論文
5-7-2

불필요한 코드 모션 재구성 알고리즘
An Algorithms of reconstruct unnecessary Code Motion

심 손 권(Son-Kweon Sim)[1]

요 약

프로그램을 계산적으로나 수명적으로 최적화하는 기법에는 수식 모션 변환과 수식 모션을 포함하는 배정문 모션 변환이 있다.

본 논문에서는 노드 단위 분석과 명령어 단위 분석의 혼용 때문에 발생하는 모호함을 가지는 Knoop의 알고리즘을 개선하는 불필요한 코드 모션 재구성 알고리즘을 제안하였다. 제안한 알고리즘은 수식이나 배정문의 불필요한 재계산이나 재수행을 피하게 함으로써 프로그램의 능률과 수행시간을 개선하였다.

ABSTRACT

There are conversion of expression motion and assignment motion inclusive expression motion in techniques that optimized program computationally and livelily.

In this paper, I suggest that an algorithms of reconstruct unnecessary code motion which is improves Knoop's algorithms that have ambiguous. It is occurred by mixing the node level analysis and the instruction level analysis. This algorithm improves runtime and efficiency of a program by avoiding the unnecessary recalculations and reexecutions of expressions and assignment statements.

논문접수 : 2004. 7. 10.
심사완료 : 2004. 7. 23.

1) 정회원 : 강원도립대학 컴퓨터응용과 겸임강사

1. 서 론

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

계산적 최적화는 식을 프로그램내의 안전한 삽입 위치에 초기 설정된 임시변수로 삽입하고, 본래의 식은 임시변수로 재배치함으로써 이루어질 수 있다. 수명 최적화(lifetime optimal)는 프로그램의 계산적 최적성이 유지되는 범위 내에서 식을 가능한 한 늦게 두는 것으로서 이 전략에 의해 계산적 최적화 과정에서 도입된 임시변수의 수명을 최소로 줄일 수 있다[2].

프로그램을 계산적으로나 수명적으로 최적화하는 기법에는 수식 모션(EM : Expression Motion) 변환[2, 3, 4, 5]과 수식 모션을 포함하는 배정문 모션(AM : Assignment Motion) 변환[7, 8, 9, 10]이 있다.

본 논문은 Knoop이 제시한 방정식 알고리즘[10-12]을 개선시킨다. Knoop이 제시한 방정식 알고리즘은 초기 설정 단계에서 도입된 $v:=h_\varepsilon$ 형태에 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 $v:=h_\varepsilon$의 형태로 도입된 h_ε은 사용횟수에서 제외한다.

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.

2. 수식 모션

수식 모션에는 프로그램의 의미 변화 없이 가장 이른 삽입위치로 수식을 끌어올리는 방법과 수식을 가장 늦은 삽입위치로 끌어내리는 방법이 있다.

수식을 가장 이른 삽입위치로 끌어올리는 방법은 안전한 삽입위치 중에서 가장 이른 삽입위치에 t를 삽입한다. 따라서 $\forall n \in N$에 대한 Earliest(n)은 (정의 2-1)과 같다[2, 11, 12].

(정의 2-1) Earliest

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

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[2, 11, 12].

(정의 2-2) Delayability

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

(정의 2-3) 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.[2, 11, 12].

(정의 2-4) Isolation

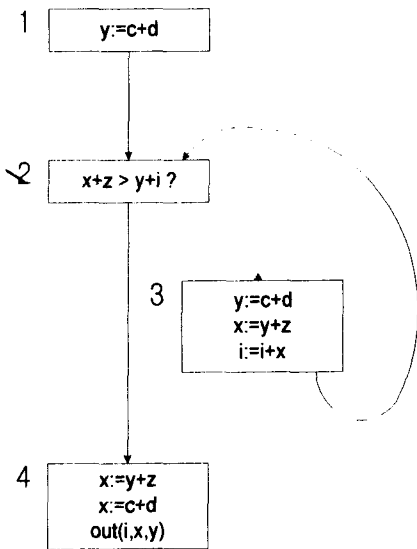
$$\forall CM \in \mathcal{CM}\ \forall n \in N.$$
$$Isolated_{CM}(n) \Leftrightarrow_{df}$$
$$\forall p \in P[n, e]\ \forall 1 < i \le \lambda_p.\ Replace_{CM}(p_i) \Rightarrow Insert^{\dashv}_{CM}(p[1, i[)$$

3. 코드 최적화 알고리즘

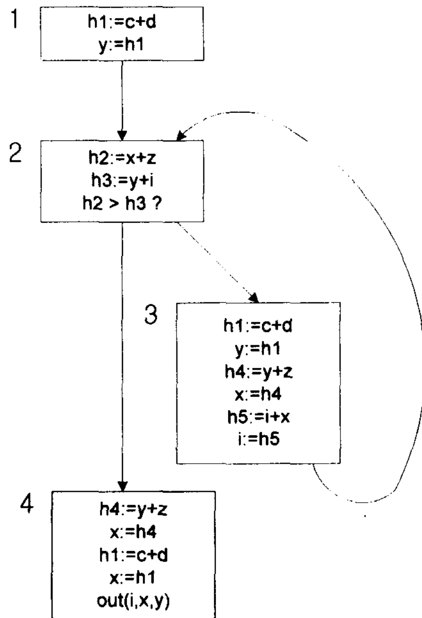
본 논문에서는 코드 최적화를 위하여 계산적으로나 수명적으로 제한이 없는 알고리즘을 제안한다. 이 알고리즘은 어떤 불필요한 코드 모션도 억제한다.

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 3-1]에 초기 설정 단계를 적용하면 [그림 3-2]와 같다.



[그림 3-1] 예제 흐름그래프
[Fig. 3-1] Example of Flow-Graph



[그림 3-2] 초기 설정 단계 수행 결과
[Fig. 3-2] Executive result of an Initial Setting Level

3.2 배정문 모션 단계

배정문 모션 단계에서는 LOC_HOISTABLE, LOC_BLOCKED, EXECUTED, ASS_TRANSP라는 지역 술어들을 이용하여 배정문 모션 후보를 프로그램의 안전한 위치로 끌어올린다.

노드 n에 대해서 배정문 패턴 α가 α≡ v:=t 라 할 때, LOC_HOISTABLE은 n의 끌어올리기 후보 α를 의미한다. 또한 LOC_BLOCKED는 α의 끌어올리기가 노드 n의 다른 문장에 의해 블록 되었다는 것을 의미하고 EXECUTED는 현재 문장이 α 패턴의 배정문이라는 것을 의미하며 ASS_TRANSP는 배정문 패턴 α의 v뿐만 아니라 t의 어떤 피연산자도 현재 문장에 의해 수정되지 않는다는 것을 의미한다. 이러한 지역 술어를 사용하여 배정문 끌어올리기와 중복 배정문 제거 알고리즘을 제안한다.

배정문 끌어올리기 알고리즘은 프로그램의 의미를 유지하면서 배정문을 본래의 위치로부터 프로그램의 가장 앞부분으로 끌어올리는 것이

다. [알고리즘 1]은 배정문을 끌어올릴 수 있는 노드의 위치를 결정한다.

```
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

[Algorithm 1] The Hoist Algorithm of assignment statement

[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.

끌어올리기 후부는 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 3-2]에서 배정문 h4:=y+z에 대한 HOISTABLE은 [알고리즘 1]에 의해 다음과 같이 계산된다.

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

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_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.LOC_BLOCKED
    end
 end;
```

[알고리즘 2] 배정문 삽입 알고리즘

[Algorithm 2] The Insert Algorithm of assignment statement

[알고리즘 2]는 배정문 패턴 α의 실제 값을 삽입할 위치를 계산한다. 술어 N-INSERT와

X- INSERT는 특별한 프로그램 지점에 삽입되어야 하는 모든 배정문은 블록되지 않아야 한다는 것을 의미한다.

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

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

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.

```
procedure Find_REDUNDANT( )
 begin
  for i := 0 to FlowG_node_MAX do
    if (FlowG_node[i] == S_node) then
        N_REDUNDANT := FALSE;
  for i := 0 to FlowG_node_MAX do
    begin
     for m:=REDUN_PRED_START(i) to
        REDUN_PRED_END(i) do
     Redun_Pred_Sum := Delay_Pred_Sum &&
     X_REDUNDANT[m];
     N_REDUNDANT[i] := Redun_Pred_Sum;
     X_REDUNDANT[i] :=
     FlowG_node[i].ASS_TRANSP &&
        (FlowG_node[i].EXECUTED ||
         N_REDUNDANT[i])
    end
 end;
```

[알고리즘3] 중복 배정문 위치 결정 알고리즘
[Algorithm 3] The Location Result Algorithm of the redundant assignment statement

[알고리즘 3]은 프로그램내의 중복된 배정문의 위치를 결정한다. [알고리즘 3]에서 N_REDUNDANT와 X_REDUNDANT는 배정문 α가 기본 블록 n의 입력 또는 출력부분에서 중복이라는 것을 의미한다. 기본 블록에서 중복된 모든 배정문을 제거한다.

배정문 h4:=y+z에 대한 REDUNDANT는 [알고리즘 3]에 의해 다음과 같이 계산된다.

$N\text{-}REDUNDANT_1 = F$
$X\text{-}REDUNDANT_1 = F \wedge (F \vee F) = F$
$N\text{-}REDUNDANT_2 = F \wedge F = F$
$X\text{-}REDUNDANT_2 = T \wedge (F \vee F) = F$
$N\text{-}REDUNDANT_3 = F$
$X\text{-}REDUNDANT_3 = T \wedge (T \vee F) = T$
$N\text{-}REDUNDANT_4 = F$
$X\text{-}REDUNDANT_4 = T \wedge (T \vee F) = T$

배정문의 중복위치를 결정하기 위해서는 그 노드에 대한 선행자의 출력 부분에서 배정문이 중복인지 아닌지에 대한 정보를 이용하게 된다. 따라서 REDUNDANT는 시작노드에서 마지막 노드로 제어흐름 방향으로 분석해 나간다.

```
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 4] 중복 배정문 제거 알고리즘
[Algorithm 4] A Deletion Algorithm of the redundant assignment statement

[알고리즘 4]에서 N_ELIMINATION과 X_ELIMINATION은 기본 블록 n의 입력이나 출력 부분에서 중복인 배정문 α를 제거한다.

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

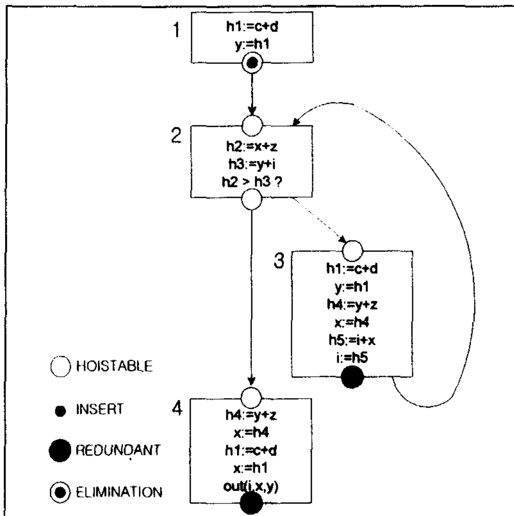
배정문 h4 := y+z에 대한 ELIMINATION은 [알고리즘 2]에 의해 다음과 같이 계산된다.

$$X_ELIMINATION_3 = T \wedge T = T$$

$$X_ELIMINATION_4 = T \wedge T = T$$

[그림 3-2]에서 노드 3의 h4 := y+z에 대한 X-ELIMINATION$_3$의 계산은 노드 3의 X-REDUN DANT$_3$이 참이고 배정문 α의 패턴이기 때문에 X-ELIMINATION$_3$은 참이다.

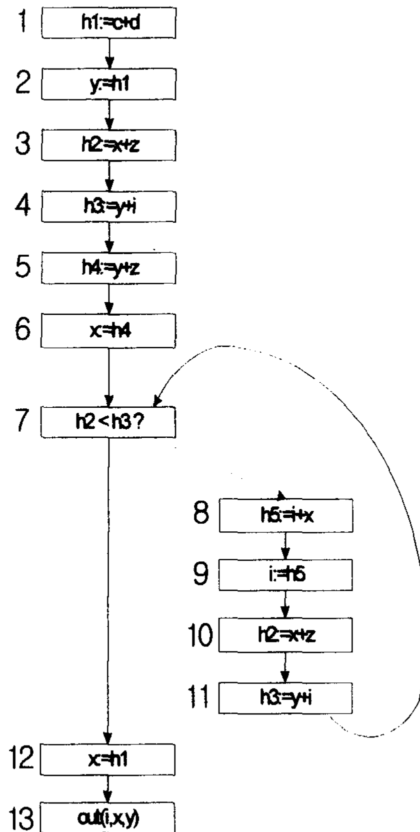
배정문 h4 := y+z에 대한 REDUNDANT와 ELIMINATION을 [알고리즘 3]과 [알고리즘 4]로 계산한 결과는 [그림 3-3]과 같다.



[그림 3-3] REDUNDANT와 ELIMINATION의 계산결과

[Fig 3-3] A calculative result of REDUNDANT and ELIMINATION

[그림 3-3]에 배정문 모션 단계를 적용한 결과는 [그림 3-4]와 같다.



[그림 3-4] 배정문 모션 수행 결과

[Fig. 3-4] Executive result of assignment Motion

4. 불필요한 코드 모션 재구성 알고리즘

4.1 재구성 알고리즘

불필요한 코드 모션 재구성 단계에서는 프로그램의 의미를 유지하면서 $h_\varepsilon:=\varepsilon$(ε는 수식 패턴) 형태의 모든 배정문을 프로그램의 가장 뒷부분으로 옮기고, h_ε가 배정문의 실제 값 이후에 많아야 한번 이용되는 모든 실제 값을 제거

한다.

[알고리즘 5]는 배정문을 원래의 식으로 재구성할 위치를 결정한다.

[알고리즘 5]에서 RECONSTRUCT는 v := h_ε 형태의 배정문에서 h_ε의 값을 원래의 식으로 재구성할 위치를 결정한다.

불필요한 코드 모션을 재구성하는 위치를 결정하는 RECONSTRUCT는 INIT의 값이 참일 경우에만 참일 수 있으므로 INIT의 값이 참인 노드에 대해서만 계산한다.

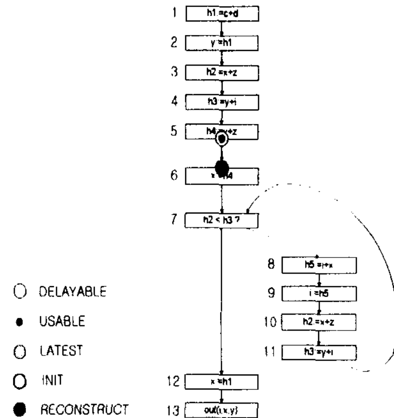
```
procedure Find_RECONSTRUCT( )
 begin
  for i := 0 to FlowG_node_MAX do
    RECONSTRUCT[i] := FALSE;
  for i :=  0 to FlowG_node_MAX do
    begin
    if (N_INIT[i]) then
       RECONSTRUCT[i] :=
       FlowG_node[i].USED&&N_INIT[i] &&
       !X_USABLE[i];
   if (INITELIM[i]) then RECONSTRUCT[i] :=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

[알고리즘 5] 재구성 알고리즘
[Algorithm 5] A Reconstruct Algorithm

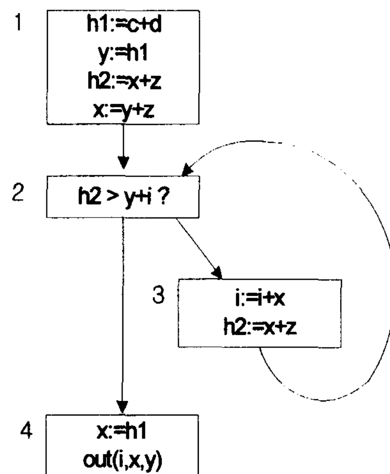
배정문 h4 := y+z에 대한 RECONSTRUCT는 [알고리즘 5]에 의해 다음과 같이 계산된다.

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

배정문 h4 := y+z에 대한 DELAYABLE과 USABLE, LATEST, INIT, RECONSTRUCT를 [알고리즘 5]로 계산한 결과는 [그림 4-1]과 같고 불필요한 코드 모션 재구성 단계를 적용한 결과는 [그림 4-2]와 같다.



[그림 4-1] DELAYABLE, USABLE, LATEST, INIT, RECONSTRUCT의 계산 결과
[Fig 4-1] A Calculative result of DELAYABLE, USABLE, LATEST, INIT, RECONSTRUCT



[그림 4-2] 재구성 알고리즘 수행 결과
[Fig. 4-2] Executive result of a Reconstruct Algorithm

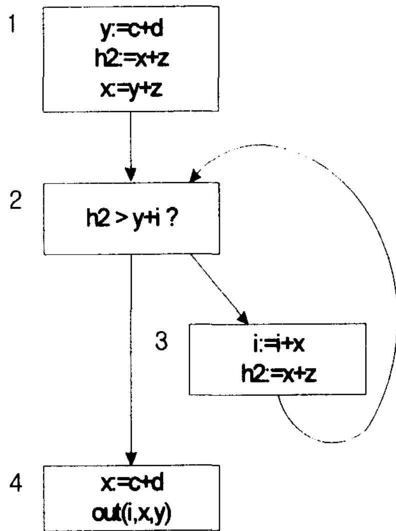
4.2 최종 최적화 단계

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 6] 최종 최적화 알고리즘
[Algorithm 6] A Final Optimization Algorithm

[그림 4-2]에 최종 최적화 단계를 적용하면 [그림 4-3]과 같다.

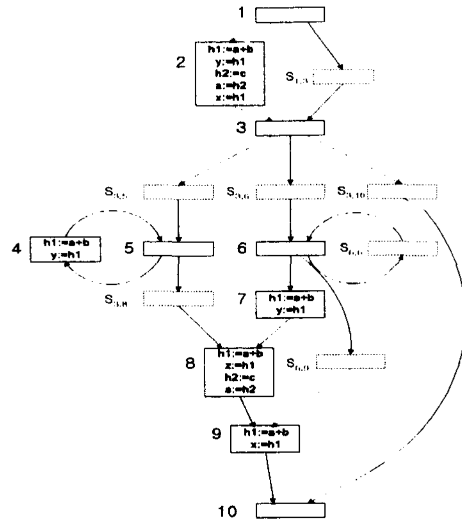


[그림 4-3] 최종 최적화 수행 결과
[Fig. 4-3] Executive result of final Optimization

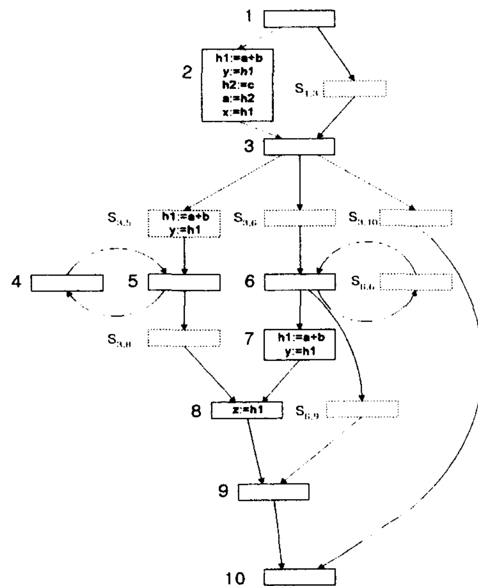
4.3 성능 분석

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

[그림 4-4]는 예제 흐름그래프의 초기설정단계 수행결과를 나타내며 [그림 4-5]는 알고리즘을 적용한 결과를 나타낸다.



[그림 4-4] 예제 흐름그래프의 초기설정단계
[Fig. 4-4] A Initial Setting Level of the Example

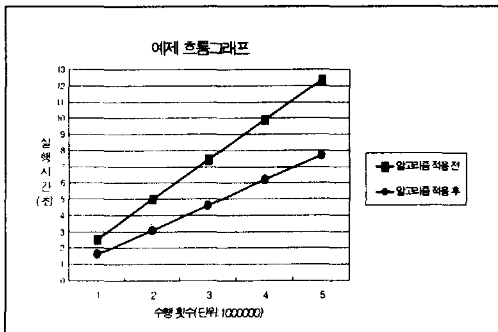


[그림 4-5] 예제흐름그래프 알고리즘 적용 결과
[Fig. 4-5] Result that apply a algorithm to an example

<표 4-1>과 [그림 4-6]은 예제 흐름그래프에 알고리즘을 적용시킨 결과를 나타낸다.

<표 4-1> 알고리즘의 수행결과
<table 4-1> Executive result of an Algorithm

구 분 / 수행 횟수	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 4-6] 알고리즘 수행결과 분석
[Fig. 4-6] An Analysis of result that execute an algorithm

성능 평가 결과, 프로그램 내에 복잡한 반복문이 많고 반복문 내의 중복 코드가 많을수록 제안한 알고리즘을 적용한 경우의 효과가 크다는 것을 알 수 있다. 따라서, 제안한 알고리즘은 프로그램의 구조가 복잡하고 루프의 횟수가 많을 수록 적용 효과가 높다는 것을 알 수 있다.

5. 결론

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

본 논문에서는 Knoop이 제시한 방정식 알고리즘을 개선시켰다. Knoop이 제시한 알고리즘은 초기 설정 단계에서 생성된 v := h_ε형태에 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_ε의 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 제시함으로서 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 함으로서 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., "*Co-mpilers Principles, Techniques, and Tools*", Addison-wesley publishing Co., 1986.
[2] Knoop, J., Rüthing, O. and Steffen, B., "Opti-mal code motion: theory and practice",

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4, pp. 1117-1155, 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., Rosen, B. K. and Zadeck, F. K., "How to analyze large programs effici-ently and informatively", In *Proc. ACM SIG-PLAN Conference on Programming Language Design and Implementation'92, of ACM SIG-PLAN Notices*, Vol. 27, No. 7, pp. 212-223, San Francisco, CA, June 1992.

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

[7] Dhamdhere, D. M., "Register assignment using code placement techniques", *Journal of Com-puter 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 Co-nference 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 SIG-PLAN Conference on Programming Language Design and Implementation'94*, of *ACM SIG-PLAN Notices*, Vol. 29, No. 6, pp. 147-158, Orlando, FL, June 1994.

심 손 권

e-mail : paransea@daum.net

1996년 관동대학교 전자계산공학과 졸업(공학사)

1998년 관동대학교 대학원 전자계산공학과 졸업(공학석사)

2003년 관동대학교 대학원 전자계산공학과 졸업(공학박사)

1998년 ~ 1999년 관동대학교 산업기술연구소 책임연구원

1998년 ~ 2003년 관동대학교 컴퓨터공학과 강사

2002년 ~ 현재 강원도립대학 컴퓨터응용과 겸임강사

관심분야 : 컴파일러, 병렬컴파일러, 프로그래밍 언어, 웹 프로그래밍 언어 등