불필요한 코드모션 억제를 위한 배정문 모션 알고리즘

신 현 덕† · 안 희 학††

요 약

본 논문에서는 코드 최적화를 위하여 계산적으로나 수명적으로 제한이 없는 배정문 모션 알고리즘을 제안한다. 이 알고리즘은 지나친 레지스터의 사용을 막기 위하여 불필요한 코드 모션을 억제한다. 본 논문은 최종 최적화단계가 추가된 배정문 모션 알고리즘을 제안한다. 또한 기존 알고리즘의 술어의 의미가 명확하지 않은 것을 개선하였고 노드 단위 분석과 명령어 단위 분석을 혼용했기 때문에 발생하는 모호함도 개선하였다. 따라서 제안한 알고리즘은 불필요하게 중복된 수식이나 배정문의 수행을 피하게 함으로써, 프로그램의 불필요한 재계산이나 재실행을 하지 않게 하여 프로그램의 능률 및 실행시간을 향상시킨다.

An Assignment Motion Algorithm to Suppress the Unnecessary Code Motion

Hyun-Deok Shin† · Heui-Hak Ahn††

ABSTRACT

This paper presents the assignment motion algorithm unrestricted for code optimization computationally. So, this algorithm is suppressed the unnecessary code motion in order to avoid the superfluous register pressure. we propose the assignment motion algorithm added to the final optimization phase. This paper improves an ambiguous meaning of the predicate. For mixing the basic block level analysis with the instruction level analysis, an ambiguity occurred in knoop's algorithm. Also, we eliminate an ambiguity of it. Our proposal algorithm improves the runtime efficiency of a program by avoiding the unnecessary recomputations and reexecutions of expressions and assignment statements.

키워드 : 코드 최적화(code optimization), 배정문 모션(assignment motion), 코드모션(code motion)

1. 서 론

코드 최적화는 실행시간에 불필요한 값의 재 계산을 피하기 위해 프로그램을 계산적으로나 수명적으로 최적인 상태로 변환하여 프로그램의 능률을 개선하는 기술이다[1].

프로그램을 계산적으로나 수명적으로 최적화하는 기법에는 수식 모션(EM : Expression Motion) 변환[2-6]과 수식 모션을 포함하는 배정문 모션(AM : Assignment Motion) 변환[7-10]이 있다.

수식 모션은 코드 모션의 후보가 되는 식을 프로그램내의 안전한 위치에 임시변수를 이용하여 초기화하고 그 후보가 되는 식이 사용되는 위치를 임시변수로 재배치한다[5]. 배정문 모션은 프로그램내의 모든 배정문에 임시변수를 도입하고(이 단계에서 배정문 모션은 수식 모션을 포함하게 된다.) 배정문 끌어올리기와 중복 배정문 제거 단계를 수행한다[10].

† 준 회 원 : 관동대학교 대학원 전자계산공학과
†† 종신회원 : 관동대학교 컴퓨터공학과 교수
논문접수 : 2000년 8월 24일, 심사완료 : 2001년 3월 11일

2. 관련 연구

수식모션에 의한 프로그램 최적화는 지금까지 많이 연구되었다. Dhamdhere에 의해 코드 모션 최적화를 위한 알고리즘이 제안되었으며[2] Drechsler와 Stadel에 의해 부분 중복식 제거에 의한 전역적 최적화 기법이 제안되었다[4]. 또한, Knoop, Rüthing, Steffen에 의해 죽은 코드 제거 및 수식모션에 의한 최적화 코드 모션 알고리즘이 제안되었다[5, 10-12].

이처럼 프로그램 최적화에 대해 철저하게 연구된 수식모션과 대조적으로 배정문 모션은 지금까지 별로 연구되지 않았다. Dhamdhere가 배정문 모션과 관련이 있는 논문을 발표했으나[7-9] 부분적으로 중복된 배정문을 제거하는 알고리즘에서 배정문에 대한 끌어올리기를 제한하기 때문에 최적인 변형 결과를 얻기는 불가능하다.

또한, Dhamdhere는 레지스터 할당에 배정문을 끌어올리고 끌어내리는 기술들의 적용을 소개하였고[7, 8] Knoop은 병렬 프로그램에서의 코드 모션에 대한 이론을 정립하였다.

1995년에 Knoop은 부분적 중복 식 제거(PREE : Partially Redundant Expression Elimination)를 위한 수식 모션 변환과 수식 모션을 포함하는 배정문 모션 변환 방정식 알고리즘을 소개했다[10].

본 논문은 Knoop이 제시한 방정식 알고리즘[10]을 개선시킨다. Knoop이 제시한 방정식 알고리즘은 초기 설정 단계에서 도입된 v := h_ε형태에 의한 h_ε의 사용을 임시변수 사용횟수에 포함시켰으며 이 불필요한 코드 모션은 레지스터 압박을 초래할 수 있다. 따라서 알고리즘에 최종 최적화 단계를 추가하여 초기 설정 단계에서 v := h_ε의 형태로 도입된 h_ε은 사용횟수에서 제외한다.

제안된 알고리즘은 수식모션을 포함하는 배정문 모션 변환 알고리즘으로서 모든 배정문에 임시 변수를 도입하고 최대 상위(as-early-as-possible) 재 배치 전략으로 가능한 모든 배정문을 끌어올려서 중복된 배정문을 제거한다. 이 단계는 계산적 최적화 단계이며 이 단계에서 불필요한 코드 모션이 일어날 수 있다. 따라서, 최대 하위(as-late-as-possible) 재 배치 전략을 이용하여 불필요한 코드 모션을 억제한다.

또한, 알고리즘의 동작 과정을 구체적으로 제시하여 Knoop의 방정식 알고리즘의 이론적 제시에 대한 술어들의 명확하지 않은 의미와 노드 단위 분석과 명령어 단위 분석을 혼용하여 발생하는 모호함도 개선하고자 한다.

3. 수식 모션 변환

수식 모션은 안전한 삽입위치 중에서 가장 이른 삽입위치에 t를 삽입한다. 따라서 ∀n∈N에 대한 Earliest(n)은 (정의 1)과 같다[5, 11, 12].

(정의 1) Earliest

$$Earliest(n) = Safe(n) \wedge \left(\begin{cases} true & \\ \bigvee_{m \in pred(n)} & \neg Transparent(m) \end{cases} \wedge \begin{cases} \text{if } n = s & \\ \neg Safe(m) & otherwise \end{cases} \right)$$

수식 모션에서의 초기화는 계산적 최적성이 유지되는 한 시작 노드에서 마지막 노드까지의 경로상에서 지연될 수 있다는 특징으로 Delayability를 정의하고, 불필요한 코드 모션을 억제하기 위해서 실행 시간을 향상시키지 못하는 코드 모션은 억제되어야 한다는 특징으로 Latest를 정의한다[5, 11, 12].

(정의 2) Delayability

$$\forall n \in N,\ Delayed(n) \Leftrightarrow_{df} \forall p \in P[s, n)\ \exists i \le \lambda_p.\ Earliest(p_i) \wedge \neg Computation^{\exists}(p[i, \lambda_p[)$$

(정의 3) Latest

$$\forall n \in N,\ Latest(n) =_{df} Delayed(n) \wedge \left(Comp(n) \vee \bigvee_{m \in succ(n)} \neg Delayed(m) \right)$$

프로그램의 실행 시간을 향상시키는 코드 모션일지라도 불필요한 임시변수 초기화를 실행할 수 있다는 특징으로 Isolated를 정의한다[5, 11, 12].

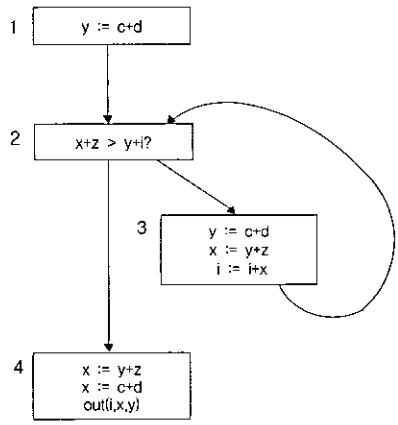
(정의 4) Isolation

$$\forall CM \in \mathfrak{CM}\ \forall n \in N,$$
$$Isolated_{CM}(n) \Leftrightarrow_{df} \forall p \in P[n, e)\ \forall 1 < i \le \lambda_p,\ Replace_{CM}(p_i) \Rightarrow Insert^{\exists}_{CM}(p)1, i))$$

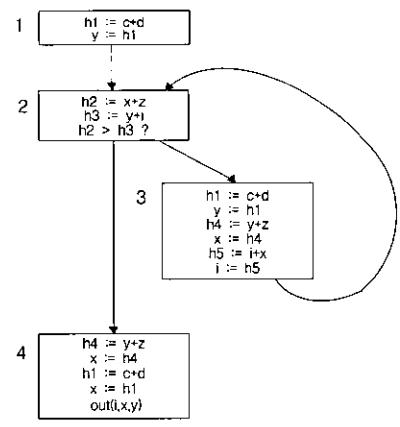
4. 코드 최적화 알고리즘

4.1 초기 설정 단계

초기 설정 단계에서는 임시 기억장소를 도입해서 프로그램 내의 모든 배정문을 분해한다. 모든 배정문 x := t를 배정문 h_t := t; x := h_t로 대체한다. h_t는 t항을 보관하는 유일한 임시 기억장소이다. 초기 설정 단계에서 배정문 모션은 수식 모션을 포함하게 된다. (그림 1)에 초기 설정 단계를 적용하면 (그림 2)와 같다.



(그림 1) 예제 흐름그래프



(그림 2) 초기 설정 단계 수행 결과

4.2 배정문 모션 단계

4.2.1 배정문 끌어올리기 알고리즘

배정문 끌어올리기 알고리즘은 프로그램의 의미를 유지하면서 배정문을 본래의 위치로부터 프로그램의 가장 앞부분으로 끌어올리는 것이다. 끌어올리기 후보는 배정문 x := t에서 t의 피연산자에 대한 수정뿐만 아니라 x에 대한 어떠한 수정도 없어야 하고 배정문의 현재 위치부터 끌어올릴 위치 사이에서 사용된 적이 없는 배정문이어야 한다.

(알고리즘 1)에서 끌어올리기 후보를 나타내는 술어 N-HOISTABLE과 X-HOISTABLE은 α의 끌어올리기 후보들을 기본 블록 n의 입력이나 출력 부분까지 각각 이동시킬 수 있다는 것을 의미한다.

(그림 2)에서 배정문 h4 := y+z에 대한 HOISTABLE은 (알고리즘 1)에 의해 다음과 같이 계산된다.

$$N\text{-}HOISTABLE_4 = T \vee F \wedge \neg F = T$$
$$X\text{-}HOISTABLE_4 = F$$
$$N\text{-}HOISTABLE_2 = F \vee T \wedge \neg F = T$$
$$X\text{-}HOISTABLE_2 = T \wedge T = T$$
$$N\text{-}HOISTABLE_3 = T \vee T \wedge \neg F = T$$
$$X\text{-}HOISTABLE_3 = T$$
$$N\text{-}HOISTABLE_1 = F \vee T \wedge \neg T = F$$
$$X\text{-}HOISTABLE_1 = T$$

```
procedure Find_HOISTABLE( )
 begin
    for i := 0 to FlowG_node_MAX do
          if (FlowG_node(i) == E_node) then
                              X_HOISTABLE(i) := FALSE;
    for i := FlowG_node_MAX to 1 do
     begin
          for m := HOIST_SUCC_START(i) to
                   HOIST_SUCC_END(i) do
                   Hoist_Succ_Sum := Hoist_Succ_Sum &&
                                     N_HOISTABLE(m);
          N_HOISTABLE(i) := FlowG_node(i).LOC_HOISTABLE ||
                            X_HOISTABLE(i) &&
                            !FlowG_node(i).LOC-BLOCKED;
          X_HOISTABLE(i) := Hoist_Succ_Sum
     end
 end;
```

(알고리즘 1) 배정문 끌어올리기 알고리즘

끌어올릴 수 있는 노드의 위치를 결정하기 위해서는 그 노드에 대한 상속자의 입력 부분이 끌어올릴 수 있는 위치인가 아닌가의 정보를 이용하게 된다. 따라서 HOISTABLE은 마지막 노드에서 시작 노드까지 제어흐름의 반대 방향으로 분석해 나간다.

4.2.2 배정문 삽입 알고리즘

(알고리즘 2)에서 술어 N-INSERT와 X-INSERT는 특별한 프로그램 지점에 삽입되어야하는 모든 배정문은 블록되지 않아야 한다는 것을 의미한다.

N-INSERT(α)(또는 X-INSERT(α))가 만족되면 배정문 패턴 α의 실제 값을 기본 블록 n의 입력(또는 출력)부분에 삽입한다.

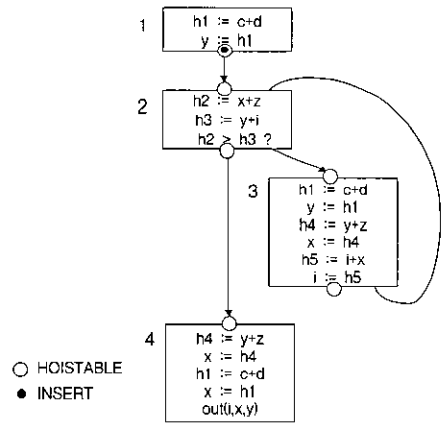
배정문 h4 := y+z에 대한 INSERT는 (알고리즘 2)에 의해 다음과 같이 계산된다.

$$N\text{-}INSERT_1 = F \wedge \neg T = F$$
$$X\text{-}INSERT_1 = T \wedge T = T$$
$$N\text{-}INSERT_2 = T \wedge (\neg T \vee \neg T) = F$$
$$X\text{-}INSERT_2 = T \wedge F = F$$
$$N\text{-}INSERT_3 = T \wedge \neg T = F$$
$$X\text{-}INSERT_3 = T \wedge F = F$$
$$N\text{-}INSERT_4 = T \wedge \neg T = F$$
$$X\text{-}INSERT_4 = F \wedge F = F$$

```
procedure Find_INSERT( )
 begin
    for i := 0 to FlowG_node_MAX do
     begin
          N_INSERT(i) := FALSE;
          X_INSERT(i) := FALSE
     end;
    for i := 0 to FlowG_node_MAX do
     begin
          for m := INS_PRED_START(i) to
                   INS_PRED_END(i) do
                   Ins_Pred_Sum := Ins_Pred_Sum ||
                   !X_HOISTABLE(m);
          if (N_HOISTABLE(i)) then
             N_INSERT(i) := N_HOISTABLE(i) && Ins_Pred_Sum
          if (X_HOISTABLE(i)) then
             X_INSERT(i) := X_HOISTABLE(i) &&
                            FlowG_node(i).LOC_BLOCKED
    end
 end;
```

(알고리즘 2) 배정문 삽입 알고리즘

배정문 h4 := y+z에 대한 HOISTABLE과 INSERT를 계산한 결과는 (그림 3)과 같다.



(그림 3) HOISTABLE과 INSERT의 계산 결과

배정문 패턴 α의 끌어올릴 위치를 결정하는 INSERT는 그 노드에 대한 선행자의 출력 부분이 끌어올릴 수 있는 위

치인지 아닌지에 대한 정보를 이용하게 된다. 따라서 INSERT는 시작노드에서 마지막 노드로 제어흐름 방향으로 분석해 나간다.

4.2.3 중복 배정문 제거 알고리즘

(알고리즘 3)에서 N-ELIMINATION과 X-ELIMINATION은 기본 블록 n의 입력이나 출력 부분에서 중복인 배정문 α를 제거한다.

```
procedure Find_ELIMINATION( )
 begin
    for i := 0 to FlowG_node_MAX do
     begin
           N_ELIMINATION(i) := FALSE;
           X_ELIMINATION(i) := FALSE
     end;
    for i := 0 to FlowG_node_MAX do
     begin
           if (N_REDUNDANT(i)) then
             N_ELIMINATION(i) := N_REDUNDANT(i) &&
                                 FlowG_node(i).EXECUTED;
           if (X_REDUNDANT(i)) then
             X_ELIMINATION(i) := X_REDUNDANT(i) &&
                                 FlowG_node(i).EXECUTED
     end
 end;
```

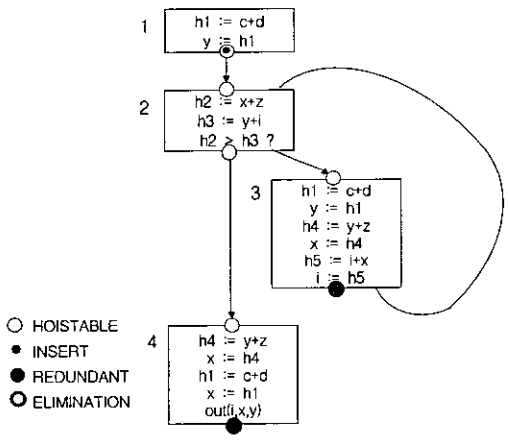
(알고리즘 3) 중복 배정문 제거 알고리즘

ELIMINATION은 REDUNDANT의 값이 참일 경우에만 참일 수 있으므로 REDUNDANT의 값이 참인 노드에 대해서만 ELIMINATION을 계산한다.

배정문 h4 := y + z에 대한 ELIMINATION은 (알고리즘 3)에 의해 다음과 같이 계산된다.

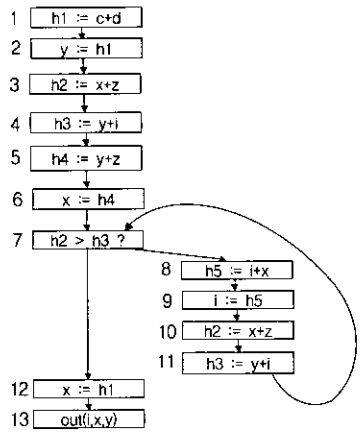
$$X-ELIMINATION_3 = T \wedge T = T$$
$$X-ELIMINATION_4 = T \wedge T = T$$

배정문 h4 := y + z에 대한 REDUNDANT와 ELIMINATION을 (알고리즘 3)으로 계산한 결과는 (그림 4)와 같다.



(그림 4) REDUNDANT와 ELIMINATION의 계산 결과

(그림 2)에 배정문 모션 단계를 적용한 결과는 (그림 5)와 같다.



(그림 5) 배정문 모션 단계 수행 결과

4.3 불필요한 코드 모션 재구성 단계

불필요한 코드 모션 재구성 단계에서는 프로그램의 의미를 유지하면서 h_ε := (ε는 수식 패턴) 형태의 모든 배정문을 프로그램의 가장 뒷부분으로 옮기고, h_ε가 배정문의 실제 값 이후에 많아야 한번 이용되는 모든 실제 값을 제거한다.

4.3.1 배정문 지연 알고리즘

(알고리즘 4)에서 N-DELAYABLE과 X-DELAYABLE은 배정문 모션에서 사용되는 초기화의 삽입이 계산적 최적화를 유지하면서 흐름그래프의 마지막 노드까지의 경로상에서 지연될 수 있음을 나타낸다.

```
procedure Find_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 4) 배정문 지연 알고리즘

DELAYABLE과 LATEST를 분석함으로써 배정문을 프로그램의 어느 위치에 삽입 할 것인지 결정한다. 이것은 불필요한 코드 모션을 억제하고 삽입된 계산의 수를 최소화시킨다.

지연될 수 있는 위치 중에서 프로그램의 가장 늦은 위치를 결정하기 위해서는 그 노드에 대한 선행자의 출력 부분이 배정문을 지연시킬 수 있는 위치인가 아닌가에 대한 정

보를 이용하게 된다. 따라서 DELAYABLE은 시작 노드에서 마지막 노드로, 제어흐름 방향으로 분석해 나간다.

4.3.2 유용한 코드 모션 알고리즘

(알고리즘 5)에서 N-USABLE과 X-USABLE은 배정문이 노드 n의 입력이나 출력 부분에서 사용되었는가를 계산해서 이 코드 모션이 유용한 코드 모션인가 아닌가를 결정한다.

```
procedure Find_USABLE( )
 begin
    for i := 0 to FlowG_node_MAX do
          if (FlowG_node(i) == E_node) then X_USABLE := FALSE;
    for i := FlowG_node_MAX to 0 do
     begin
          for m := USE_SUCC_START(i) to
                   USE_SUCC_END(i) do
                   Use_Succ_Sum := Use_Succ_Sum ||
                   N_USABLE(m);
          N_USABLE(i) := FlowG_node(i).USED ||
                         !FlowG_node(i).IS_INST&&
                         X_USABLE(i);
          X_USABLE(i) := Use_Succ_Sum
     end
 end;
```

(알고리즘 5) 유용한 코드 모션 알고리즘

유용한 코드 모션 위치를 결정하기 위해서는 그 노드에 대한 상속자의 입력 부분이 유용한 코드 모션 위치인가 아닌가의 정보를 이용하게 된다. 따라서 USABLE은 마지막 노드에서 시작 노드로 제어흐름 반대 방향으로 분석해 나간다.

4.3.3 블록된 배정문 지연 알고리즘

(알고리즘 6)에서 N-LATEST와 X-LATEST는 배정문이 지연될 수 있는 위치들 중에서 배정문이 블록되거나 사용된 위치를 결정한다.

```
procedure Find_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 6) 블록된 배정문 지연 알고리즘

LATEST는 DELAYABLE의 값이 참일 경우에만 참일 수 있으므로 DELAYABLE의 값이 참인 노드에 대해서만 LATEST를 계산한다. 배정문 h4 := y + z에 대한 LATEST는 (알고리즘 6)에 의해 다음과 같이 계산된다.

$$N\text{-}LATEST_6 = T \wedge (T \vee F) = T$$
$$X\text{-}LATEST_5 = T \wedge \neg T = F$$

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.

4.3.4 변환 전 배정문 삽입 알고리즘

(알고리즘 7)에서 N-INIT와 X-INIT는 LATEST의 위치들 중에서 유용하지 않은 코드 모션 위치를 결정해서 v := t 형태의 변환 전 배정문을 삽입한다.

INIT는 LATEST의 값이 참일 경우에만 참일 수 있으므로 LATEST의 값이 참인 노드에 대해서만 INIT를 계산한다. 배정문 h4 := y + z에 대한 INIT는 (알고리즘 8)에 의해 다음과 같이 계산된다.

$$N\text{-}INIT_6 = T \wedge \neg F = T$$

α의 실제 값을 삽입할 위치 INIT의 결정은 LATEST가 참인 경우에 한해서 계산한다. INIT는 $h_\varepsilon := \varepsilon$ 형태의 배정문을 임시 기억장소 도입 이전의 v := t의 형태로 복원하는 것이다.

```
procedure Find_INIT( )
 begin
    for i := 0 to FlowG_node_MAX do
      begin
          N_INIT(i) := FALSE;
          X_INIT(i) := FALSE
      end;
    for i := 0 to FlowG_node_MAX do
      begin
          if (N_LATEST(i)) then
             N_INIT(i) := N_LATEST(i) && !X_USABLE(i);
          if (X_LATEST(i)) then
             X_INIT(i) := X_LATEST(i)
      end
 end;
```

(알고리즘 7) 변환 전 배정문 삽입 알고리즘

4.3.5 재구성 알고리즘

(알고리즘 9)에서 RECONSTRUCT는 v := h_ε 형태의 배정문에서 h_ε의 값을 원래의 식으로 재구성할 위치를 결정한다.

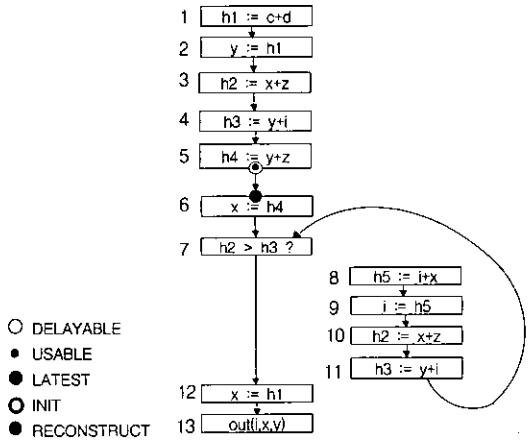
```
procedure Find_RECONSTRUCT( )
 begin
    for i := 0 to FlowG_node_MAX do
          RECONSTRUCT(i) := FALSE;
    for i := 0 to FlowG_node_MAX do
      begin
          if (N_INIT(i)) then
            RECONSTRUCT(i) := FlowG_node(i).USED &&
                              N_INIT(i)&& !X_USABLE(i);
          if (INITELIM(i)) then
            RECONSTRUCT(i) := INITELIM(i)
      end
 end;
```

(알고리즘 8) 재구성 알고리즘

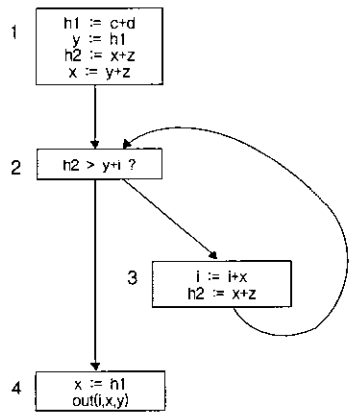
불필요한 코드 모션을 재구성하는 위치를 결정하는 RECONSTRUCT는 INIT의 값이 참일 경우에만 참일 수 있으므로 INIT의 값이 참인 노드에 대해서만 계산한다. 배정문 h4 := y + z에 대한 RECONSTRUCT는 (알고리즘 9)에 의해 다음과 같이 계산된다.

$$RECONSTRUCT_6 = T \wedge T \wedge \neg F = T$$

(그림 5)에서 배정문 h4 := y + z에 대한 DELAYABLE과 USABLE, LATEST, INIT, RECONSTRUCT를 계산한 결과는 (그림 6)과 같고 (그림 5)에 불필요한 코드 모션 재구성 단계를 적용한 결과는 (그림 7)과 같다.



(그림 6) DELAYABLE, USABLE, LATEST,INIT, RECONSTRUCT의 계산 결과



(그림 7) 불필요한 코드 모션 재구성 단계 수행 결과

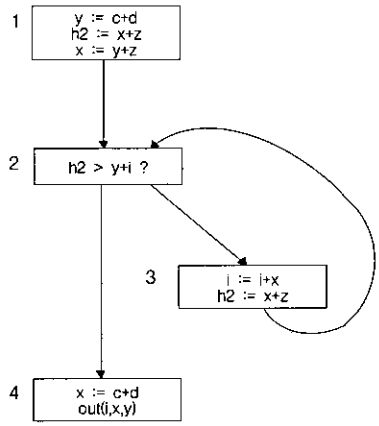
4.4 최종 최적화 단계

최종 최적화 단계에서는 초기 설정 단계에서 생성한 v := h_ε형태에 의한 h_ε의 사용을 사용횟수에서 제외시킨다. v := h_ε 형태의 사용을 사용횟수에 포함시키는 것은 불필요한 코드 모션이며 레지스터 압박을 초래할 수 있다. 따라서 v := h_ε의 형태로 사용된 h_ε은 사용횟수에서 제외하는 알고리즘을 제안한다.

```
procedure Find_INITELIM( )
 begin
      for i := 0 to FlowG_node_MAX do
            if (FlowG_node(i).ASSIGNMENT == INI_ASS) then
              INTELIM(i) := TRUE
 end;
```

(알고리즘 9) 최종 최적화 단계 알고리즘

(알고리즘 9)에서 INITELIM은 초기 설정 단계에서 생성된 v := h_ε의 형태는 h_ε의 사용횟수에서 제외하기 위해 계산된다. 따라서 INITELIM은 v := h_ε형태의 배정문에만 알고리즘을 적용한다. (그림 5)에 최종 최적화 단계를 적용하면 (그림 8)과 같다.



(그림 8) 최종 최적화 단계 수행 결과

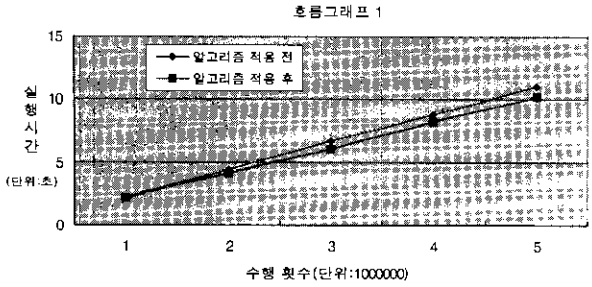
4.5 성능 분석

본 논문에서 제안한 배정문 모션 알고리즘의 성능 분석 결과는 다음과 같다.

<표 1>과 (그림 9)는 (그림 1)의 예제흐름그래프에 알고리즘을 적용시킨 결과를 나타낸다.

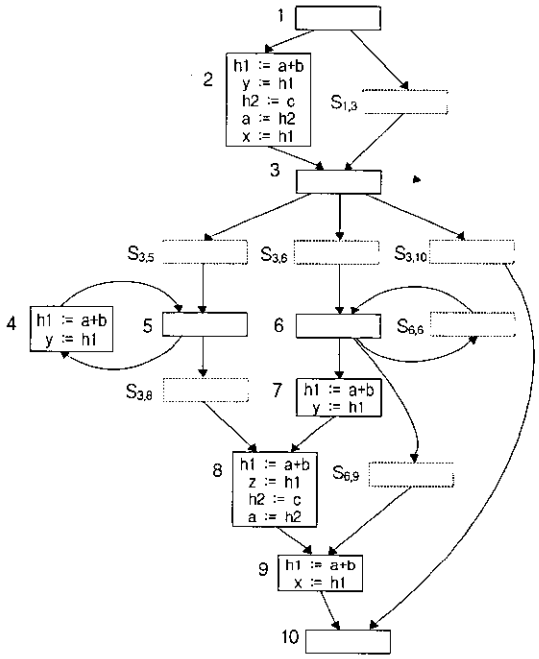
〈표 1〉 흐름그래프 1의 알고리즘 수행 결과

구 분 / 수행횟수	흐름그래프 1	
	알고리즘 적용 전	알고리즘 적용 후
1000000	2.197802	2.197802
2000000	4.450549	4.140659
3000000	6.648352	6.083516
4000000	8.791209	8.226373
5000000	11.043956	10.199231

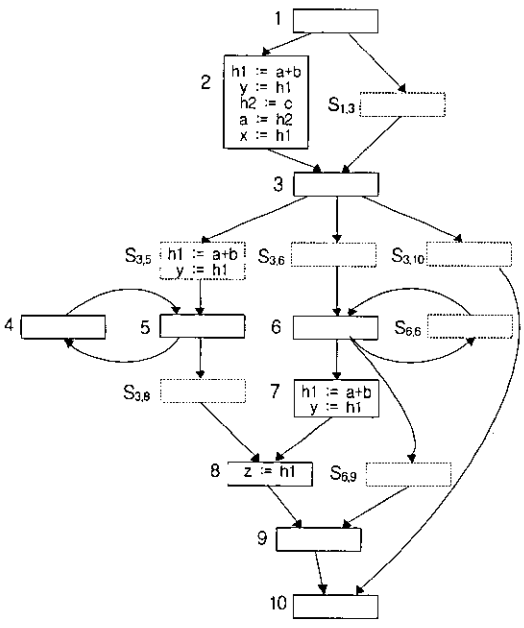


(그림 9) 흐름그래프1의 알고리즘 수행결과 분석

(그림 10)은 예제 흐름그래프 2의 초기설정단계 수행결과를 나타내며 (그림 11)은 알고리즘을 적용한 결과를 나타낸다.



(그림 10) 흐름그래프 2의 초기설정단계

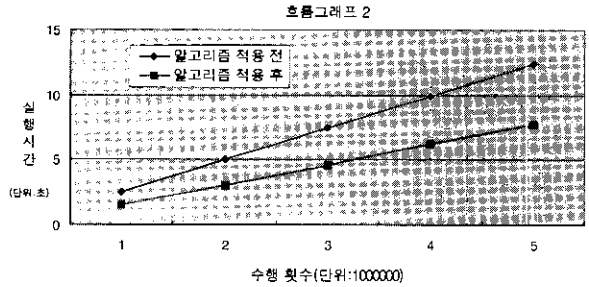


(그림 11) 흐름그래프 2의 알고리즘 적용 결과

<표 27>와 (그림 12)는 예제 흐름그래프 2에 알고리즘을 적용시킨 결과를 나타낸다.

〈표 2〉 흐름그래프 2의 알고리즘 수행 결과

구 분 / 수행 횟수	예제 흐름그래프 2	
	알고리즘 적용 전	알고리즘 적용 후
1000000	2.527473	1.593407
2000000	5.000000	3.076923
3000000	7.417582	4.615385
4000000	9.890110	6.208791
5000000	12.362637	7.692308

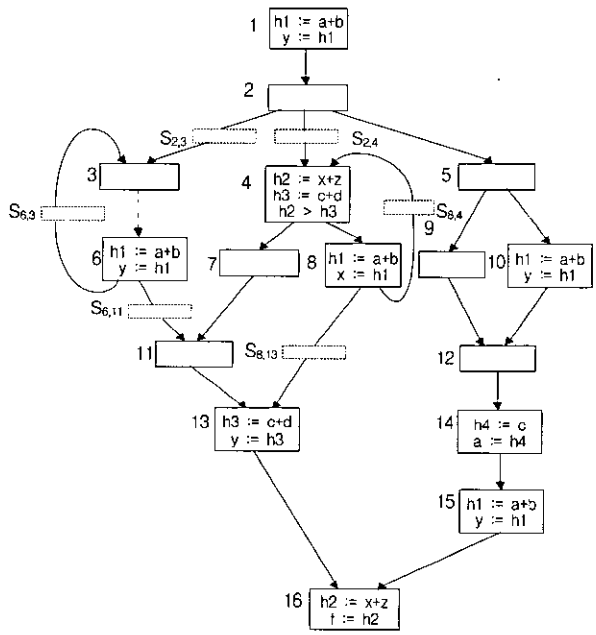


(그림 12) 흐름그래프 2의 알고리즘 수행 결과 분석

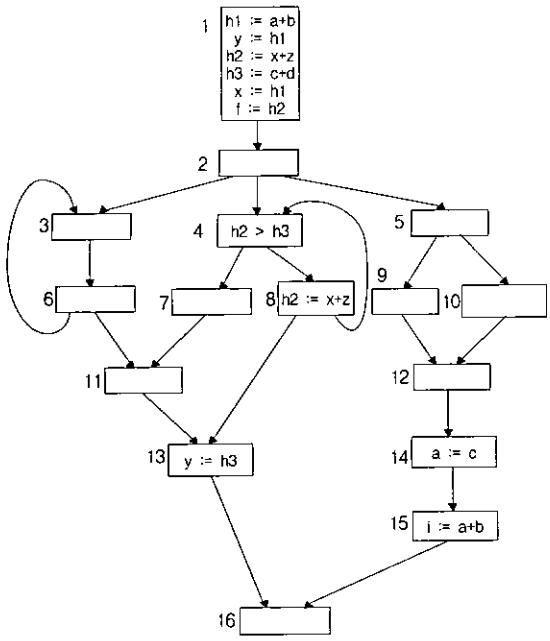
3가지 유형의 흐름그래프에 대한 성능 평가를 한 결과, 프로그램 내에 복잡한 반복문이 많고 반복문 내의 중복 코드가 많을수록 제안한 알고리즘을 적용한 경우의 효과가 크다는 것을 알 수 있다.

예를 들어, 수행 횟수가 5인 경우에 흐름그래프 1은 알고리즘 적용 후 실행시간이 약 7.6% 감소했지만 흐름그래프 3은 약 41.6% 감소했다. 따라서, 제안한 알고리즘은 프로그램의구조가 복잡하고 루프의 횟수가 많을 수록 적용 효과가 높다는 것을 알 수 있다.

(그림 13)은 예제 흐름그래프 3의 초기설정단계 수행결과를 나타내며 (그림 14)는 알고리즘을 적용한 결과를 나타낸다.



(그림 13) 흐름그래프 3의 초기설정단계

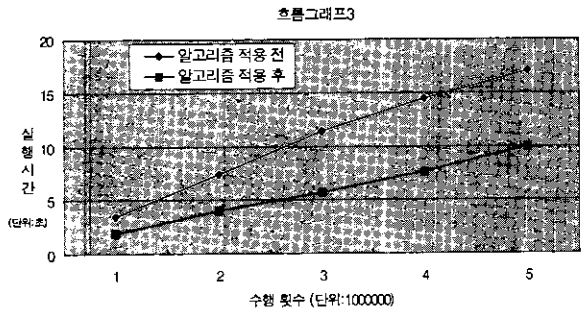


(그림 14) 흐름그래프 3의 알고리즘 적용 결과

<표 3>과 (그림15)는 예제 흐름그래프 3에 알고리즘을 적용시킨 결과를 나타낸다.

〈표 3〉 흐름그래프 3의 알고리즘 수행 결과

구 분 / 수행 횟수	흐름그래프3	
	알고리즘 적용 전	알고리즘 적용 후
1000000	3.457885	1.902143
2000000	7.424657	4.004571
3000000	11.362209	5.671439
4000000	14.417263	7.549824
5000000	17.011197	9.921502



(그림 15) 흐름그래프 3의 알고리즘 수행결과 분석

5. 결 론

프로그램 실행 시간의 효율성을 높이기 위해서는 수식과 배정문을 불필요하게 재 계산하고 재실행하는 것을 피해야 한다. 즉, 프로그램의 실행 전에 불필요한 코드와 중복된 코드들을 제거하고 코드들을 재결합하는 변환이 이루어져야 한다.

기존의 Knoop이 제시한 알고리즘은 초기 설정 단계에서 생성된 $v := h_\varepsilon$형태에 의한 h_ε의 사용을 사용횟수에 포함시켰으며 이 불필요한 코드 모션은 레지스터 압박을 초래할 수 있다. 따라서 배정문 모션 알고리즘의 최종 최적화 단계를 추가하여 $v := h_\varepsilon$의 형태로 사용된 h_ε은 사용횟수에서 제외하는 알고리즘을 제안했다.

본 논문에서 제안한 알고리즘은 모든 불필요한 코드 모션을 억제시키기 때문에 계산적으로나 수명적으로 최적인 알고리즘이다. 또한, 제안한 알고리즘의 동작 과정을 구체적으로 제시함으로써 Knoop의 방정식 알고리즘의 명확하지 않은 술어의 의미와, 노드 단위 분석과 명령어 단위 분석의 혼용 때문에 발생하는 모호함도 제거했다.

따라서, 알고리즘의 성능평가를 해 본 결과 불필요하게 중복된 수식이나 배정문의 수행을 피하게 함으로써 프로그램의 불필요한 재 계산이나 재실행을 하지 않도록 하여 기존의 방법보다 프로그램의 능률 및 실행시간을 향상시켰다.

본 논문에서 제안한 알고리즘을 병렬 프로그램의 코드 모션에 적용하여 재구성하는 것이 향후 연구 과제이다.

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

[3] Dhamdhere, D. M., Rosen, B. K. and Zadeck, F. K., "How to analyze large programs efficiently and informatively," In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92, of ACM SIGPLAN Notices, Vol.27, No.7, pp.212-223, San Francisco, CA, June 1992.

[4] 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.

[5] 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.

[6] Morel, E. and Renvoise, C., "Global optimization by suppression of partial redundancies," Communications of the ACM, Vol.22, No.2, pp.96-103, 1979.

[7] Dhamdhere, D. M., "Register assignment using code placement techniques," Journal of Computer Languages, Vol.13, No.2, pp.75-180, 1988.

[8] Dhamdhere, D. M., "A usually linear algorithm for register assignment using edge placement of load and store instructions," Journal of Computer Languages, Vol.15, No.2, pp.83-94, 1990.

[9] Dhamdhere, D. M., "Practical adaptation of the global optimization algorithm of Morel and Renvoise," ACM Transactions on Programming Languages and Systems, Vol.13, No.2, pp.291-294, 1991.

[10] 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.

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

[12] Knoop, J., Rüthing, O. and Steffen, B., "Partial dead code elimination," In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'94, of ACM SIGPLAN Notices, Vol.29, No.6, pp.147-158, Orlando, FL, June 1994.

신 현 덕

e-mail : ubhd@mail.kwandong.ac.kr

1998년 관동대학교 전자계산공학과 졸업 (공학사)

2000년 관동대학교 대학원 전자계산공학과 졸업(공학석사)

2000년~현재 관동대학교 대학원 전자계산공학과 박사학위 과정 중

2000년~현재 동우대학 컴퓨터정보과 겸임교수

2000년~현재 관동대학교 컴퓨터공학과 강사

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

안 희 학

e-mail : hhahn@mail.kwandong.ac.kr

1981년 숭실대학교 전자계산학과 졸업 (공학사)

1983년 숭실대학교 대학원 전자계산학과 졸업(공학석사)

1994년 숭실대학교 대학원 전자계산학과 졸업(공학박사)

1994년~1996년 관동대학교 전자계산소 소장

1984년~현재 관동대학교 컴퓨터공학과 교수

2001년~현재 관동대학교 전자계산소 소장

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