論文
5-9-26

수명적, 계산적 최적화를 위한 희소코드모션 알고리즘
A Sparse Code Motion Algorithm forlifetime and computational optimization

심 손 권 (Son-Kweon Sim)[1]

요 약

일반적으로 코드 모션 알고리즘은 계산적 최적화와 레지스터 과부하와 연관되는 실행시간 최적화를 수행한다.

본 논문은 계산적 최적화와 수명적 최적화에 더하여 코드의 크기를 고려하는 희소 코드 모션 알고리즘을 제안한다. 희소 코드 모션 알고리즘에서 BCM 알고리즘은 계산적으로 최적 코드 모션을 수행하고, LCM 알고리즘은 레지스터 과부하를 감소시킨다. 희소 코드 모션 알고리즘은 불필요한 코드 모션을 억제시키기 때문에 계산적으로나 수명적으로 최적인 알고리즘이다.

희소 코드 모션 알고리즘은 성능평가를 통하여 기존의 연구보다 프로그램의 능률 및 실행시간을 향상시켰다.

ABSTRACT

Generally, the code motion algorithm accomplishes the run-time optimal connected with the computational optimization and the register overhead

This paper proposes a sparse code motion, which considers the code size, in addition to computational optimization and lifetime optimization. The BCM algorithm carries out the optimal code motion computationally and the LCM algorithm reduces the register overhead in a sparse code motion algorithm. A sparse code motion algorithm is optimum algorithm computationally and lifetime because of suppression unnecessary code motion

This algorithm improves runtime and efficiency of the program than the previous work through the performance test.

논문접수 : 2004. 11. 25.
심사완료 : 2004. 12. 15.

1) 정회원 : 강원도립대학 컴퓨터응용과 겸임강사

1. 서 론

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

일반적으로 부분 중복제거(PRE : Partial Redundancy Elim- ination) 알고리즘[2]은 실행시간에 수행되는 계산수의 감소와 불필요한 레지스터 과부하를 피하기 위한 수명의 최소화라는 두 가지 목적을 갖는다[3,4].

프로그램을 계산적으로나 수명적으로 최적화하는 기법에는 수식 모션(EM : Expression Motion) 변환[3,5-8]과 배정문 모션(AM : Assignment Motion) 변환[4,9-11], 희소 코드 모션(SpCM : Sparse Code Motion)[12] 등이 있다.

본 논문에서는 BCM(Busy Code Motion) 알고리즘과 LCM (Lazy Code Motion) 알고리즘[9,11]을 확장한 희소 코드 모션 알고리즘을 제안하고, 알고리즘의 동작 과정을 구체적으로 제시하여 Knoop의 희소 코드 모션 알고리즘의 이론적 제시에 대한 술어들의 모호한 의미의 방정식 형태인 BCM 알고리즘과 LCM 알고리즘을 명확하게 재구성하여 성능 평가를 통해 실제적인 코드 최적화가 되었는지 살펴보고자 한다.

2. 코드 모션

코드 모션 변환은 삽입될 수 있는 노드를 결정하는 $Insert_{CM}$과 재배치될 수 있는 노드를 결정하는 $Replace_{CM}$의 두 가지 술어로 표현된다. 흐름 그래프에서의 각 노드는 코드 모션 후보를 포함하고 있는 노드와 코드 모션 후보에 대한 수정을 포함하지 않은 노드로 정의 할 수 있다.

프로그램의 각 변수의 의미적 보증을 위해 삽입은 안전해야 하며 코드 모션 후보도 올바르게 재배치되어야 하고, 삽입위치는 안전한 삽입위치 중에서 가장 이른 삽입위치에 임시변수 t를 삽입한다. 코드 모션에서의 초기화는 계산적 최적성이 유지되는 한 시작 노드에서 마지막 노드까지의 경로 상에서 지연될 수 있고, 불필요한 코드 모션을 억제하기 위해서 실행 시간을 향상시키지 못하는 코드 모션은 억제되어야 한다.

일반적인 코드 모션 변환 형태는 먼저 모든 코드 모션 후보 t에 대해 임시 변수 h_{CM}을 도입한다.

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.

재 배치는 입력 부분 재 배치 조건을 만족하는 모든 $n \in N$에서 t에 대한 유일한 입력 계산을 h_{CM}으로 재 배치하고, 출력 부분 재 배치 조건을 만족하는 모든 $n \in N$에서 t에 대한 유일한 출력 계산을 h_{CM}으로 재 배치한다.

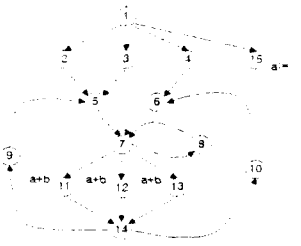
코드 모션의 첫 번째 목표는 모든 프로그램 경로상의 계산 수를 최소화하는 것이다. 계산적 최적화 코드 모션은 최대 상위 재배치 전략을 이용한다. 이 기법은 흐름그래프의 Safe 부분에서 Earliest 프로그램 위치를 결정짓는다.

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 임시 변수의 수명을 최소화하기 위해 수명적 최적화 코드 모션 변환을 필요로 한다.[10-12]

3. 희소 코드 모션



[그림 1] 희소 코드 모션을 위한 프로그램 예
[Fig. 1] Example of program for a Sparse Code Motion

희소 코드 모션 알고리즘은 BCM 알고리즘과 LCM 알고리즘을 확장한다.

[그림 1]에서 삽입 위치는 우선 {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.

3.1 수명적 최적화

수명적 최적화를 위한 알고리즘은 우선 LCM 변환을 수행하고 $Comp_SpCM_{Profit}()$를 수행한다. 그리고 최소 결합 집합 STS(Smallest Tight Set)를 계산하여 수명적으로 최적인 삽입 위치 Ins_SpCM_{co}를 계산한다.

```
procedure Comp_SpCMco( )
 begin
  LCM();  // LCM_DELABLE(), LCM_LATEST(),
          //                LCM_ISOLATED(),
LCM_Insert_Replace()
  Comp_SpCMProfit();
  STS();
 end;
```

[알고리즘 1] S_pCM_{co} 알고리즘
[Algorithm 1] A S_pCM_{co} algorithm

3.1.1 LCM

[알고리즘 2]는 배정문이 지연될 수 있는 위치를 결정한다. [알고리즘 2]에서 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 삽입할 것인지 결정한다. 이것은 불필요한 코드 모션을 억제하고 삽입된 계산의 수를 최소화시킨다.

```
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 ||
 _DELAYABLE[i] && !FlowG_node[i].USED
       && !FlowG_node[i].BLOCKED
   end
 end;
```

[알고리즘 2] 배정문 지연 알고리즘
[Algorithm 2] A Delayed Algorithm of assignment statement

```
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 3] 블록된 배정문 지연 알고리즘
[Algorithm 3] A Delayed Algorithm of blocked assignment statement

두 번째로 블록된 배정문을 지연시킬 위치를 결정한다. [알고리즘 3]에서 N-LATEST와 X-LAT EST는 배정문이 지연될 수 있는 위치들 중에서 배정문이 블록되거나 사용된 위치를 결정한다.

LATEST는 DELAYABLE의 값이 참일 경우에만 참일 수 있으므로 DALAYABLE의 값이 참인 노드에 대해서만 LATEST를 계산한다.

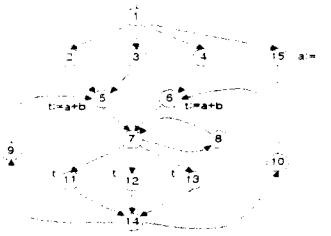
LATEST의 위치를 결정하기 위해서는 그 노드에 대한 상속자의 입력 부분이 배정문을 지연시킬 수 있는 위치인지 아닌지에 대한 정보를 이용하게 된다. 따라서 LATEST는 마지막 노드에서 시작 노드로 제어흐름 반대 방향으로 분석해 나간다. 또한, LATEST의 위치 결정은 DALAYABLE이 참인 경우에 한해서 계산된다.

세 번째로, 프로그램의 실행 시간을 향상시키는 코드 모션일지라도 불필요한 임시변수 초기화를 실행할 수 있다. 이러한 결점을 피하기 위해서 [알고리즘 4]에서 독립 수식을 결정하여 임시변수의 삽입을 제한한다.

```
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]);
         SOL_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 4] 독립 수식 위치 결정 알고리즘
[Algorithm 4] A location decision algorithm of independence expression



[그림 2] LCM 변환의 결과

[Fig 2] Result of LCM transformation

네 번째로, LCM 변환의 삽입 위치에 임시변수를 도입한 배정문을 삽입하는 과정이 있다.

[그림 1]에 LCM 변환을 적용하면 [그림 2]와 같다.

3.1.2 최소 결합 집합

```
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 SpCMco=(DnComp∪Γ(Ts(S)))-Ts(S)
 end
```

[알고리즘 5] 최소 결합 집합 알고리즘

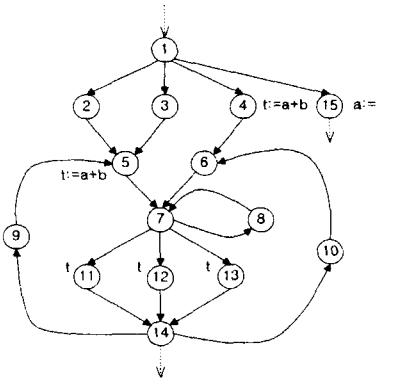
[Algorithm 5] An Algorithm of smallist tight set

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

[알고리즘 5]는 최소 결합 집합을 계산한다.

[그림 6]의 이분 그래프에 대한 최소 결합 집합을 [알고리즘 5]로 계산한 결과는 $T_s(S)$={5, 6} 이다. 따라서, 삽입 위치는{2, 3, 4}이다.

[그림 2]에 수명적 최적화를 적용하면 [그림 3]과 같다.



[그림 3] 최적의 희소 코드 모션 결과

[Fig. 3] Result of optimal sparse code motion

3.2 계산적 최적화

```
procedure Comp_SpCMProfit( )
 begin
  BCM_Predicate();      //      BCM_SAFE(),
BCM_EARLIEST(),
  Bipar_Graph();
  LTS();
 end;
```

[알고리즘 6] S_pCM_{Profit} 알고리즘

[Algorithm 6] A S_pCM_{Profit} algorithm

계산적 최적화를 위한 알고리즘은 우선 실제적인 코드모션을 제외한 BCM의 술어들을 계산하고, BCM 술어들을 이용하여 이분 그래프

를 작성하여 최대 결합 집합 LTS(Largest Tight Set)을 계산한다. 그리고, 마지막 단계로 최대 결합 집합을 이용하여 계산적 최적화를 위한 삽입 위치를 계산한다.

3.2.1 BCM

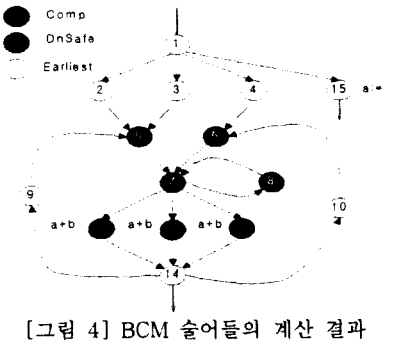
```
procedure BCM_SAFE( )
 begin
  for i := 0 to FlowG_node_MAX do
  begin if (FlowG_node[i] == E_node) then
    FlowG_node[i].X_D_SAFE := FALSE;
   else if (FlowG_node[i] == S_node) then
    FlowG_node[i].N_U_SAFE := FALSE;
  end;
  for i := FlowG_node_MAX downto 1 do
   begin
    for m := SAFE_Succ_Start(i) to SAFE_Succ_End(i) do
     begin
     SAFE_Succ_Sum := SAFE_Succ_Sum &&
                      FlowG_node[m].N_D_SAFE;
     FlowG_node[i].N_D_SAFE := FlowG_node[m].N_COMP ||
      FlowG_node[i].TRANSP && FlowG_node[i].X_D_SAFE;
     FlowG_node[i].X_D_SAFE:= FlowG_node[m].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 7] 안전한 삽입 위치 결정 알고리즘
[Algorithm 7] An Algorithm that decision safe insert location

BCM에서 프로그램의 각 변수의 의미적 보증을 위해 삽입은 안전해야 하며 코드 모션 후보도 옳게 재배치되어야 한다. 프로그램 내의 안전한 삽입 위치를 결정하는 알고리즘은 [알고리즘 7]과 같다. [알고리즘 7]은 임시변수의 삽입 위치부터 재배치되는 위치까지의 경로에 계산식에 대한 새로운 값의 도입이 있어서는 안 됨을 의미한다.

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

[알고리즘 8] 가장 이른 삽입 위치 결정 알고리즘
[Algorithm 8] An Algorithm that decision earliest insert location



[그림 4] BCM 술어들의 계산 결과
[Fig. 4] Result that computing of BCM predicates

코드 모션에서 삽입 위치는 안전한 삽입 위치 중에서 가장 이른 삽입위치에 임시변수 t를 삽입한다. 가장 이른 삽입 위치를 결정하는 알고리즘은 [알고리즘 8]과 같다.

[그림 1]에 BCM 술어들을 적용하면 [그림 4]와 같다.

3.2.2 이분 그래프

무향 그래프 (N, E)에서 N는 노드를 E는 가지를 의미한다. n∈N인 노드의 이웃하는 노드 Γ(n)는 $\Gamma(n)=_{df}\{w|\{n,w\}\in E\}$로 정의된다. E에 포함되는 모든 가지 e에 대해 N=S∪T이고 e∩S≠Φ∧e∩T≠Φ인 노드 S와 T의 집합이 두 개로 분리된다면, 무향 그래프 (N, E)는 (S∪T, E)의 두 부분으로 나누어 질 수 있다. [알고리즘 9]는 이분 그래프(bipartite graphs)를 작성하는 알고리즘이다. [알고리즘 9]에서 T_{DS}를 상위 계층이라 하고, S_{DS}를 하위 계층이라 한다.

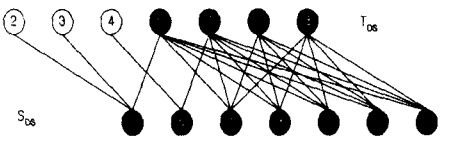
```
procedure Bipar_Graph()
 begin
  SDS := DnSafe ∪ !(UpSafe ∪ Earliest);
  TDS := DnSafe ∪ !(UpSafe ∪ Comp);
  for n:=0 to n<SDS_Max do
   Con_Edge(TDS[m], DnSafe(pred(SDS[n])))
 end;
```

[알고리즘 9] 이분 그래프 작성 알고리즘
[Algorithm 9] A constructing algorithm of bipartite graphs

[그림 4]의 흐름 그래프에 [알고리즘 9]를 적용하여 T_{DS}와 S_{DS}를 계산하여 구성된 이분 그래프는 [그림 5]와 같다.

<계층 구성 결과>
S_{DS} = {5, 6, 7, 8, 11, 12, 13}
T_{DS} = {2, 3, 4, 5, 6, 7, 8}

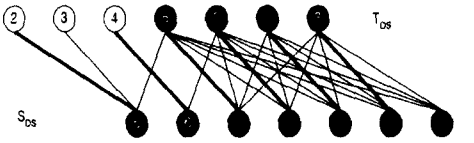
<가지 구성 결과>
ρ(pred(5)) = ρ{2, 3} = {2, 3, 5}
ρ(pred(6)) = ρ{4} = {4, 6}
ρ(pred(7)) = ρ{5, 6, 8} = {5, 6, 7, 8}
ρ(pred(8)) = ρ{7} = {5, 6, 7, 8}
ρ(pred(11)) = ρ{7} = {5, 6, 7, 8}
ρ(pred(12)) = ρ{7} = {5, 6, 7, 8}
ρ(pred(13)) = ρ{7} = {5, 6, 7, 8}



[그림 5] 예제 흐름그래프에 대한 이분 그래프
[Fig. 5] The bipartite graphs for example flow graph

[그림 4]의 흐름 그래프에 [알고리즘 8]을 적용하여 S_{DS}와 T_{DS}를 계산하여 구성된 이분 그래프는 [그림 5]와 같다.

3.2.3 매칭과 최대 결합 집합



[그림 6] 이분 그래프에 대한 최대 매칭
[Fig. 6] Maximum matching for bipartite graphs

M에 속한 가지 e_1, e_2이 $e_1\cap e_2=\Phi$라면, M⊆E인 가지들의 집합을 매칭(matching)이라 한다. 어떤 e∈M에 대하여 n∈e라면, 노드 n는 M에 의해 매치 된다. 어떤 매칭 M'⊆E에 대해 |M|

≥|M'|라면, M은 최대 매칭(maximum matching)이다.

[그림 6]의 굵은 가지는 [그림 4]의 최대 매칭을 나타낸다.

```
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);
     T_L(S) := S_Match
    end;
    Ins SpCM Profit = (DnComp⋃Π(T L(S))) − T L(S
    end;
```

[알고리즘 10] 최대 결합 집합 알고리즘

[Algorithm 10] An Algorithm of maximum tight set

최대 매칭은 이분 그래프의 최대 결합 집합을 계산하는데 중요하게 사용된다. [알고리즘 10]은 이분 그래프와 최대 매칭 M을 입력받아 최대 결합 집합 LTS를 계산한다.

[그림 6]의 이분 그래프에 대한 최대 결합 집합을 [알고리즘 10]으로 계산한 결과는 $T_L(S)$ = {6, 7, 8, 11, 12, 13}이다. 따라서, 삽입위치는 {4, 5}이다.

4. 성능 분석

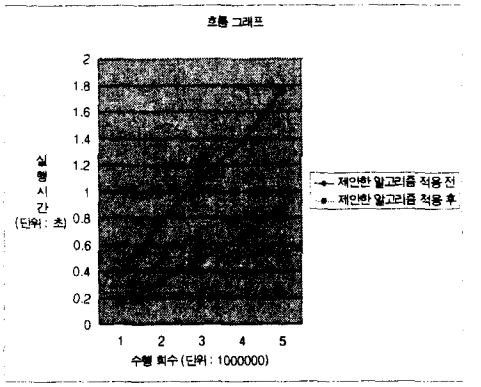
본 논문에서 제안한 희소 코드 모션 알고리즘의 성능 분석 결과는 다음과 같다.

[그림 1]과 [그림 3]의 예제 흐름 그래프에 제안한 알고리즘을 각각 전용하기 전과 적용시킨 결과를 나타내면 [표 1]과 [그림 7]과 같다.

<표 1> 알고리즘 수행 결과

<Table 1> Executive result of an Algorithm

구분 / 수행회수	적용 전	적용 후
1	0.38	0.16
2	0.77	0.33
3	1.10	0.49
4	1.37	0.66
5	1.76	0.88



[그림 7] 알고리즘 수행 결과

[Fig. 7] Executive result of an Algorithm

<표 1>에서 제안한 알고리즘을 [그림 1]에 적용하기 전과 [그림 3]에 적용시킨 흐름 그래프에 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.

5. 결론

본 논문에서는 계산 능력과 레지스터 과부하

에 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, 제안한 알고리즘의 동작 과정을 구체적으로 제시하였다. 제안한 알고리즘의 성능평가를 해 본 결과 프로그램의 불필요한 재 계산이나 재 실행을 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*, of *ACM SIGPLAN Notices*, Vol. 29, No. 6, pp. 159-170, Orlando, FL, June 1994.

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

[6] 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, ACM SIGPL-AN Notices*, Vol. 27, No. 7, pp.212-223, San Francisco, CA, June 1992.

[7] 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.

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

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

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

[11] 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.

[3] 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.

[4] 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.

[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

심 손 권

1996년 2월 관동대학교 전자계산공학과 졸업(공학사)
1998년 2월 관동대학교 대학원 전자계산공학과 졸업(공학석사)
2003년 2월 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