論文 90-27-3-6

RISC 컴파일러의 기계독립적 Global Optimizer 설계

(The Design of A Machine-independent Global Optimizer for RISC Compilers)

朴 鍾 得* 林 寅 七*

(Jong Deuk Park and In Chil Lim)

要 約

본 논문에서는 RISC 컴파일러 시스템 구현에 필요한 기계독립적인 광역적 최적화부(global optimizer)의 설계 방식을 제안하고 실현한다. 제안된 최적화부는 트리플 표현을 입력으로 받아 데이타 흐름 분석 및 공통부식제거와 코드 이동을 수행하고 최적화된 트리플 표현을 출력시킨다. 본 최적화부는 기계 독립적인 중간언어를 대상으로 하기 때문에 다양한 고급언어와 타겟 머신에 대해서 이식성이 용이하며 프로그램의 실행속도를 향상시킬 수 있는 효율적인 최적화를 수행하도록 구성된다.

Abstract

This paper describes a design and an implementation of a machine-independent global optimizer which is a required module of RISC compiler system designs. It receives a triple as input and performs data flow analysis, common subexpression elimination and code motion and finally generates the optimized code. Since the implemented optimizer operates on the machine-independent intermediate code, its portability is good for many high level languages and target machines. It performs the effective optimizations to improve the execution time of programs.

I. 서 론

현재 RISC 아키텍춰는 하드웨어 자원의 이용 효율을 높이고 고속의 클럭사이클로 대부분의 명령어를 단일 사이클로 수행하는 방식으로, 성능이 우수한 마이크로프로세서 설계에 널리 이용되고 있다.[1],[2]

RISC 개념으로 설계된 컴퓨터 시스템은 하드웨어의 단순화를 지향하여 대부분의 명령어들을 단일 사이클 내에 실행 가능토록 한다. 즉 RISC 아키텍춰는 명령어의 크기를 동일하게 하고 명령어의 형식을 고정시켜 명령어 디코드 시간을 줄이고, 대부분의 명령어에 공통적으로 사용되는 데이타 패스외에 특별한 하드웨어 지원이 필요한 명령어와 사용 빈도수가 적은 명령어를 배제하여 개개의 명령어를 더 빠르게 실행시키는 설계 방식이다. 그런데 RISC 기술은 한 명령어를 한 사이클로 처리할 수 있는 능력을 갖춘 반면 기본적인 명령어들을 사용하기 때문에 생성된 코드가 길어지게 되므로 프로그램 실행속도를 저하시키는 요인이 된다. 따라서 이러한 코드를 최적

*正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)
接受日字 : 1989年 10月 17日

화 하여 RISC의 특성을 최대한 살릴 수 있는 컴파일러의 설계기법이 필요하며 이에 대한 많은 연구가 이루어지고 있다.[3][6][7]

컴파일러의 역할은 고급언어 프로그램을 실제의 실행 스텝으로 변환하는 것이며 RISC의 경우 그 변환 자체가 수월하다는 특징을 갖는다. RISC 시스템에서는 한정된 명령어 세트에 따라 명령어를 선택하는 경우의 수가 국한되어 CISC 컴파일러에서와 같이 동일한 기능의 여러가지 명령어 중에서 어떤 명령어를 취할 것인지에 관여할 필요가 없으므로 컴파일러의 구성이 간단해진다. 다만 최적화된 코드를 생성하기 위해서는 효율이 좋은 고성능의 최적화부가 필요하게 된다. 최적화는 기계 종속적인 최적화와 기계 독립적인 최적화로 나눌 수 있다. 기계 종속적인 최적화는 타겟이 되는 머신의 하드웨어 특성과 명령어를 참조하기 때문에 일반적으로 코드 생성기(code generator)에 의해 수행되며 최적화되는 범위도 주로 프로그램의 국소적인 영역으로 한정되어 진다. 기계독립적인 최적화 알고리듬은 소스 프로그램이나 코드 생성기에 무관하게 생성되므로 다양한 고급 언어와 타겟 머신에 대해 유용하다. 현재 Stanford U-Code 컴파일 시스템[7]은 기계 독립적인 최적화를 위해 U-Code를 중간 언어로 사용하고 있다. 그러나 대부분의 RISC 컴파일 시스템이 어셈블러 수준의 기계종속적인 단계에게 최적화를 수행하고 있으며 수행되는 최적화의 범위도 파이프라인으로 기계어를 효율적으로 실행할 수 있도록 코드를 재배열하는 정도에 국한되어 있는 실정이다.

본 논문에서는 RISC 머신을 대상으로한 기계 독립적인 최적화 컴파일러의 설계에 필요한 광역적 최적화부를 제안하고 실현한다. 제안된 광역적 최적화부는 고급언어 프로그램으로부터 번역된 중간코드에 대해 최적화 알고리듬을 수행하여 최적화된 중간 코드를 생성한다. 중간 코드의 표현은 트리플(triple) 형식으로 하여 최적화를 수행한다. 또한 어느 한 최적화 알고리듬을 수행함에 따라 또 다른 최적화 알고리듬을 적용시킬 수도 있으므로 더 이상 최적화되지 않을 때까지 반복적으로 수행 되도록 한다.

Ⅱ. 중간 코드

1. 3주소 코드(3 address code)

일반적으로 이식이 용이한 컴파일러 설계는 중간 코드를 이용한다. 고급 언어 프로그램을 기계독립적인 중간언어로 변화함으로써 컴파일러의 이식성이 높아지며, 컴파일러를 파싱(parsing)을 위한 전반부(front-end)와 코드 생성을 위한 후반부(back-end)로 분리하는 것이 가능하다. 본 최적화부 설계에서는 중간코드의 효과적인 최적화를 위하여 3주소 코드를 중간언어로 사용한다.

3주소 코드의 일반형식은 x:=y op z이다. 여기서 x, y, z는 컴파일러에 의하여 생성된 임시 변수 또는 상수이고 op는 산술 연산자나 논리 연산자이다. 3주소코드의 형식은 다음과 같다.

(1) x:=y op z 형식의 문(statement)

여기서 op는 2항 연산자이고 산술 또는 논리 연산을 행한다.

(2) x:=op y 형식의 문

여기서 op는 단항 연산자이고 기본적인 연산은 단항 마이너스, 논리연산, 시프트 연산자를 포함한다.

(3) x:=y 형식의 카피 문

(4) go to L 형식의 무조건 점프문

레이블 L이 있는 3번지 문이 다음에 실행될 문이다.

(5) if x relop y goto L 형식의 조건 점프문

이 조건문은 x와 y에 대해 관계 연산자(<, <=, ==, !=, >=, >)를 적용하여 만일 x가 y에 대한 relop 관계를 나타내면 레이블 L 다음의 문을 실행한다.

(6) 서브 프로시쥬어(subprocedure)의 호출과 복귀를 위한 push, call, ret(return)문

서브 프로시쥬어 $p(x_1, x_2, \cdots, x_n)$를 호출할 경우 생성된 3 주소문의 시퀀스(sequence)는 다음과 같다.

```
push xn          _P:
...                :
push x2            :
push x1          ret
call _P
```

여기서 n은 프로시쥬어 p를 호출할 때 사용하는 실파라메터(actual parameter)의 갯수이다. 또한 호출된 프로시쥬어의 종료는 ret문에 의해서 표시된다. 호출된 서브 프로시쥬어에서 return문을 만나면 제어는 호출 함수로 넘어간다.

(7) x:=&y와 x:=*y의 번지와 포인터 형식문

문 x:=&y에 있어서 x의 값은 y의 주소(address)가 된다. 문 x:=*y에 있어서 x의 값은 y가 가리키는 주소의 내용이다.

3주소 문은 중간코드의 abstract 형식이다. 그런데 실제의 컴파일러에서는 연산자와 오퍼랜드(operand)를 위한 필드(field)를 가진 레코드로 구현할 수 있고 그 방식은 쿼드러플(quadruple)과 트리플 방식이 있

다. 트리플 방식이 파서 트리(parser tree)를 쉽게 다룰 수 있으므로 트리플 형식의 중간 코드를 채택한다.

2. 트리플

트리플 형식의 표현은 임시 변수 이름이 심볼(symbol)로 표시되지 않도록 하기 위하여 오퍼랜드 필드에 이미 계산된 문의 문 번호를 사용할 수 있도록 한 것이며 그림 1과 같이 오직 3개의 필드가 있는 레코드(⟨op, cleft, cright⟩)에 의해서 표시될 수가 있다. Cleft와 cright는 임시 변수값에 대한 레코드의 엔트리 포인터이다. 그림 1은 a:=b*c+d문에 대해서 트리플 형식으로 표현한 것이다. (a, b, c, d는 임시 변수)

1	변수a		
2	변수b		
3	변수c		
4	*	(2)	(3)
5	변수d		
6	+	(4)	(5)
7	:=	(1)	(6)

그림 1. a:=b*c+d의 트리플 형식 중간 코드
Fig. 1. Triple intermediate code of a:b*c+d.

Ⅲ. 제어 흐름도와 제어흐름 분석

1. 기본 블럭(basic block)

기본 블럭은 제어 흐름도를 구성하는 기본 노드가 되는 것으로 기본 블럭의 시작으로부터 분기없이 그 기본 블럭의 마지막 문까지 실행하는 연속적인 문의 시퀀스이다. 기본블럭의 구성은 리더를 포함하며 다음 리더 및 프로그램 종료가 아닌 모든 문으로 이루어진다.

여기서 리더는 기본 블럭의 첫문을 결정하는 것으로 첫 문, 조건 분기 또는 무조건 분기 문의 타겟인 모든 문, 무조건 분기 명령어나 조건 분기 명령어 다음의 모든 문들이다.

2. 제어 흐름도

제어 흐름도는 제어의 흐름을 표시하며 제어 흐름도에서 각 노드는 기본 블럭을 나타낸다. 아래에 1부터 9까지의 합을 구하는 C프로그램에 대한 중간코드를 나타내었다.

```
main ( )
{
  int i, s;
  s=0;
  for (i=1 ;i<10 ;i++)
    s=s+i;
}
```

```
 1  main :
 2       t1 : =fp−8
 3     *t1 : =0
 4       t2 : =fp−4
 5    **t2 : =1
 6  L18 :
 7       t3 : =fp−4
 8       t4 : =*t3
 9       t5 : t4<10
10       if ! t5 goto L17
11       t6 : fp−8
12       t7 : =fp−8
13       t8 : =*t7
14       t9 : =fp−4
15       t10 : *t9
16       t11 : =t8+t10
17     *t6 : =t11
18 L16 :
19       t12 : =fp−4
20       t13 : =*t12
21     *t12 : =t13+1
22       goto L18
23 L17 :
24       ret
```

위의 중간코드에서 fp는 메모리의 변수 참조를 위한 프레임 포인터(frame pointer)이다. 또한 레이블 L16은 for문의 특성상 문법(gramma)기술시에 세만틱 액션(semantic action)에 따라 더미(dummy)로 생성된 것이다. 그림 2는 위 예제의 중간코드에 대한 제어흐름도를 나타낸다.

그림 2에서 노드B_1은 노드B_2의 이전자 노드(predecessor node)이고, 노드B_2는 노드B_1의 계승자 노드(successor node)라고 부른다.

3. 도미네이트(dominator)

만일 제어 흐름도의 초기 노드로 부터 노드 n으로의 경로가 노드 d를 통해 간다면 제어 흐름도의 노드 d가 노드 n을 지배(dominate)한다고 말하며 d dom n으로 쓴다. 따라서 이 정의 하에서, 모든 노드는

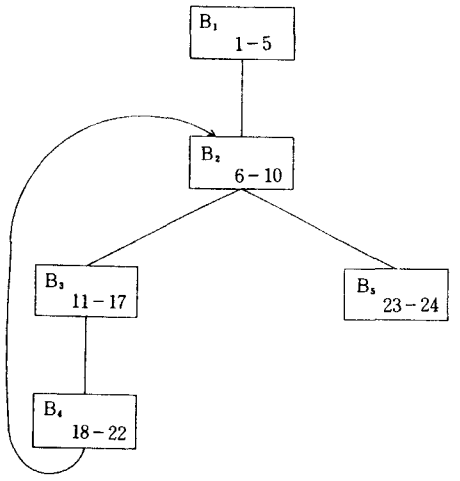


그림 2. 예제 중간코드의 제어 흐름도
Fig. 2. Control flow graph of the above intermediate code.

그 자신을 지배하며 루프의 엔트리는 루프내의 모든 노드를 지배한다. 그림2의 제어 흐름도에서 5번 노드로 가기위해서는 1, 2, 5번 노드를 반드시 거쳐야 하므로 5번 노드를 지배하는 노드들은 1, 2, 5번 노드들이다.

4. 깊이 우선 탐색(depth first search)

제어 흐름도에서 루프를 찾기 위해 사용될 수 있으며, 반복 데이타 흐름 알고리듬(inerative data flow algorithm)을 빠르게 하도록 돕는 깊이 우선 탐색방법을 사용한다. 깊이 우선 탐색은 가능한 한 빨리 초기 노드로 부터 멀리 떨어진 노드를 방문하려고 시도하면서 전체 그래프를 탐색함으로써 만들어진다. 이 탐색 루트(route)는 트리를 형성한다.

제어 흐름도의 깊이 우선 탐색은 트리의 프리오더 운행법(preorder traversal)과 일치한다. 그림2의 경우 1→2→3→4→3→2→5→2→1의 순으로 방문한다.

Ⅳ. 광역적 최적화

1. 순방향 최적화(forward optimization)

1) 유효식(available expression)

초기 노드로 부터 어떤 지점 p에 이르는 모든경로에서 계산식 x op y를 계산하고 그 지점 p에 도달하기까지 변수 x나 변수 y에 대한 assign 문이 없다면, 즉 변수 x나 변수 y를 새로이 정의하지 않는다면, 계산식 x op y는 그 지점 p에서 유효식이 된다. 기본블럭내에서 어떤 유효식에 대해 변수 x나 변수 y를 정의하고 정의 후 계산식 x op y가 없다면 블럭은 식 x op y를 kill한다고 말하고, 기본 블럭이 식 x op y를 계산하고 나서 변수 x나 변수 y를 정의하지 않으면 식 x op y를 생성한다고 한다.

집합U가 프로그램의 하나 또는 그 이상의 문 오른쪽에 나타나는 모든 식의 전체 집합이라고 가정한다. 각 블럭 B에 대해, in[B]가 B의 시작 직전의 문장에서 유효한 U안의 식의 집합이라고 하고 out[B]가 B의 종료 직후의 문장에서 유효한 집합이라고 하자. 다음 방정식은 각 기본 블럭에서 알려진 ae_gen과 ae_kill을 이용하여 in과 out을 구하기 위해 쓰여진다.

$$out[B] = in[B] - ae\_kill[B] + ae\_gen[B]$$

$$in[B] = \cap out[P] \quad \text{(여기서 B는 초기 노드가 아니고 P는 B의 이전자 노드)} \qquad (1)$$

$$in[B_1] = \{\ \} \quad \text{(여기서 } B_1 \text{은 초기 노드)}$$

in, out, ae_gen, ae_kill은 모두 비트 벡터(bit vector)로 표시되는데 ae_gen은 해당 블럭에서 생성되는 유효식의 집합이며 ae_kill은 kill되는 유효식의 집합이다. (1)식을 반복 데이타 흐름 알고리듬 방식으로 예제의 중간 코드에 대하여 적용시키면 그림 3과 같은 비트 벡터 정보를 얻을 수 있다.

그림 3에서 각 문번호에 대한 정보표현은 계산식 x op y에 관한 것이다.

```
기본블럭 1.
ae_in[1]   2 3 4 5 6 8 9 10111213141516171920 21
ae_in[1]   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ae_gen[1]  1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ae_kill[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ae_out[1]  1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
기본블럭 2.
ae_in[2]   1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ae_gen[2]  0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
ae_kill[2] 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
ae_out[2]  1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
기본블럭 3.
ae_in[3]   1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
ae_gen[3]  0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0
ae_kill[3] 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0
ae_out[3]  1 1 1 1 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0
```

그림 3. 예제에 대한 데이타 흐름 정보
Fig. 3. Data flow information for the above intermediate code.

2) 광역적 공통부식 제거

유효식의 정보를 사용하는 이유는 제어 흐름도에서 각 노드 간의 공통부식에 대하여 최적화를 수행하기 위해서이다. 예를 들면 그림 4 에서 B_1은 a:=b*c+d를 계산하는 식이고 B_2는 e:=b*c+e를 계산하는 식일때 그림 4(a)에서 (4)번 식b*c는 (1)번식 b*c와 같은 값을 가지고 있다. 따라서 그림 4(b)와 같이 (5)번 식을 계산하기 위해서 (4)번 식을 다시 계산하지 않고 이미 계산된 공통 부식인 (1)번 식을 이용하여 최적화 시킬 수 있다. 광역적 공통부식 제거를 수행한 후 유효식의 광역적 데이타 흐름 정보에 따라 수행된다.

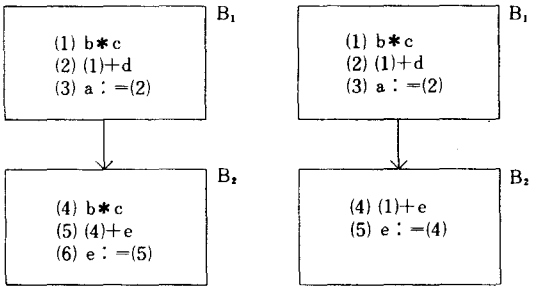


그림 4. 공통 부식 제거 예
(a) 제거 전
(b) 제거 후

Fig. 4. Common subexpression elimination example.
(a) before.
(b) after.

2. 역방향 최적화(backward optimization)

1) 루프 인베리언트 계산(loop invariant computation)의 발견

루프의 수행동안 루프안의 어떤 식이 항상 일정한 값을 유지할 때 이 식을 루프 인베리언트 계산식이라 하고 루프안에서 이러한 식의 발견을 위해서 ud-chain을 사용한다. 여기서 ud-chain은 사용된 각 변수에 대하여 그 사용된 각 변수에 대하여 그 사용된 변수에 도달하는 정의의 리스트(list)이다. 루프는 다른 모든 블럭을 지배하는 해더(header)를 포함하는 블럭의 집합이다. 따라서 루프로 들어가는 유일한 경로는 헤더를 통하는 경로 뿐이다. 또한 루프는 루프 내의 모든 블럭으로 부터 헤더로 되돌아 갈 수있는 적어도 하나 이상의 경로를 가져야 한다.

만약 계산식 x:=y+z의 변수 y와 변수 z의 모든 가능한 정의가 루프밖에 위치한다면, 계산식 y+z는 루프 인베리언트 계산식이다. 왜냐하면 제어가 루프안에 있는한 계산식 x:=y+z를 만날 때마다 동일한 값을 유지하기 때문이다. 이러한 루프 인베리언트 계산식은 ud-chain, 즉 계산식 x:=y+z의 모든 정의 포인트 리스트를 이용하여 발견할 수 있다.

계산식 x:=y+z에서 계산된 변수x의 값이 루프안에서 변하지 않는다는 것을 알고 다른 계산식 v:=x+w가 있다고 가정하자. 여기서 변수w가 오직 루프의 밖에서 정의되었거나 루프 인베리언트일 경우 계산식 x+w 또한 루프 인베리언트 계산식이다. 계산식의 값이 루프안에서 일정하게 유지되는, 보다 많은 루프 인베리언트 계산식을 발견하기 위해서 더이상 루프 인베리언트 계산식이 발견되지 않을때까지 루프를 반복 수행한다.

2) 코드 모션의 실행

루프 안에서 루프 인베리언트 계산식을 발견하면 코드 모션은 루프안에서 인베리언트한 계산식을 루프의 프리 헤더로 이동시킨다. 여기서 프리 헤더란 루프 헤더를 유일한 계승자로 가지는 것으로 루프 밖으로부터 루프 안으로 들어가는 모든 에지는 반드시 프리 헤더를 거치도록 에지를 연결해 주어야 한다.

3. 적용예 및 검토

본 논문에서는 기계독립적인 광역적 최적화부를 VAX-11/750(4.3BSD)에서 C언어로 실현하여 그 효율성을 입증한다.

프로그램 언어로는 현재 널리 사용되고 있는 C언어를 대상으로 한다. 앞서의 예제 중간코드에 대한 최적화 수행 후의 트리플 표현을 3주소 코드로 나타내면 다음과 같다.

```
1  main:
2      t1:=fp-8
3    *t1:=0
4      t2:fp-4
5    *t2:=1
6  L18:
7      t4:=*t2
8      t5:=t4<10
9      if ! t5 goto L17
10     t10:=*t2
11     t11:=*t1+t10
12   *t1:=t11
13 L16
```

```
14      t13:=*t2
15    *t2:=t13+1
16      goto L18
17 L17:
18      ret
```

표 1은 테스트 프로그램으로 사용한 여러 개의 프로그램 코드에 대해 생성된 트리플에 대한 최적화 결과를 나타낸 것이다.

표 1. 테스트 프로그램에 대한 본 최적화부의 수행 결과

Table 1. Execution result of the optimizer for test programs.

프로그램	최적화 전 중간 코드 수	최적화 후 중간 코드 수	감소율 (%)
bsearch	43	38	11.6
bubble sort	49	39	20.4
merge sort	202	175	13.4
shaker sort	93	72	22.6
shell sort	61	41	16.4

Ⅳ. 결 론

본 논문에서는 RISC 컴파일러의 설계에 필요한 기계독립적인 광역적 최적화부를 실현하여 생성된 코드의 실행 속도를 향상시킬 수 있도록 하였다.

설계된 최적화부는 컴파일러의 전반부와 후반부 사이에서 동작하며 타겟 머신이 변경되더라도 이식이 용이하도록 하였다. 중간 코드는 트리플 표현을 사용하였으며 트리플에 고유의 번호를 부여하여 코드의 이동에 유연성을 부여하였다. 그리하여 트리플 표현에 대한 제어 흐름도를 작성하고 데이타 흐름분석의 결과를 바탕으로 순방향 최적화 기법인 공통부식제거와 역방향 최적화 기법인 코드 모션의 적용으로 최적화된 트리플 표현이 출력으로 생성되도록 하였다. 또한 어느 한 최적화 알고리듬을 수행함으로써 또다른 최적화 알고리듬을 적용하게 될 수도 있으므로 더 이상 최적화될 수 있는 경우가 없을 때까지 반복적으로 수행하게 하였다. 그러나 최적화율이 낮은 알고리듬의 적용은 컴파일 시간을 과다하게 할 우려가 있으므로 가급적 주요한 알고리듬 만을 적용하였다.

앞으로의 연구과제로는 RISC 머신의 특징 중의 하나인 레지스터 파일을 이용한 레지스터 할당 알고리듬의 개발과 트리플 표현을 타겟 머신 코드로 출력하는 코드 생성기의 개발이 요망되고 있다.

參 考 文 獻

[1] William Stallings, "Reduced instruction set computer architecture," *Proc. of the IEEE*, no. 1, Jan. 1988.

[2] M.G.H. Katevenis, "Reduced instruction Set computer architectures for VLSI," Ph.D. Dissertation, Computer Science Dept., Univ. of California, Berkeley, Oct. 1983.

[3] J. Hennessy, N. Jouppy, F. Baskett, T. J. Gill, "Hardware/Software Tradeoffs for Increased Performancd," Proc. of ACM. Symp. on Architectural Support for Programming Lang. and Operating Systems, pp. 2-11, Mar. 1982.

[4] A.V. Aho and R. Sethi and J.D. Ullman, Compilers Principles, Techniques, and Tools. Addison-Wesley, 1986.

[5] M.S. Hecht, Flow Analysis of Computer Programs. North-Holland, NewYork. 1977.

[6] F.C. Chow, "A portable machine-independent global optimizer-Design and measurements," Ph.D. Dissertation, Electrical Engineering Dep., Univ. of Stanford, Dec. 1983.

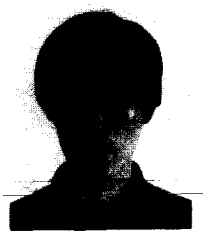
[7] F. Chow, M. Himelstein, E. Killian and L. Weber, "Engineering a RISC Compiler system," (Proceedings of COMPCON Spring 86), pp. 132-137, Mar. 1986.

[8] J.W. Davidson and C.W. Fraser, "The design and application of the peephole optimizer," ACM Trans. Program. Lang. Syst. 2, 2 (Apr. 1980), 191-202.

[9] "RISC 파이프라인 아키텍춰의 코드 스케줄링 알고리듬",황병현, 김주형, 이병노, 진은경, 박종득, 김은성, 임인칠, 대한전자공학회, 추계종합학술대회 논문집, 제11권 제 1 호, pp. 226-230, 1988. 11.

[10] "RISC 머신을 위한 기계독립적 Optimizer 의 설계",조정삼, 이형우, 이병노, 김주형, 황병현, 진은경, 박종득, 김은성, 임인칠, 대한전자공학회 하계종합학술대회 논문집, 제12권 제1호 pp. 304-307, 1989. 7.

著 者 紹 介



朴 鍾 得(正會員)
1959年 9月 13日生. 1983年 2月 한양대학교 전자공학과 졸업. 1985年 2月 한양대학교 대학원 전자공학과 졸업 공학석사 학위취득. 1985年 3月~현재 한양대학교 대학원 전자공학과 박사과정 재학중. 주관심분야는 RISC Compiler Design, Computer Architecture, Microprogramming 등임.

林 寅 七 (正會員) 第25卷 第8號 參照
현재 한양대학교 전자공학과 교수