불필요한 코드 모션 억제를 위한 배정문 모션

An Assignment Motion to Suppress the Unnecessary Code Motion

신 현 덕* 이 대 식** 안 희 학***
Hyun-Deok Shin Dae-Sik Lee Heui-Hak Ahn

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.

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.

☞ keyword : code optimization, assignment motion, code motion, 코드 최적화, 배정문 모션, 코드 모션

1. 서 론

코드 최적화는 실행시간에 불필요한 값의 재계산을 피하기 위해 프로그램을 계산적으로나 수명적으로 최적인 상태로 변환하여 프로그램의 능률을 개선하는 기법이다.

프로그램을 계산적으로나 수명적으로 최적화하는 기법에는 수식 모션(EM : Expression Motion)과 수식 모션을 포함하는 배정문 모션(AM : Assignment Motion)이 있다[1,2].

수식모션에 의한 프로그램 최적화는 지금까지 많이 연구되었다. Dhamdhere에 의해 코드 모션 최적화를 위한 알고리즘이 제안되었으며[3] Knoop, Rüthing, Steffen에 의해 죽은 코드 제거 및 수식모션에 의한 최적화 코드 모션 알고리즘이 제안되었다[2,4-6].

부분 중복 제거의 기존 접근법들[7-11]은 프로그램의 수행 속도를 최적화하는데 중점을 두었으며 따라서 프로그램의 크기를 최적화하는 기법에 대한 연구가 필요하다.

프로그램 최적화에 대해 활발하게 연구가 수행된 수식모션과 대조적으로 배정문 모션은 지금까지 별로 연구되지 않았다.

제안된 알고리즘은 수식모션을 포함하는 배정

* 정 회 원 : 유한대학 컴퓨터정보과 강의전담교수
ubhd@yuhan.ac.kr
** 정 회 원 : 안동과학대학 사이버테러대응과 교수
dslee@andong-c.ac.kr
*** 정 회 원 : 관동대학교 공과대학 컴퓨터학과 교수
hhahn@kd.ac.kr
[2007/05/30 투고 - 2007/06/25 심사 - 2007/08/13 심사완료]

문 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 계산 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.

본 논문은 최종 최적화단계가 추가된 배정문 모션 알고리즘을 제안한다. 또한 기존 알고리즘 [2,12]의 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.

2. 배정문 모션을 위한 이론적 배경

수식 모션은 코드 모션의 후보가 되는 식을 프로그램내의 안전한 위치에 임시변수를 이용하여 초기화하고 그 후보가 되는 식이 사용되는 위치를 임시변수로 재 배치한다.

배정문 모션은 프로그램내의 모든 배정문에 임시변수를 도입하고(이 단계에서 배정문 모션은 수식 모션을 포함하게 된다.) 배정문 끌어올리기와 중복 배정문 제거 단계를 수행한다.

수식 모션의 동작과정은 먼저 프로그램내의 가장 이른 삽입 위치(earliest)를 결정한 후 이 삽입 위치들 중 프로그램의 경로 상에서 지연될 수 있는 가장 늦은 삽입 위치(delayability와 latest)를 계산하고 불필요한 임시변수 도입을 피하기 위한 독립 수식 위치(isolated)를 계산하여 수행된다.

수식 모션은 안전한 삽입위치 중에서 가장 이른 삽입위치에 t를 삽입한다. 따라서 ∀n∈N에 대한 Earliest(n)은 (정의 1)과 같다[4-6].

(정의 1) Earliest

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

수식 모션에서의 초기화는 계산적 최적성이 유지되는 한 시작 노드에서 마지막 노드까지의 경로상에서 지연될 수 있다. 이러한 특징으로 Delayability의 개념을 유도하면 정의 2와 같다 [4-6].

(정의 2) Delayability

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

불필요한 코드 모션을 억제하기 위해서 실행시간을 향상시키지 못하는 코드 모션은 억제되어야 한다. 프로그램에서 최대로 지연될 수 있는 포인트를 나타내는 술어 Latest는 정의 3과 같다 [4-6].

(정의 3) Latest

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

프로그램의 실행 시간을 향상시키는 코드 모션일지라도 불필요한 임시변수 초기화를 실행할 수 있다. 이러한 결점을 피하기 위한 Isolated는 정의 4와 같다[4-6].

(정의 4) Isolation

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

3. 배정문 모션 알고리즘

제안하는 배정문 모션 알고리즘은 기존의 Knoop 알고리즘을 개선한 알고리즘이다. Knoop이 제안한 배정문 모션은 노드 단위 분석을 이용하므로 알고리즘 적용이 모호하며 초기 설정 단계에서 도입된 h_t := t형태에 의한 t의 사용을 임시변수 사용횟수에 포함시키기 때문에 불필요한 코드 모션을 초래할 수 있다.

개선된 알고리즘은 배정문 모션을 명령어 단위로 적용하도록 재구성하고 최종 최적화 단계를 추가하여 초기 설정 단계에서 h_t := t의 형태로 도입된 t는 사용횟수에서 제외한다.

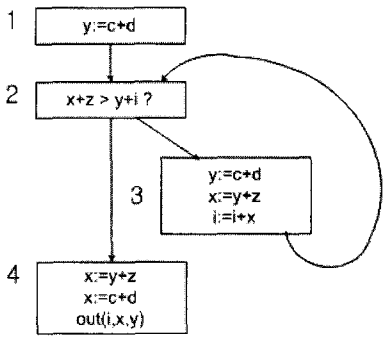
3.1 초기 설정 단계

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 1에 초기 설정 단계를 적용하면 그림 2와 같다.

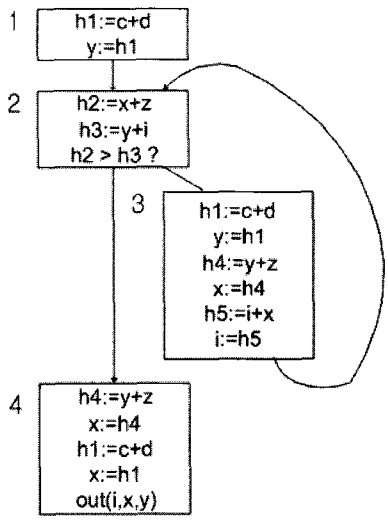
3.2 배정문 모션 단계

(1) 배정문 끌어올리기 알고리즘

배정문 끌어올리기 알고리즘은 프로그램의 의미를 유지하면서 배정문을 본래의 위치로부터 프로



(그림 1) 예제 흐름그래프



(그림 2) 초기 설정 단계 수행 결과

그램의 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.

그림 3에서 끌어올리기 후보를 나타내는 술어 N-HOISTABLE과 X-HOISTABLE은 배정문 패턴 α의 끌어올리기 후보들을 기본 블록 n의 입력이나 출력 부분까지 각각 이동시킬 수 있다는 것을 의미한다.

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.

```
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;
```

(그림 3) 배정문 끌어올리기 알고리즘

그림 2에서 배정문 h4 := y+z에 대한 HOISTABLE은 그림 3에 의해 다음과 같이 계산된다.

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

(2) 배정문 삽입 알고리즘

그림 4에서 술어 N-INSERT와 X-INSERT는 특별한 프로그램 지점에 삽입되는 배정문은 삽입에 의해 블록되지 않아야 한다는 것을 의미한다.

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

배정문 h4 := y+z에 대한 INSERT는 그림 4에 의해 다음과 같이 계산된다.

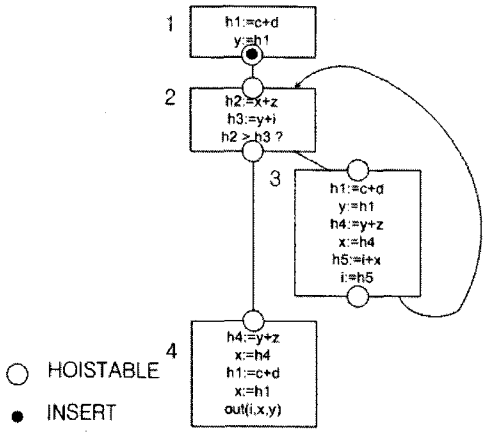
$$N\text{-}INSERT_1 = F \wedge \neg T = F$$
$$X\text{-}INSERT_1 = T \wedge T = T$$
$$N\text{-}INSERT_2 = T \wedge (\neg T \vee \neg T) = F$$
$$X\text{-}INSERT_2 = T \wedge F = F$$
$$N\text{-}INSERT_3 = T \wedge \neg T = F$$
$$X\text{-}INSERT_3 = T \wedge F = F$$
$$N\text{-}INSERT_4 = T \wedge \neg T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

```
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;
```

(그림 4) 배정문 삽입 알고리즘

배정문 패턴 α의 끌어올릴 위치를 결정하는 INSERT는 그 노드에 대한 선행자의 출력 부분이 끌어올릴 수 있는 위치인지 아닌지에 대한 정보를 이용하게 된다. 따라서 INSERT는 시작노드에서 마지막 노드로 제어흐름 방향으로 분석해 나간다.

배정문 h4 := y+z에 대한 HOISTABLE과 INSERT를 계산한 결과는 그림 5와 같다.



(그림 5) HOISTABLE과 INSERT의 계산 결과

그림 5에서 노드 1의 h4 := y+z에 대한 X-INSERT$_1$의 계산은 노드 1의 X-HOISTABLE$_1$이 참이고 h4 := y+z의 끌어올리기가 블록 되었기 때문에(LOC-BLOCKED$_1$이 참이기 때문에) X-INSERT$_1$은 참이다.

또한, 노드 2의 h4 := y+z에 대한 N-INSERT$_2$는 N-HOISTABLE$_2$가 참이고 노드 2의 선행자 1과 3의 X-HOISTABLE이 참이기 때문에 N-INSERT$_2$는 거짓이다. 즉, h4 := y+z를 노드 1의 출력 부분에만 삽입하게 된다.

(3) 중복 배정문 제거 알고리즘

그림 6에서 N-ELIMINATION과 X-ELIMINATION은 기본 블록 n의 입력이나 출력 부분에서 중복인 배정문 α를 제거한다.

배정문 h4 := y+z에 대한 ELIMINATION은 그림 6에 의해 다음과 같이 계산된다.

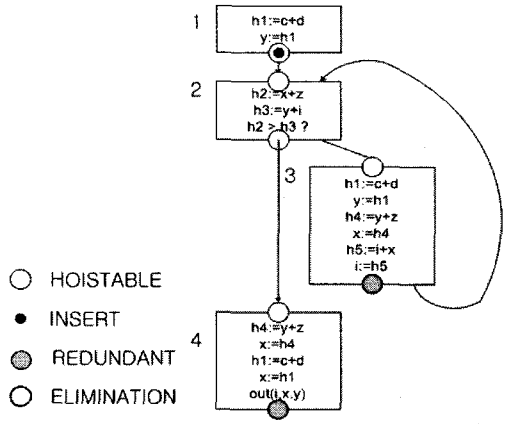
$$X-ELIMINATION_3 = T \wedge T = T$$
$$X-ELIMINATION_4 = T \wedge T = T$$

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

```
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;
```

(그림 6) 중복 배정문 제거 알고리즘

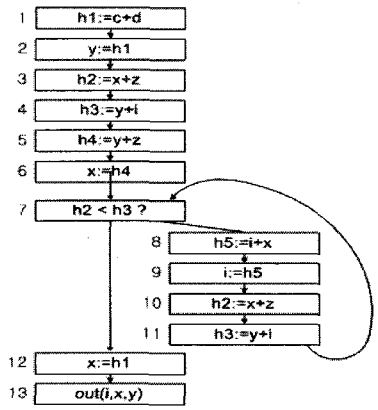
배정문 h4 := y+z에 대한 REDUNDANT와 ELIMINATION을 그림 6으로 계산한 결과는 그림 7과 같다.



(그림 7) REDUNDANT와 ELIMINATION의 계산 결과

그림 7에서 노드 3의 h4 := y+z에 대한 X-ELIMINATION$_3$의 계산은 노드 3의 X-REDUNDANT$_3$이 참이고 배정문 α의 패턴이기 때문에(EXECUTED$_3$이 참이기 때문에) X-ELIMINATION$_3$은 참이다.

그림 2에 배정문 모션 단계를 적용한 결과는 그림 8과 같다.



(그림 8) 배정문 모션 단계 수행 결과

3.3 불필요한 코드 모션 재구성 단계

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

(1) 배정문 지연 알고리즘

그림 9에서 N-DELAYABLE과 X-DELAYABLE은 배정문 모션에서 사용되는 초기화의 삽입이 계산적 최적화를 유지하면서 흐름그래프의 마지막 노드까지의 경로상에서 지연될 수 있음을 나타낸다. 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 정보를 이용하게 된다.

```
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;
```

(그림 9) 배정문 지연 알고리즘

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

(2) 유용한 코드 모션 알고리즘

그림 10에서 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;
```

(그림 10) 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, 제어흐름의 반대 방향으로 분석한다.

(3) 블록된 배정문 지연 알고리즘

그림 11에서 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.

```
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;
```

(그림 11) 블록된 배정문 지연 알고리즘

배정문 h4 := y+z에 대한 LATEST는 그림 11에 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) 변환 전 배정문 삽입 알고리즘

그림 12에서 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.

```
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;
```

(그림 12) 변환 전 배정문 삽입 알고리즘

배정문 h4 := y+z에 대한 INIT는 그림 12에 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_ε := ε 형태의 배정문을 임시 기억장소 도입 이전의 v := t의 형태로 복원하는 것이다.

(5) 재구성 알고리즘

그림 13에서 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;
```

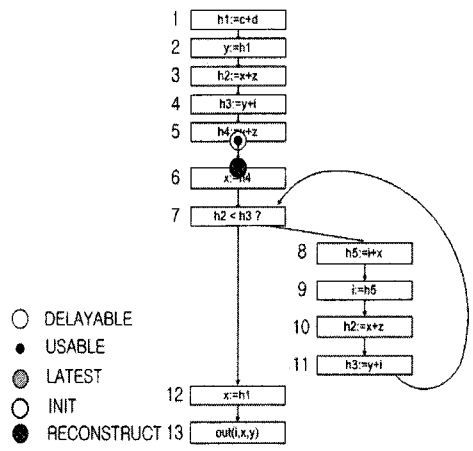
(그림 13) 재구성 알고리즘

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 그림 13에 의해 다음과 같이 계산된다.

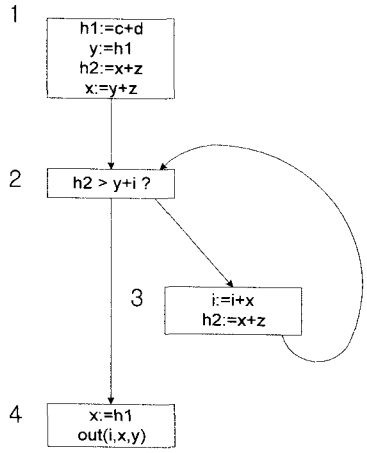
$$RECONSTRUCT_6 = T \wedge T \wedge \neg F = T$$

그림 8에서 배정문 h4 := y+z에 대한 DELAYABLE과 USABLE, LATEST, INIT, RECONSTRUCT를 계산한 결과는 그림 14와 같고 그림 8에 불필요한 코드 모션 재구성 단계를 적용한 결과는 그림 15와 같다.

그림 14에서 노드 1의 h4 := y+z에 대한 X-DELAYABLE$_1$의 계산은 노드 1이 h4 := y+z가 아니고(IS-INST$_1$이 거짓이고) N-DELAYABLE이 거짓이고 h4가 사용되지 않았고(USED$_1$이 거짓이고) h4 := y+z가 블록되지 않았기 때문에 X-DELAYABLE$_1$은 거짓이다.



(그림 14) DELAYABLE, USABLE, LATEST, INIT, RECONSTRUCT의 계산 결과



(그림 15) 불필요한 코드 모션 재구성 단계 수행 결과

또한 노드 1의 h4 :=y+z에 대한 N-USABLE$_1$의 계산은 h4 := y+z이 사용되지 않았고(USED$_1$이 거짓이고) 노드 1이 h4 := y+z가 아니고(IS-INST$_1$이 거짓이고) X-USABLE$_1$이 거짓이기 때문에 N-USABLE$_1$은 거짓이다.

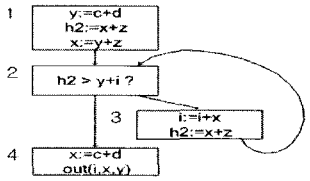
3.4 최종 최적화 단계

최종 최적화 단계에서는 초기 설정 단계에서 생성한 v:=h$_\varepsilon$형태에 의한 h$_\varepsilon$의 사용을 사용횟수에서 제외시킨다. v:=h$_\varepsilon$ 형태의 사용을 사용횟수에 포함시키는 것은 불필요한 코드 모션이며 레지스터 압박을 초래할 수 있다. 따라서 v:=h$_\varepsilon$의 형태로 사용된 h$_\varepsilon$은 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;
```

(그림 16) 최종 최적화 단계 알고리즘

그림 16에서 INITELIM은 초기 설정 단계에서 생성된 v:=h$_\varepsilon$의 형태는 h$_\varepsilon$의 사용횟수에서 제외하기 위해 계산된다. 따라서 INITELIM은 v:=h$_\varepsilon$ 형태의 배정문에만 알고리즘을 적용한다. 그림 8에 최종 최적화 단계를 적용하면 그림 17과 같다.



(그림 17) 최종 최적화 단계 수행 결과

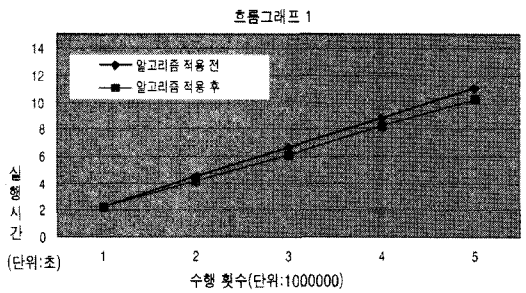
4. 성능 분석

본 논문에서 제안한 코드 모션 알고리즘의 성능 분석 결과는 다음과 같다. 표 1은 예제 흐름그래프에 알고리즘을 적용한 결과를 나타낸 것이다.

〈표 1〉 알고리즘 적용 결과 단위 : 초

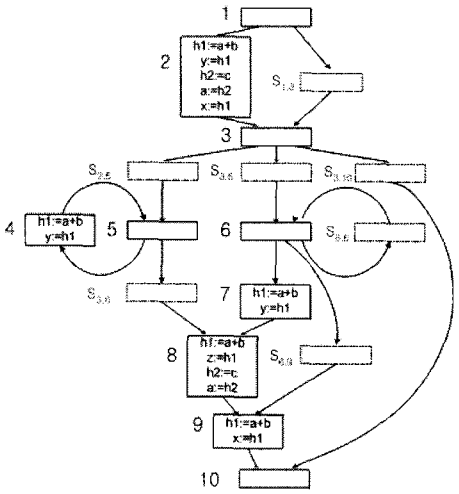
루프 반복	흐름그래프 1		흐름그래프 2		흐름그래프 3	
	적용전	적용후	적용전	적용후	적용전	적용후
1	2.20	2.20	2.53	1.60	3.46	1.90
2	4.45	4.14	5.00	3.08	7.42	4.00
3	6.65	6.08	7.42	4.62	11.36	5.67
4	8.80	8.23	9.89	6.21	14.42	7.55
5	11.04	10.20	12.36	7.70	17.01	9.92

흐름그래프 1은 그림 1의 예제 흐름그래프이며 그림 18은 알고리즘 수행 결과 분석을 나타낸 것이다.

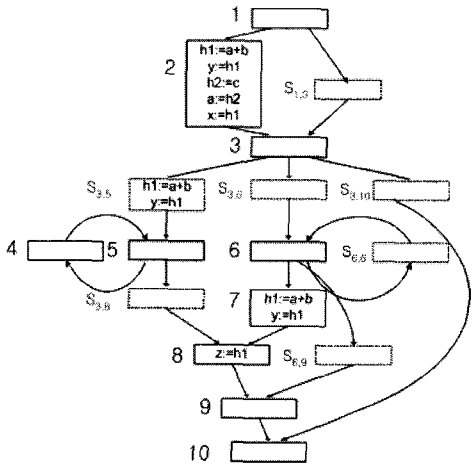


(그림 18) 흐름그래프 1의 알고리즘 수행 결과 분석

그림 19는 예제 흐름그래프 2의 초기설정단계 수행결과를 나타내며 그림 20은 알고리즘을 적용한 결과를 나타낸다.

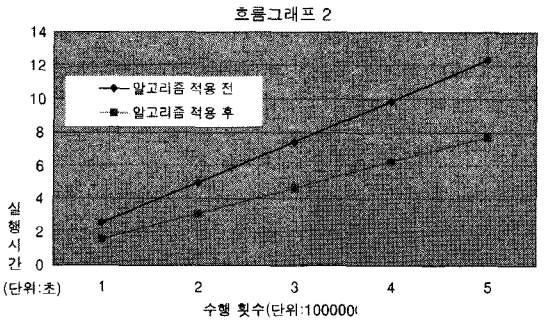


(그림 19) 흐름그래프 2의 초기설정단계



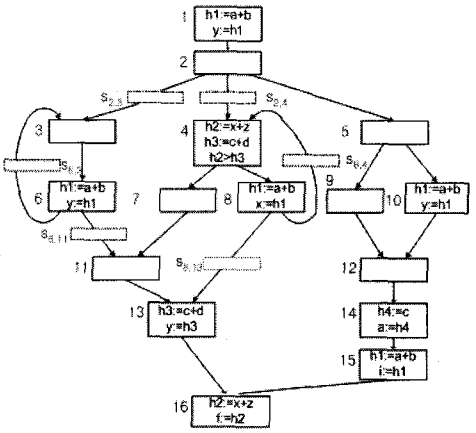
(그림 20) 흐름그래프 2의 적용 결과

그림 21은 흐름그래프 2에 알고리즘을 적용시킨 결과를 나타낸 것이다.



(그림 21) 흐름그래프 2의 알고리즘 수행 결과 분석

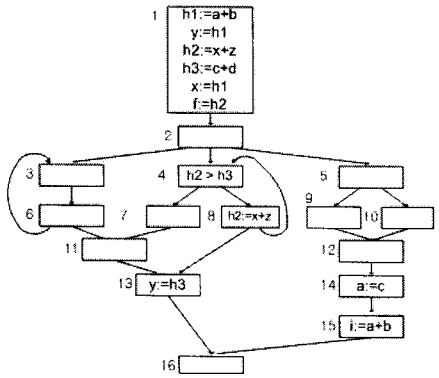
그림 22는 예제 흐름그래프 3의 초기설정단계 수행결과를 나타내며 그림 23은 알고리즘을 적용한 결과를 나타낸다.



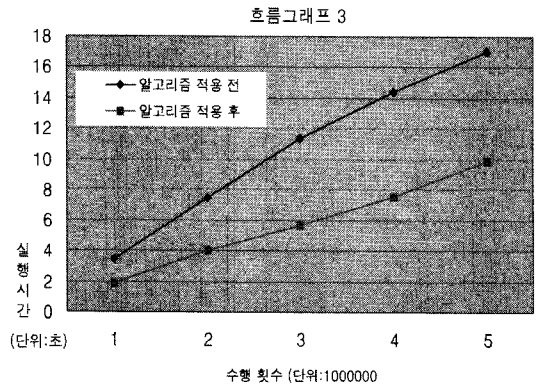
(그림 22) 흐름그래프 3의 초기설정단계

그림 24는 흐름그래프 3에 알고리즘을 적용시킨 결과를 나타낸 것이다.

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



(그림 23) 흐름그래프 3의 적용 결과



(그림 24) 흐름그래프 3의 알고리즘 수행 결과 분석

5. 결론

본 논문에서는 코드 최적화를 위하여 계산적, 수명적으로 제한이 없는 배정문 모션 알고리즘을 제안했다. 이 알고리즘은 배정문 모션 변환 알고리즘으로서 모든 배정문에 임시 변수를 도입하고 최대 상위 재 배치 전략으로 가능한 모든 배정문을 끌어올려서 중복된 배정문을 제거한다. 이 단계에서 불필요한 코드 모션이 일어날 수 있기 때문에 최대 하위 재 배치 전략을 이용하여 불필요한 코드 모션을 억제한다.

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

본 논문에서는 불필요한 코드 모션을 억제시키기는 알고리즘을 제안하고 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 재 실행을 하지 않도록 하여 기존의 방법보다 프로그램의 능률 및 실행시간을 향상시켰다.

참 고 문 헌

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] Bodik, R., Gupta, R. and Soffa, M. L., "Complete removal of redundant computations", Proceedings of ACM Conference on Programming Language Design and Implementation, Vol. 33, No. 5, pp. 1-14, New York, June 1998.

[8] Dai, X., Zhai, A., Hsu, W. C. and Yew, P. C. A "General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion", Proceedings of the international symposium on Code generation and optimization CGO '05, pp. 280-290, March 2005.

[9] Lin, J., Hsu, W. C., Yew, P. C., Ju, R. D. and Ngai, T. F. "Recovery code generation for general speculative optimizations", ACM Transactions on Architecture and Code Optimization(TACO), Vol 3, Issue 1, pp. 67-89, March 2006.

[10] Scholz, B., Horspool, N. and Knoop, J., "Optimizing for space and time usage with speculative partial redundancy elimination", ACM SIGPLAN Notices, Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, Vol. 39, No. 7, pp. 221-230, June 2004.

[11] Xue, J. and Cai, Q. "A lifetime optimal algorithm for speculative PRE", ACM Transactions on Architecture and Code Optimization(TACO), Vol. 3, Issue 2, pp. 115-155, June 2006.

[12] 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.

◐ 저 자 소 개 ◑

신 현 덕(Hyun-Deok Shin)
1998년 관동대학교 전자계산공학과 졸업(공학사)
2000년 관동대학교 대학원 전자계산공학과 졸업(공학석사)
2006년 관동대학교 대학원 전자계산공학과 졸업(공학박사)
2007년 ~현재 유한대학 컴퓨터정보과 강의전담교수
관심분야 : 컴파일러, 프로그래밍 언어, 최적화, 임베디드 시스템
E-mail : ubhd@yuhan.ac.kr

이 대 식(Dae-Sik Lee)
1995년 관동대학교 전자계산공학과 졸업(공학사)
1999년 관동대학교 대학원 전자계산공학과 졸업(공학석사)
2004년 관동대학교 대학원 전자계산공학과 졸업(공학박사)
관심분야 : 시스템 소프트웨어 및 컴퓨터보안
E-mail : dslee@andong-c.ac.kr

안 희 학(Heui-Hak Ahn)
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 컴퓨터학과 교수
관심분야 : 컴파일러, 병렬컴파일러, 프로그래밍 언어, 함수언어, 오토마타 등
E-mail : hhahn@kd.ac.kr