희소코드모션을 위한 효율적인 알고리즘

신 현 덕[†] · 유 희 종[†] · 안 희 학[††]

요 약

이 논문에서는 계산적으로나 수명적으로 코드를 최적화 하기 위해 절제된 코드 모션 알고리즘을 제안한다. 제한한 알고리즘은 BCM과 LCM 알고리즘을 확장한 SpCM 알고리즘이다. SpCM 알고리즘에서 BCM 알고리즘은 계산적으로 최적의 코드 모션을 수행하며, LCM 알고리즘은 레지스터 압박을 감소시킨다. 일반적으로, 코드 모션 알고리즘은 계산의 최적화와 레지스터 압박에 관련된 실행시간 최적화를 실행한다. 이 논문에서는 계산 비용과 레지스터 압박에 코드의 크기를 고려하는 부분을 추가하였다. 코드의 계산적 최적화와 수명적 최적화에 이어 코드의 크기를 고려하는 SpCM 알고리즘에 의해 코드 모션의 최적화 결과를 얻을 수 있다. 이 논문에서 제안한 알고리즘은 모든 불필요한 코드 모션을 억제시키기 때문에 계산적으로나 수명적으로 최적인 알고리즘이라 할 수 있다.

An Efficient Algorithm for Sparse Code Motion

Hyun-Deok Shin[†] · Heui-Jong Yu[†] · Heui-Hak Ahn[††]

ABSTRACT

This paper suggests that sparse code motion algorithm should be used to make the code optimal in the respect of computation and lifetime. This algorithm is SpCM algorithm, which expand BCM and LCM algorithm. BCM algorithm carries out the optimal code motion computationally and LCM algorithm reduces the register pressure in SpCM algorithm. Generally, code motion algorithm accomplishes the run-time optimal connected with the optimum of computation and the register pressure. Computational cost and consideration of the code size in the register pressure are also added in the paper. The optimum of code motion could be obtained through SpCM algorithm, which considers the code size , in addition to computational optimal and lifetime optimal. The algorithm presented in this paper is the most optimal algorithm in the respect of computation and lifetime, as all the unnecessary code motions are restrained.

키워드 : 코드모션(Code Motion), 희소코드모션(Sparse Code Motion), BCM, LCM

1. 서 론

코드 최적화는 실행시간에 불필요한 값의 재 계산을 피하기 위하여 프로그램을 계산적으로나 수명적으로 최적인 상태로 변환하여 프로그램의 능률을 개선하는 기술이다[1]. 코드 모션은 프로그램의 계산을 적당한 위치에 임시 변수로 대체함으로써 수행된다. 일반적으로 부분 중복제거(PRE : Partial Redundancy Elimination) 알고리즘[2]은 실행시간에 수행되는 계산수의 감소와 불필요한 레지스터 과부하를 피하기 위한 수명의 최소화라는 두 가지 목적을 갖는다[7,8]. 계산적 최적화(computation optimal)는 식을 프로그램내의 안전한 삽입 위치에 초기 설정된 임시변수를 삽입하고, 본래의 식은 임시변수로 재배치함으로써 이루어질 수 있다. 이러한 계산적 최적화 과정에서 본래의 프로그램의 의미를 유지하기 위해 임시변수의 삽입은 안전해야 한다[8]. 수명 최적화(lifetime optimal)는 프로그램의 계산적 최적성이 유지되는 범위 내에서 식을 가능한 한 늦게 두는 것으로써 이 전략에 의해 계산적 최적화 과정에서 도입된 임시변수의 수명을 최소로 줄일 수 있다[7]. 프로그램을 계산적으로나 수명적으로 최적화 하는 기법에는 수식 모션(EM : Expression Motion) 변환[3,5,7]과 배정문 모션(AM : Assignment Motion) 변환[8], 희소 코드 모션(SpCM : Sparse Code Motion)[9] 등이 있다. 본 논문에서는 코드 최적화를 위하여 계산적, 수명적으로 제한이 없는 코드 모션 기법인 희소 코드 모션 알고리즘을 소개한다. 이 알고리즘은 불필요한 레지스터 과부하를 피하기 위해 어떤 불필요한 코드 모션 변환도 억제한다. 본 논문에서 제안하는 알고리즘은 희소 코드 모션 알고리즘으로서 수명적 최적화인 LCM(Lazy Code Motion)을 계산하고 계산적 최적화인 BCM(Busy Code Motion) 술어 계산을 통하여 중복된 배정문을 제거하고 코드 최적화를 이룬다. 본 논문에서는 BCM 알고리즘과 LCM 알고리즘[6,7]을 확장한 희소 코드 모션 알고리즘을 제안하고 제안한 알고리즘의 성능을 평가하고

† 준 회 원 : 관동대학교 대학원 컴퓨터공학과
†† 종신회원 : 관동대학교 컴퓨터공학과 교수
논문접수 : 2004년 10월 11일, 심사완료 : 2004년 12월 20일

자 한다. 또한, 알고리즘의 동작 과정을 구체적으로 제시하여 Knoop의 희소 코드 모션 알고리즘의 이론적 제시에 대한 술어들의 모호한 의미의 방정식 형태인 BCM 알고리즘과 LCM 알고리즘을 명확하게 재구성하여 실제적인 코드 최적화를 수행하고자 한다.

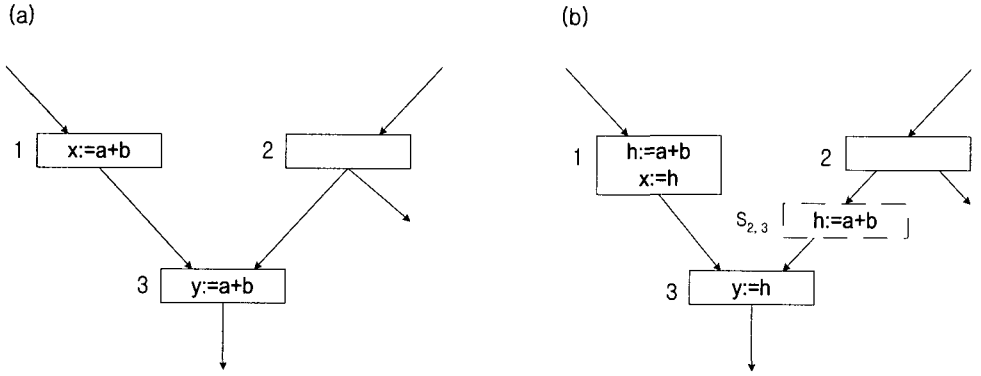
2. 코드 모션

코드 모션의 이론적 배경이 되는 개념으로, 완전한 그래프 구조에서 코드 모션 변환은 임계 가지(critical edge)에 의해 블록화 될 수 있다[3, 7]. 즉, 가지에 의해 하나 이상의 선행자에서 하나 이상의 상속자로 유도된다.

(그림 1) (a)에서 노드 3의 "a+b" 계산은 노드 1의 "a+b" 계산에 대해서 부분 중복이다. 이 부분 중복은 (그림 1) (b)에서처럼 임계가지 (2, 3)에 합성 노드 $S_{2,3}$을 삽입 한 후 "a+b"를 선행하는 노드 1로 이동시켜야만 안전하게 제거된다. 따라서 프로그램에 대한 모든 임계 가지는 합성 노드에 의해 분리되어 있다고 가정한다. 흐름그래프에 합성노드를 추가하면 모든 흐름그래프는 다음을 만족한다.

(1) $\forall n \in N, |pred(n)| \geq 2 \Rightarrow succ(pred(n)) = \{n\}$
(2) $\forall n \in N, |succ(n)| \geq 2 \Rightarrow pred(succ(n)) = \{n\}$

(그림 1) (a)에서 선행자가 2개인 노드 3의 경우 (1)을 만족하지 못한다. 노드 3의 각각의 선행자에 대한 상속자는 노드 3이여야 하지만 선행자 2에 대한 상속자는 노드 3이 아닌 다른 노드일 수 있다. 따라서 노드 2와 3의 임계 가지 (2, 3)에 합성노드 $S_{2,3}$을 삽입한다. (2)도 이와 유사하다.



(그림 1) 임계 가지

코드 모션 변환은 삽입될 수 있는 노드를 결정하는 $Insert_{CM}$과 재배치될 수 있는 노드를 결정하는 $Replace_{CM}$의 두 가지 술어로 표현된다. 흐름 그래프의 각 노드 n은 코드 모션 후보를 포함하고 있는 노드 Computation(n)과 코드 모션 후보에 대한 수정을 포함하지 않은 노드 Transparent(n)으로 정의 할 수 있다[6]. 프로그램의 각 변수의 의미적 보증을 위해 삽입은 안전해야 하며 코드 모션 후보도 올바르게 재배치되어야 한다. 즉, 임시변수의 삽입 위치부터 재배치되는 위치까지의 경로에 계산식에 대한 새로운 값의 도입이 있어서는 안 된다. 일반적인 코드 모션 변환 형태는 먼저 모든 코드 모션 후보 t에 대해 임시 변수 h_{CM}을 도입하여, 삽입은 모든 $n \in N$에 대한 입력 부분의 삽입 위치에 배정문 $h_{CM} := t$를 삽입하고, 모든 $n \in N$에 대한 출력 부분의 삽입 위치에 배정문 $h_{CM} := t$를 삽입한다.

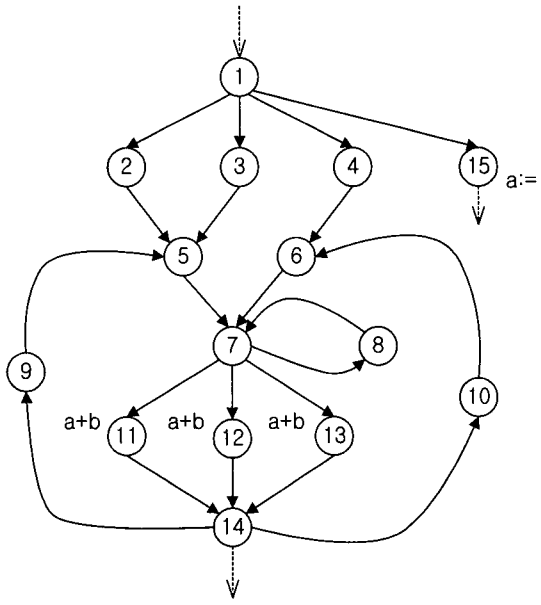
재배치는 입력 부분 재배치 조건을 만족하는 모든 $n \in N$에서 t에 대한 유일한 입력 계산을 h_{CM}으로 재배치하고, 출력 부분 재배치 조건을 만족하는 모든 $n \in N$에서 t에 대한 유일한 출력 계산을 h_{CM}으로 재배치한다[3, 4, 6, 7, 8].

코드 모션의 첫 번째 목표는 모든 프로그램 경로상의 계산수를 최소화하는 것이다. $\forall p \in P[s, e], | \{i|Computation_{CM}(p_i)\} | \leq | \{i|Computation_{CM'}(p_i)\} |$ 일 때, 코드 모션 변환 CM은 코드 모션 변환 CM'보다 계산적으로 더 좋다. 계산적 최적화 코드 모션은 최대 상위 재배치 전략을 이용한다. 이 기법은 흐름그래프의 Safe 부분에서 Earliest 프로그램 위치를 결정짓는다. 코드 모션의 두 번째 목표는 불필요한 코드 모션을 억제하는 것이다. $\forall p \in LtRg(CM)\ \exists q \in LtRg(CM').\ p \subseteq q$ 일 때, 코드 모션 변환 $CM \in \mathfrak{CM}$은 코드 모션 변환 $CM' \in \mathfrak{CM}$보다 수명적으로 좋다. 계산적 최적화 코드 모션 변환이 다른 어떤 계산적 최적화 코드 모션 변환 보다 수명적으로 좋다면 이것이 계산적 최적화다. 불필요한 코드 모션은 과도한 레지스터 과부하의 원인이 될 수 있다. 이것은 임시변수의 수명을 최소화하기 위해 수명적 최적화 코드 모션 변환을 필요로 한다[6, 7].

3. 희소 코드 모션

본 논문에서는 코드 최적화를 위하여 BCM의 계산 능력

과 LCM의 레지스터 과부하뿐만 아니라 코드의 크기를 줄이는 것을 고려하였다. 코드의 계산적 최적화와 수명적 최적화에 이어 코드의 크기를 최소화하는 희소 코드 모션 알고리즘에 의해 코드 모션의 최적화 결과를 얻을 수 있다. 희소 코드 모션 알고리즘은 BCM 알고리즘과 LCM 알고리즘을 확장한다.



(그림 2) 희소코드모션을 위한 예제 흐름그래프

(그림 2)에서 삽입 위치는 {5, 6}이다. 여기서 노드 6으로의 삽입은 노드 4로 이동되고 노드 5로의 삽입은 {2, 3}으로 이동된다면, 계산적으로 최적인 삽입 위치는 {2, 3, 4}이지만 이것은 희소 코드 모션이 아니다.

수명적 최적화를 위한 알고리즘은 LCM 변환을 수행한 후에 계산적 최적화를 수행하고, 최소 결합 집합 STS (Smallest Tight Set)를 계산하여 수명적으로 최적인 삽입 위치를 계산한다.

3.1 LCM

LCM은 우선 배정문이 지연될 수 있는 위치를 결정한다. 이 것은 불필요한 코드 모션을 억제하고 삽입된 계산의 수를 최소화시킨다.

(알고리즘 1)은 배정문이 지연될 수 있는 위치를 결정한다. (알고리즘 1)에서 N-DELAYABLE과 X-DELAYABLE은 배정문 모션에서 사용되는 초기 삽입이 계산적 최적화를 유지하면서 흐름그래프의 마지막 노드까지의 경로상에서 지연될 수 있음을 나타낸다.

DELAYABLE과 LATEST를 분석함으로써 배정문을 프로그램의 어느 위치에 삽입할 것인지 결정한다. 이것은 불필요한 코드 모션을 억제하고 삽입된 계산의 수를 최소화시킨다. (알고리즘 1)은 블록된 배정문을 지연시킬 위치를 결정한다.

```
procedure LCM_DELAYABLE( )
 begin
      for i := 0 to FlowG_node_MAX do
           if (FlowG_node[i] == S_node) then N_DELAYABLE := FALSE;
      for i := 0 to FlowG_node_MAX do
       begin
           for m := DELAY_PRED_START(i) to DELAY_PRED_END(i) do
              Delay_Pred_Sum := Delay_Pred_Sum && X_DELAYABLE[m];
           N_DELAYABLE[i] := Delay_Pred_Sum;
           X_DELAYABLE[i] := FlowG_node[i].IS_INST || N_DELAYABLE[i]
                                 && !FlowG_node[i].USED
                                 && !FlowG_node[i].BLOCKED
       end end;
```

(알고리즘 1) 배정문 지연 알고리즘

```
procedure LCM_LATEST( )
 begin
     for i := 0 to FlowG_node_MAX do
       begin
           N_LATEST[i] := FALSE;
           X_LATEST[i] := FALSE
       end;
     for i := FlowG_node_MAX to 0 do
       begin
           for m := LATE_SUCC_START(i) to LATE_SUCC_END(i) do
              Late_Succ_Sum := Late_Succ_Sum || !N_DELAYABLE[m];
           if (N_DELAYABLE[i]) then
            N_LATEST[i] := N_DELAYABLE[i] &&
              (FlowG_node[i].USED || FlowG_node[i].BLOCKED);
           if (X_DELAYABLE[i]) then
            X_LATEST[i] := X_DELAYABLE[i] && Late_Succ_Sum
       end
 end;
```

(알고리즘 2) 블록된 배정문 지연 알고리즘

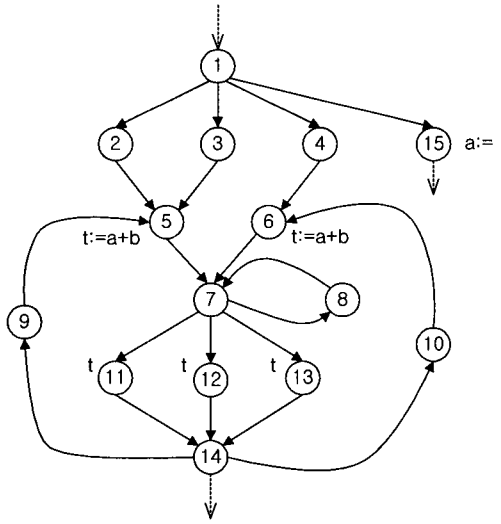
```
procedure LCM_ISOLATED( )
 begin
     X_ISOLATED_MAX := TRUE;
       for i := FlowG_node_MAX to 0 do
       begin
          for m := ISOL_Succ_Start[i] to ISOL_Succ_End(i) do
             begin ISOL_Succ := N_EARLIEST[m] ||
              (!(FlowG_node[m].N_COMP) && N_ISOLATED[m]);
              ISOL_Succ_Sum := ISOL_Succ_Sum || ISOL_Succ
             end;
          N_ISOLATED[i] := X_EARLIEST[i] || X_ISOLATED[i];
          X-ISOLATED[i] := ISOL_Succ_Sum
 end end;
```

(알고리즘 3) 독립 수식 위치 결정 알고리즘

(알고리즘 2)에서 N-LATEST와 X-LATEST는 배정문이 지연될 수 있는 위치들 중에서 배정문이 블록되거나 사용된 위치를 결정한다. LATEST는 DELAYABLE의 값이 참일 경우에만 참일 수 있으므로 DELAYABLE의 값이 참인 노드에 대해서만 LATEST를 계산한다. LATEST의 위치를 결정하기 위해서는 그 노드에 대한

상속자의 입력 부분이 배정문을 지연시킬 수 있는 위치인지 아닌지에 대한 정보를 이용하게 된다. 따라서 LATEST는 마지막 노드에서 시작 노드로 제어흐름 반대 방향으로 분석해 나간다. 또한, LATEST의 위치 결정은 DELAYABLE이 참인 경우에 한해서 계산된다. 프로그램의 실행 시간을 향상시키는 코드 모션일지라도 불필요한 임시변수 초기화를 실행할 수 있다. 이러한 결점을 피하기 위해서 (알고리즘 3)에서 독립 수식을 결정하여 임시변수의 삽입을 제한한다.

(그림 2)에 LCM 변환을 적용하면 (그림 3)과 같다.



(그림 3) LCM 변환의 결과

3.2 BCM

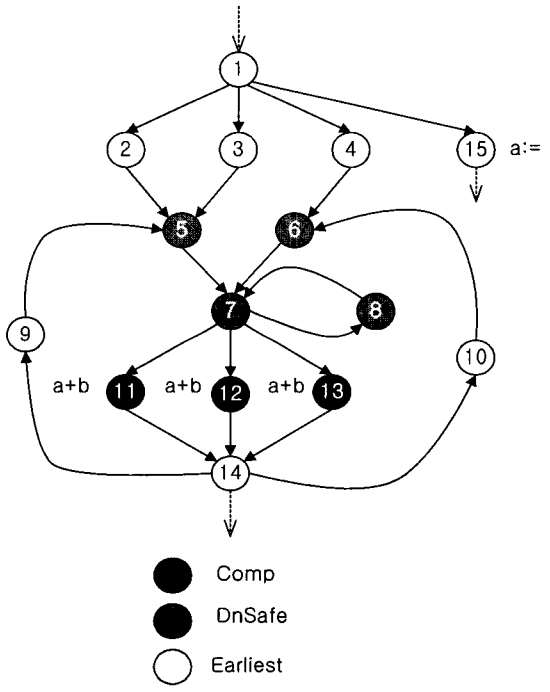
BCM에서 프로그램의 각 변수의 의미적 보증을 위해 삽입은 안전해야 하며 코드 모션 후보도 옳게 재배치되어야 한다.

```
procedure BCM_SAFE( )

 begin
  for i := 0 to FlowG_node_MAX do
    begin  if (FlowG_node[i] == E_node) then
                 FlowG_node[i].X_D_SAFE := FALSE;
           else if (FlowG_node[i] == S_node) then
                 FlowG_node[i].N_U_SAFE := FALSE
    end;
  for i := FlowG_node_MAX downto 1 do
    begin  for m := SAFE_Succ_Start(i) to SAFE_Succ_End(i) do
           SAFE_Succ_Sum := SAFE_Succ_Sum &&
           FlowG_node[m].N_D_SAFE;)
           FlowG_node[i].N_D_SAFE := FlowG_node[i].N_COMP ||
           (FlowG_node[i].TRANSP && FlowG_node[i].X_D_SAFE;
           FlowG_node[i].X_D_SAFE := FlowG_node[i].X_COMP ||
           SAFE_Succ_Sum
    end;
  for i := 1 to FlowG_node_MAX do
    begin  for m := SAFE_Pred_Start(i) to SAFE_Pred_End(i) do
        begin  SAFE_Pred := FlowG_node[m].X_COMP ||
                     FlowG_node[m].X_U_SAFE;
           SAFE_Pred_Sum := SAFE_Pred_Sum && SAFE_Pred
        end;
        FlowG_node[i].N_U_SAFE := SAFE_Pred_Sum;
        FlowG_node[i].X_U_SAFE := FlowG_node[i].TRANSP &&
           (FlowG_node[i].N_COMP || FlowG_node[i].N_U_SAFE)
    end
 end;
```

(알고리즘 4) 안전한 삽입 위치 결정 알고리즘

```
procedure BCM_EARLIEST( )
 begin
  for i := 0 to FlowG_node_MAX do
   begin
     for m := 0 to Pred_node do
        begin
           EARL_Pred[m] := !(FlowG_node[m].X_U_SAFE
           || FlowG_node[m].X_D_SAFE);
           EARL_Pred_Sum := EARL_Pred_Sum || EARL_Pred[m]
        end;

        if (FlowG_node[i].N_D_SAFE) then
          N_EARLIEST[i] := FlowG_node[i].N_D_SAFE &&
                           EARL_Pred_Sum;
        if (FlowG_node[i].X_D_SAFE) then
          X_EARLIEST[i] := FlowG_node[i].X_D_SAFE &&
                           !(FlowG_node[i].TRANSP)
   end
 end;
```

(알고리즘 5) 가장 이른 삽입 위치 결정 알고리즘



(그림 4) BCM 술어들의 계산 결과

3.3 이분 그래프

프로그램 내의 안전한 삽입 위치를 결정하는 알고리즘은 (알고리즘 4)와 같다. (알고리즘 4)는 임시변수의 삽입 위치부터 재배치되는 위치까지의 경로에 계산식에 대한 새로운 값의 도입이 있어서는 안 됨을 나타낸다

코드 모션에서 삽입 위치는 안전한 삽입 위치 중에서 가장 이른 삽입위치에 임시변수 t를 삽입한다. (알고리즘 5)는 가장 이른 삽입 위치 Earliest를 결정한다.

(그림 2)에 BCM 술어들을 적용하면 (그림 4)와 같다.

무향 그래프 (N, E)에서 N는 노드를 E는 가지를 의미한다. $n \in N$인 노드의 이웃하는 노드 $\Gamma(n)$은 $\Gamma(n) =_{df} \{w \mid \{n, w\} \in E\}$로 정의된다. E에 포함되는 모든 가지 e에 대해 $N = S \cup T$이고 $e \cap S \neq \emptyset \wedge e \cap T \neq \emptyset$인 노드 S와 T의 집합이 두 개로 분리된다면, 무향 그래프 (N, E)는 (S∪T, E)의 두 부분으로 나누어 질 수 있다. (알고리즘 6)은 이분 그래프(bipartite graph)를 작성하는 알고리즘이다. (알고리즘 6)에서 T_{DS}를 상위 계층이라 하고, S_{DS}를 하위 계층이라 한다.

(그림 4)의 흐름 그래프에 (알고리즘 6)을 적용하여 S_{DS}와 T_{DS}를 계산하여 구성된 이분 그래프는 (그림 5)와 같다.

```
procedure Bipar_Graph()
  begin
        SDS := DnSafe ∪ !(UpSafe ∪ Earliest);
        TDS := DnSafe ∪ !(UpSafe ∪ Comp);
        for n:=0 to n<SDS_Max do
              Con_Edge(TDS[m], DnSafe(pred(SDS[n]))
  end;
```

(알고리즘 6) 이분 그래프 작성 알고리즘

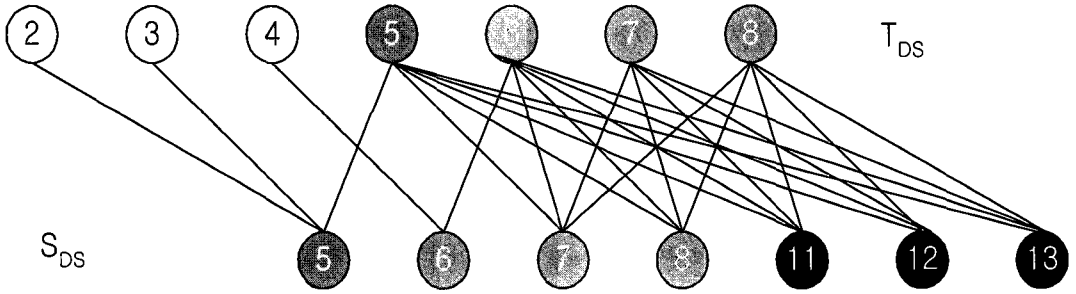
〈계층 구성 결과〉

S_{DS} = {5, 6, 7, 8, 11, 12, 13}

T_{DS} = {2, 3, 4, 5, 6, 7, 8}

〈가지 구성 결과〉

ρ(pred(5)) = ρ{2, 3} = {2, 3, 5}
ρ(pred(6)) = ρ{4} = {4, 6}
ρ(pred(7)) = ρ{5, 6, 8} = {5, 6, 7, 8}
ρ(pred(8)) = ρ{7} = {5, 6, 7, 8}
ρ(pred(11)) = ρ{7} = {5, 6, 7, 8}
ρ(pred(12)) = ρ{7} = {5, 6, 7, 8}
ρ(pred(13)) = ρ{7} = {5, 6, 7, 8}

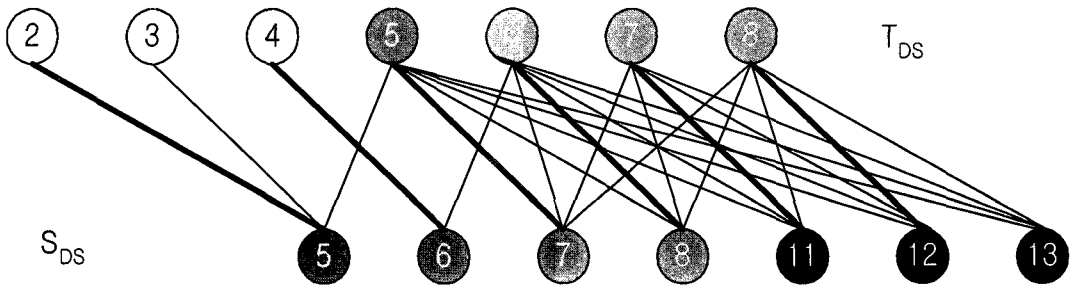


(그림 5) 예제 흐름그래프에 대한 이분 그래프

3.4 매칭과 최대 결합 집합

M에 속한 가지 e_1, e_2이 $e_1 \cap e_2 = \phi$라면, $M \subseteq E$인 가지들의 집합을 매칭(matching)이라 한다. 어떤 $e \in M$에 대하여 $n \in e$라면, 노드 n는 M에 의해 매치 된다. 어떤 매칭 $M' \subseteq E$에 대해 $|M| \geq |M'|$라면, M은 최대 매칭(maximum matching)이다.

(그림 6)의 굵은 가지는 (그림 4)의 최대 매칭을 나타낸다.



(그림 6) 이분 그래프에 대한 최대 매칭

(그림 6)의 이분 그래프에 대한 최소 결합 집합을 (알고리즘 8)로 계산한 결과는 $T_S(S) = \{7, 8, 11, 12, 13\}$ 이다. 따라서, 삽입 위치 $Ins_{SpCMco} = \{5, 6\}$이다. (그림 3)에 수명적 최적화 $SpCM_{CO}$를 적용하면 (그림 7)과 같다.

최대 매칭은 이분 그래프의 최대 결합 집합을 계산하는데 사용된다. (알고리즘 7)은 이분 그래프 B_{DS}와 최대 매칭 M을 입력받아 최대 결합 집합 LTS(Largest Tight Set)를 구성한다.

(그림 6)의 이분 그래프에 대한 최대 결합 집합을 (알고리즘 7)로 계산한 결과는 $T_L(S) = \{6, 7, 8, 11, 12, 13\}$이다. 따라서, 삽입위치 $INS_{SpCMprofit} = \{4,5\}$이다.

```
procedure LTS()
begin
for i := 0 to Un_Match_T_Max do
  begin
          x := Un_Match_T[i];
          Un_Match_T[] := Un_Match_T[] - {x};
           if x∈S then
                  begin
                  S_Match := S_Match - {x};
                  Un_Match_T[] := Un_Match_T[] ∪ Max_Match(x, y);
                  end;
                else
                  Un_Match_T := Un_Match_T ∪ (Γ(x) ∩ S_Match);
          TL(S) := S_Match
  end;
   Ins SpCM Profit = (DnComp⋃Γ(T L(S))) − T L(S)
end;
```

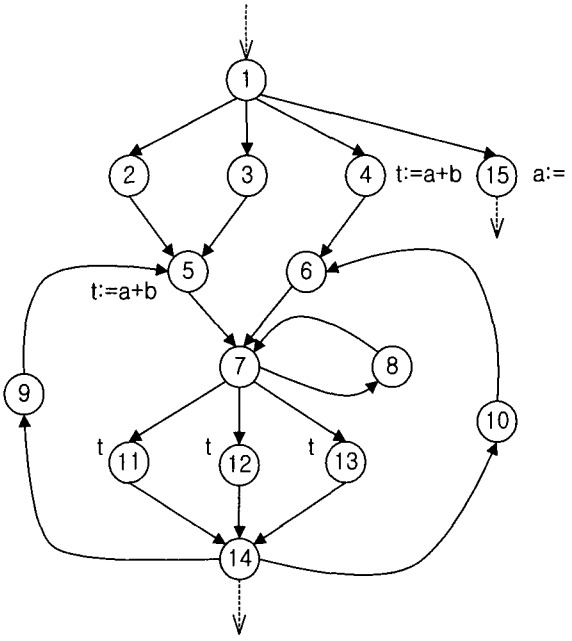
(알고리즘 7) 최대 결합 집합 알고리즘

3.5 최소 결합 집합

희소 코드 모션은 계산적으로 최적인 코드 모션뿐만 아니라 수명적으로도 최적인 코드 모션이다. 최소 결합 집합의 계산은 수명적으로 최적인 코드 모션을 수행한다.

```
procedure STS()
  begin
    for i := 0 to Un_Match_S_Max do
      begin
          x := Un_Match_S_[i];
          Un_Match_S := Un_Match_S[] - {x};
           if x∈S then
             begin
             S_Match := S_Match ∪ {x};
             Un_Match_S[] := Un_Match_S[] ∪ (Γ(x) ∩ S_Match);
             end;
            else
          Un_Match_S := Un_Match_S ∪ Max_Match(x, y);
      TS(S) := S_Match
    end;
    Ins SpCM co = (DnComp⋃Γ(T S(S))) − T S(S)
end
```

(알고리즘 8) 최소 결합 집합 알고리즘



(그림 7) 최적의 희소 코드 모션 결과

(알고리즘 8)은 최소 결합 집합을 계산한다.

3.6 성능 분석

본 논문에서 제안된 희소 코드 모션 알고리즘의 성능 분석 결과는 다음과 같다. 성능 분석은 Pentium-Ⅳ 1.7의 Windows 2000 환경에서 C 언어를 사용하여 성능 분석을 하였으며, (그림 2)의 흐름 그래프에 제안한 알고리즘을 적용한 결과는 <표 1>과 (그림 8)과 같다.

제안한 알고리즘을 적용하기 전과 적용한 후의 흐름 그래프를 비교한 결과, 프로그램 내에 복잡한 반복문이 많고 반복문 내의 중복 코드가 많을수록 제안한 알고리즘을 적용한 경우의 효과가 크다는 것을 알 수 있다.

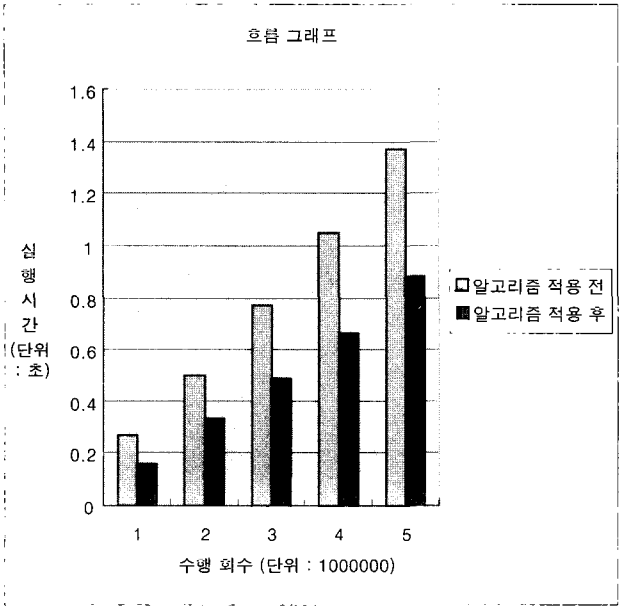
〈표 1〉 흐름 그래프의 알고리즘 수행 결과

(단위 : 백만번)

구분 / 수행회수	적용 전	적용 후
1	0.27	0.16
2	0.50	0.33
3	0.77	0.49
4	1.05	0.66
5	1.37	0.88

수행 횟수가 4인 경우에 알고리즘 적용 후의 실행시간이 적용 전에 비해 약 38% 감소됐다. 따라서, 제안된 알고리즘

은 프로그램의 구조가 복잡하고 루프의 횟수가 많을수록 적용 효과가 높다는 것을 알 수 있다.



(그림 8) 제안한 알고리즘 수행 결과

4. 결 론

일반적으로, 코드 모션 알고리즘은 계산의 최적화와 레지스터 과부하에 관련된 실행시간 최적화의 관점에서 시작된다. 수식 모션은 코드 모션의 후보가 되는 식을 프로그램내의 안전한 위치에 임시변수를 이용하여 초기화하고 그 후보가 되는 식이 사용되는 위치를 임시변수로 재배치한다. 수식 모션을 포함하는 배정문 모션은 프로그램 내의 모든 배정문에 임시 변수를 도입하고 배정문 끌어올리기와 중복된 배정문 제거 단계를 수행한다. 본 논문에서는 계산 능력과 레지스터 과부하에 더하여 코드의 크기를 고려하는 부분을 추가하였다. 코드의 계산적 최적화와 수명적 최적화에 이어 코드의 크기를 고려하는 희소 코드 모션 알고리즘에 의해 코드 모션의 최적화 결과를 얻을 수 있다. 본 논문에서 제안한 알고리즘은 모든 불필요한 코드 모션을 억제시키기 때문에 계산적으로나 수명적으로 최적인 알고리즘이다. 또한, 제안된 알고리즘의 동작 과정을 구체적으로 제시하였다. 제안한 알고리즘의 성능평가를 해본 결과 프로그램의 불필요한 재계산이나 재실행을 하지 않도록 함으로서 기존의 방법보다 프로그램의 능률 및 실행시간을 향상시켰다. 또한, 희소 코드 모션 알고리즘에서 BCM 알고리즘은 계산적으로 최적의 코드 모션을 수행하며, LCM 알고리즘은 레지스터 과부하를 감소시켰다. 향후 연구 방향으로는 코드의 크기를 고려한 코드 모션 알고리즘의 확장과 이를 통해 병렬 프로그램에 적용하여 재구성해 보는 것이다.

참 고 문 헌

[1] Aho, A. V., Sethi, R., and Ullman. J. D., "*Compilers Principles, Techniques, and Tools*", Addison-Wesley publishing Co., 1986.

[2] Briggs, P and Cooper, K. D., "Effective partial redundancy elimination", *In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'94, ACM SIGPLAN Notices*, Vol. 29, No.6, pp.159-170, Orlando, FL, June, 1994.

[3] Dhamdhere, D. M., "A fast algorithm for code movement optimization", *ACM SIGPLAN Notices*, Vol.23, No. 10, pp.172-180, 1998.

[4] Dhamdhere, D. M. and Patil. H., "An elimination algorithm for bidirectional data flow problems using edge placement", *ACM Transactions of Programming languages and Systems*, Vol.15, No.2, pp.312-336, April, 1993.

[5] Drechsler, K. H. and Stadel, M. P., "A variation of Knoop, Rüthing and Steffen's lazy code motion", *ACM SIGPLAN Notices*, Vol.28, No.5, pp.29-38, 1993.

[6] Knoop, J., Rüthing, O. and Steffen, B., "Lazy code motion", *In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92, ACM SIGPLAN Notices*, Vol.27, No.7, pp.224-234, San Francisco. CA, June, 1992.

[7] Knoop, J., Rüthing, O. and Steffen, B., "Optimal Code motion: Theory and Practice", *ACM Transactions on Programming Languages and Systems*, Vol.16, No.4, pp.1117-1155, 1994.

[8] Knoop, J., Rüthing, O. and Steffen, B., "The Power of Assignment Motion", *Proceedings of the Conference on Programming Language Design and Implementation*, Vol.30, No.6, pp.233-245, 1995.

[9] Knoop, J. and Steffen, B., "Sparse Code Motion", *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming languages*, pp.170-183, January, 2000.

신 현 덕

e-mail : ubhd@kd.ac.kr

1998년 관동대학교 전자계산공학과(공학사)

2000년 관동대학교 대학원 전자계산공학과 (공학석사)

2003년 관동대학교 대학원 전자계산공학과 박사 수료

관심분야 : 컴파일러, 병렬 컴파일러, 프로그래밍 언어, 코드 최적화

유 희 종

e-mail : blackyu@kd.ac.kr

1999년 관동대학교 전자계산공학과(공학사)

2003년 관동대학교 대학원 전자계산공학과 (공학석사)

2003년~현재 관동대학교 대학원 전자계산공학과 박사과정

관심분야 : 인공지능, 알고리즘, 코드 최적화

안 희 학

e-mail : hhahn@kd.ac.kr

1981년 숭실대학교 전자계산학과(공학사)

1983년 숭실대학교 대학원 전자계산학과 (공학석사)

1994년 숭실대학교 대학원 전자계산학과 (공학박사)

1994년~1996년 관동대학교 전자계산소 소장

2001년~2003년 관동대학교 전산정보원장

1984년~현재 관동대학교 공과대학 컴퓨터학부 교수

관심분야 : 컴파일러, 병렬컴파일러, 프로그래밍 언어, 함수언어 오토마타 등