구성;
Code Scheduling;
Live range set generation;// 그림 6
While(lr_set_long){// phase 1
   Pop a variable from the stack;
   Assign a physical register;
}
For(remained variables){// phase 2
   Chaitin's graph coloring algorithm;
}
```

그림 7. 전체적인 2-단계 컬러링 알고리즘
Fig 7. 2-phase coloring algorithm

단계 2에서는 기존에 많이 응용되고 있는 Chaitin의 그래프 컬러링 레지스터 할당 알고리즘을 본 논문에서도 적용하였다. Chaitin의 방법은 그림 8과 같다.

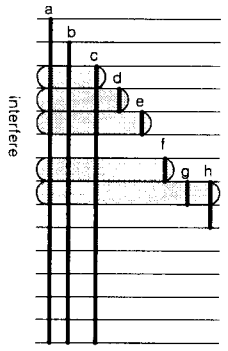
```
Do {
 Construction of the interference graph for
remained variables;
  Do {
    If (degree a node < color) {
      Prepare to coloring;
      }
    Else {
      Prepare to spilling;
        }
  }While (graph is not empty)
  Add the spill code;
  }While (spill is need)
 Do {
  Assign a color;
 }While (coloring stack not empty)
```

그림 8. Chaitin의 레지스터 할당 알고리즘
Fig 8. Chaitin's register allocation algorithm

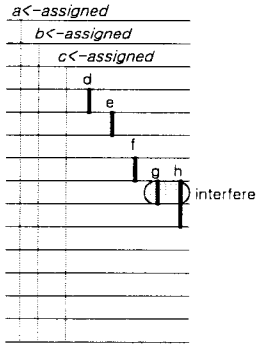
그림 8에서 보는 것처럼 레지스터 할당에서는 간섭 그래프 구성이 매우 중요하고 시간도 많이 필요로 한다.

따라서, 본 논문에서는 레지스터 할당 알고리즘을 수행하는 과정 중 단계 1에서 생존거리가 긴 변수들을 레지스터에 미리 배정함으로써 간섭 그래프를 간소화시킬 수 있는 방법을 제시하였다.

만일 중간코드가 코드 재정렬을 거쳐 그림 9(a)와 같은 생존 범위를 갖는다면 이를 단계 1을 거치면 변수 a, b, c가 레지스터에 배정되기 때문에 그림 9(b)의 생존 범위를 갖게된다. 이것은 간섭 그래프가 간소화되었음을 의미하며 따라서, 단계 2에서 빠른 레지스터 할당이 가능하다.



(a) 코드 재정렬 후의 생존 범위
(a) Live range after the code reordering



(b) 단계 1 후의 생존 범위
(b) Live range after the phase 2

그림 9. 생존 범위 비교
Fig 9. Compare live range

실제 중간코드들을 가지고 실험한 결과 그림 9(a)와 같은 형태의 생존 범위를 갖는 경우가 대부분이었다. 즉, 생존범위가 긴 변수들이 사용자 정의 변수들로 몇 개 존재하고 나머지는 생존 범위가 짧은 임시변수들이 대부분인 것으로 판명되었다.

Ⅳ. 시뮬레이션

제안한 방법의 타당성을 증명하기 위하여 11개의 벤치마크 프로그램을 가지고 다양하게 실험하였다.

먼저 간섭 에지수를 비교하였다. 결과는 표 1과 같다.

표 1에서 간섭 에지수는 단계 1을 거친 후의 간섭 에지수를 의미하며 표에서 'Other'에 대한 정의는 본 논문에서는 lr_set_{long}인 변수들을 미리 레지스터에 배정했지만 이와 다르게 생존거리가 짧은 임시변수들($live_{dist}=1$)을 미리 레지스터에 배정한 방법이다. 이 방법도 효과적인 것이 표 1의 실험 결과를 보면 Chaitin이 제안한 방법에 비하여 간섭 에지수가 현저히 감소되었음을 알 수 있다.

표 1. 간섭 에지수 비교
Table 1. Compare no. of interfere edge

벤치마크 프로그램	간섭 에지수		
	Chaitin의 방법	2-단계 컬러링	Other
fibo.c	76	1	20
diff.c	120	7	66
mami.c	116	5	35
patt.c	55	9	16
bubble.c	110	13	51
shell.c	109	3	31
quick.c	174	17	116
selection.c	42	2	8
insert.c	105	5	32
binsh.c	107	6	36
wc.c	147	7	30

표 1에서 2-단계 컬러링의 경우 피보나치 수열(fibo.c)에서 간섭 에지가 1개 밖에 존재하지 않아 레지스터 할당이 신속히 이루어질 수 있으며 다른 벤치마크 프로그램들도 간섭 에지수가 다른 실험 결과들에 비해 훨씬 더 감소하였다. 이를 통해서 2-단계 컬러링 알고리즘을 적용한 제안한 방법이 효과적임을 증명해준다.

표 1을 도표로 나타내면 그림 10과 같다.

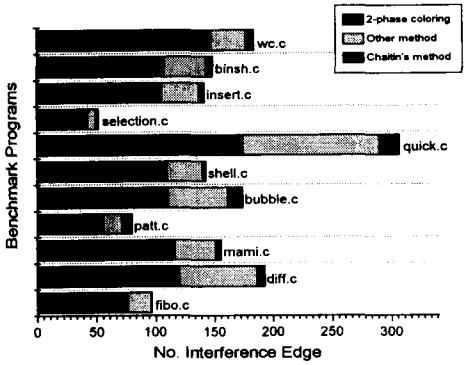


그림 10. 표 1에 대한 도표
Fig 10. Chart of Table. 1

각 벤치마크 프로그램들에 대해 단계 2를 수행 한 후의 실제 소요되는 레지스터 개수를 측정하였다. 이것은 본 논문에서 제안하고자 하는 방법이 Chaitin의 방법에 비해 소요되는 레지스터 수가 어느 정도 증가하는지를 보기 위한 실험이다. 결과는 표 2, 그림 11과 같다.

표 2. 소요 레지스터 수 비교
Table 2. Compare no. of used registers

벤치마크 프로그램	소요 레지스터 수		
	Chaitin의 방법	2-단계 컬러링	Other
fibo.c	7	9	9
diff.c	9	9	10
mami.c	8	8	9
patt.c	5	7	7
bubble.c	8	8	8
shell.c	8	8	9
quick.c	10	13	13
selection.c	5	6	5
insert.c	7	8	8
binsh.c	7	8	8
wc.c	8	8	8

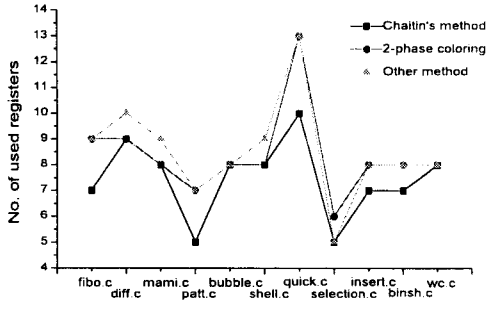


그림 11. 표 2에 대한 도표
Fig 11. Chart of Table. 2

그림 11에서 보면 제안한 방법이 Chaitin의 방법에 비해 총 여섯 개의 벤치마크 프로그램에서 1개~3개까지의 레지스터가 더 소요되는 것으로 측정되었다. 그러나, Other 방법을 적용했을 때에 비해 더 적은 수의 레지스터가 소요됨을 볼 수 있고 이것은 간섭 그래프의 간소화로 인한 결과로 생각된다.

기존의 방법에 비해 레지스터가 더 소요되는 부분은 본 알고리즘을 개선할 필요성이 있다는 것을 보여준다. 만일, 최근 ILP 프로세서들이 레지스터 개수를 늘려가고 있는 추세로 미루어 볼 때 충분한 레지스터를 보유하고 있다면, 제안한 방법이 더욱 효율적일 수 있을 것으로 판단된다.

실행 클럭수 측정 결과는 세 방법 모두 동일하였다. 4-이슈일 때의 실행 클럭수 측정 결과는 표 3에서 보인다.

표 3. 실행 클럭수
Table 3. No of execution clock cycles

벤치마크 프로그램	실행 클럭수
fibo.c	300
diff.c	156
mami.c	165
patt.c	227
bubble.c	2777
shell.c	1701
quick.c	1390
selection.c	2003
insert.c	420
binsh.c	2541
wc.c	1579

V. 결론 및 과제

지금까지 선코드 스케줄링의 최적화를 위한 연구에 대한 연구 과정 및 실험 결과들에 대해서 논하였다.

코드 스케줄링을 먼저 수행하고 레지스터 할당을 다음에 수행하는 선코드 스케줄링의 경우 코드 스케줄링으로 인한 자료 종속 관계가 복잡해져서 레지스터 할당을 제약한다. 즉, 간섭 그래프를 복잡하게 만들어 레지스터 할당 시간을 증가시키게된다. 본 논문에서는 이러한 제약을 완화시키기 위하여 2-단계 컬러링 기법을 제안하였다. 단계 1에서 생존거리가 긴 변수들을 레지스터에 배정하고 단계 2에서 나머지 변수들을 가지고 레지스터 할당을 수행하는데, 이미 생존거리가 긴 변수들을 레지스터에 배정했기 때문에 짧은 생존거리를 갖는 변수들은 레지스터 할당 시에 빠른 속도로 배정 받을 수 있다.

실험을 통하여 제안한 방법의 타당성을 확인하였고 이를 증명하였다. 그러나, 소요되는 실제 레지스터 개수가 제안한 방법이 Chaitin의 방법에 비해 더 필요한 것은 제안한 방법의 개선을 필요로 한다는 것을 보여준다. 그리고, 레지스터 할당 후에 코드 스케줄링 루틴을 더 추가하여 보다 완전한 최적화 시스템을 만들고자 한다.

참고문헌

[1] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, D. I. August, "Compiler Technology for Future Microprocessors," Proceedings of the IEEE, Vol. 83, No. 12, pp. 1625~1640, Dec. 1995.

[2] 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.

[3] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, 1991.

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

[5] A. Nicolau and R. Potasman, "Realistic Scheduling: Compaction for Pipelined Architecture," Proc. of MICRO-23, pp. 69~79, 1990.

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

[7] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to Graph Coloring Register Allocation," ACM Transactions on Programming Languages and Systems, pp. 428~455, 1994.

[8] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," Computer Languages, pp. 47~57, 1981.

[9] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen and W. 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.

[10] John L Hennessy, David A Patterson, Computer Architecture a Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc.

[11] K. D. Cooper, T. J. Harver, L. Torczon, "How to Build an Interference Graph," Software-Practice and Experience, 1998.

저 자 소 개

최준기

1993. 2. 순천향대학교 전산학과(공학사)

1995. 2. 순천향대학교 전산학과(공학석사)

1999. 2. 순천향대학교 전산학과(공학박사)

1999. 3. ~ 현재 인덕대학 여성정보행정과 전임강사