論文
7-3-2

임베디드 시스템에서 네트워크 분할을 이용한 프로그램 최적화 (Program Osptimality Using Network Partiton in Embedded System)

최강희(Choi Kang Hee)[1] 신현덕(Sin Hyun Duck)[2]

요 약

본 논문에서는 프로그램 최적화를 위해 개선된 추론적 부분 중복 제거(SPRE) 알고리즘을 제안했다. 본 논문에서 제안한 SPRE 기법은 컴파일러의 프로필링 기법 등을 통해 얻어진 실행 빈도에 대한 정보를 이용하여 실행 속도 최적화를 수행한다. 제안하는 알고리즘의 첫 번째 목적은 프로그램 실행 시 요구되는 메모리의 감소이며 두 번째는 실행 시간을 감소시키는 것이다. 단지 프로그램의 실행 속도만을 고려하는 경우에는 메모리 요구가 크게 증가하기 때문에 메모리 감소에 대한 고려도 중요하다. 이것은 프로그램을 실행하는데 요구되는 메모리의 크기가 실행 속도 보다 더 중요한 임베디드 시스템에 적합한 최적화 기법이다.

본 논문에서는 제어흐름그래프를 네트워크로 구성하여 분할하는데 사용하는 Min-Cut 알고리즘을 구현한다.

ABSTRACT

This paper improves algorithms of Speculative Partial Redundancy Elimination(SPRE) proposed by Knoop et al. Improving SPRE algorithm performs the execution speed optimization based on the information of the execution frequency from profiling and the memory space optimization. The first purpose of presented algorithm is to reduce in space requirements and the second purpose is to decrease the execution time. Since too much weight on execution speed optimization may cause the explosion of the memory space, it is important to consider the size of memory. This fact can be a big advantage in the embedded system which concerns the required memory size more than the execution speed. In this paper we implemented the min-cut algorithm, and this algorithm used the control flow graph is constructed with network and partitioned.

논문접수 : 2006. 5. 10.
심사완료 : 2006. 5. 25.

1) 정회원 : 대원과학대학 컴퓨터정보계열 교수

2) 정회원 : 관동대학교 컴퓨터공학과 겸임교수

1. 서 론

프로그램의 부분 중복 제거(PRE : Partial Redundancy Elimination)는 현대 컴파일러에서 사용하는 표준화된 최적화 기술이다. 대부분의 부분 중복 제거 알고리즘은 코드모션[6]을 포함하지만 코드의 실행 빈도와 같은 정보 측면은 고려하지 않는다.

Stone은 두 개의 프로세서에 프로세스들을 할당하기 위한 문제에 대해 s-t min-cut 해법을 제시했다[10]. Stone이 제기한 문제점은 서로 다른 실행 시간을 요구하는 프로세스들을 서로 다른 실행 속도를 갖는 두 개의 프로세서에 최소 비용으로 할당하는 문제다.

추론 부분 중복 제거(SPRE : Speculative Partial Redundancy Elimination)는 부분 중복 제거에 추론을 통한 부분 중복제거라는 새로운 접근법을 도입하여 수행하는 기법이다[2,3,5]. 추론 부분 중복 제거는 min-cut 알고리즘 사용을 제안한 Horspool과 Ho[5]에 의해 처음 소개되었으나 이 알고리즘은 최적의 해법을 찾지 못했다. Bodik은 추론 부분 중복 제거의 이 문제에 대한 해법을 설명했으나 알고리즘의 최적화를 위해 만든 해법은 증명도 없고 실험 결과도 없다. 추론 부분 중복 제거는 Gupta, Bodik과 Soffa에 의해 연구되었고[1,3], 2004년에 Scholz, Horspool과 Knoop에 의해 메모리와 실행 속도 최적화를 위한 알고리즘이 제안됐다[9].

부분 중복 제거의 기존 접근법[7,8]에서는 프로그램의 수행 속도는 빨라질 수 있으나 크기가 커질 수 있는 문제를 제어할 수 없다. Hailperin은 전체적 변환에 비용 함수를 도입하는 부분 중복 제거 접근법을 제안하였다[4]. 이 접근법은 부분 중복 제거 변환에서의 상수 전파와 감축을 병행하는데 코드 크기의 문제는 제어할 수 없다.

본 논문에서는 임베디드 시스템에서 사용되는 응용 프로그램의 최적화를 목적으로 네트워크 분할을 이용한 추론 부분 중복 제거 기법에 대한 연구를 수행하고자 한다. 본 논문에서 제안한 네트워크 분할 알고리즘은 네트워크로 구성된 제어 흐름 그래프를 실행 속도 최적화, 메모리 최적화, 실행 속도/메모리 최적화의 최적화 분야별로 분할하여 부분 중복 제거를 수행한다.

2. 추론 부분 중복 제거의 이론적 배경

2.1 Correctness 정의

추론 부분 중복 제거 SPRE에서는 프로그램 변환을 위해, 임시 변수 t를 도입한다. t는 계산식 e의 값을 다시 사용하기 위해 보존하는데 사용된다. 제어 흐름 그래프의 모든 노드 $u \in N$는 I_u로 표시하는 노드의 입구와 O_u로 표시하는 노드의 출구 부분으로 구분된다. 제안하는 알고리즘은 계산 e가 프로그램의 이들 각 입구/출구 지점에서 사용 가능하도록 할 것인지 아닌지를 결정한다. 이 분석이 노드 u의 입구에서 계산 e가 사용 가능하도록 결정했다면, 노드 u에서 e를 재계산하는 대신에 변수 t로 재배치 될 수 있다.

입구와 출구를 표시하는 레이블은 2|N|이며, 제안하는 알고리즘에 의해 이들을 네트워크로 구성하여 두 영역 *Available*과 $\neg$*Available*로 분할할 것이다. *Available*은 변수 t에 의해 e가 사용 가능하다는 것을 나타내며 $\neg$*Available*은 *Available*을 제외한 나머지 노드들이다.

SPRE에 의한 프로그램 변환은 [정의 1]을 만족해야 한다[11].

[정의 1] *Correctness*

$\forall p = \langle u_1, \ldots, u_k \rangle \in Path(s,f): \forall i(1 \le i \le k):$

$replace(u_i) \Rightarrow$

$\{(\exists j(1 \le j \le i): store_M(u_j): \forall k(j \le k < i): COMP(u_k) \vee NULL(u_k))$

$\vee (\exists j(1 \le j \le i): store_E(u_j): \forall k(j < k < i): COMP(u_k) \vee NULL(u_k))$

2.2 SPRE 비용함수

SPRE에서 각 노드의 실행 비용을 계산하기 위한 비용함수는 다음과 같다.

$$f = \sum_{u} cst_u(i_u, o_u)(\alpha frq(u) + \beta spc(u))$$

frq(u)는 노드 u의 실행 빈도, 즉 식의 수행 횟수를 의미한다. 그리고 spc(u)는 노드 u에 삽입으로 초래되는 메모리 공간 비용 즉, 식의 계산 수를 의미한다. spc(u)는 식을 평가하는데 필요한 명령 바이트의 수를 의미하는 상수값을 갖는다. α와 β는 실행 시간 최적화와 메모리 공간 최적화의 수행 여부를 결정한다. α가 0인 경우는 메모리 공간만을 위한 최적화가 수행됨을 의미하고 β가 0인 경우는 속도만을 위한 최적화가 수행됨을 의미한다.

<표 1>은 제어 흐름 그래프를 네트워크로 구성해서 *Available*과 ¬*Available* 영역으로 분할한 후, 각 노드의 비용을 나타낸다[11].

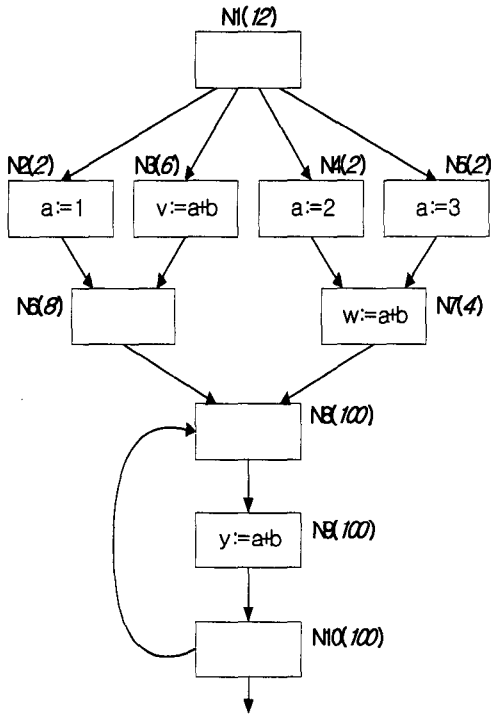
<표 1> 노드 종류별 비용 계산

COMP	$cst_u(i_u, o_u) = \begin{bmatrix} 1, i_u \in \bar{A}_e \\ 0, i_u \in A_e \end{bmatrix}$
MOD	$cst_u(i_u, o_u) = \begin{bmatrix} 1, o_u \in A_e \\ 0, o_u \in \bar{A}_e \end{bmatrix}$
NULL	$cst_u(i_u, o_u) = \begin{bmatrix} 1, i_u \in \bar{A}_e \wedge o_u \in A_e \\ 0, otherwise \end{bmatrix}$

3. 네트워크 분할 알고리즘을 이용한 부분 중복 제거

3.1 네트워크 구성

[그림 1]의 제어 흐름 그래프에서 최적화 되어야할 식은 a+b다. 이 제어 흐름 그래프의 노드는 N1에서 N10까지 번호가 부여되었으며, 프로필링(profiling) 기법에 의해 얻어진 실행 빈도를 각 노드 번호 오른쪽 괄호에 나타냈다. 이 흐름 그래프는 식 a+b에 대한 3개의 계산을 가지며 이 식에 대해 총 110회의 수행 횟수를 갖는다.



[그림 1] 제어 흐름 그래프

SPRE 알고리즘에서 사용되는 레이블 I_u와 O_u는 Stone이 제안한 네트워크의 노드와 일치시키기에 알맞으며 따라서, 프로그램에 Stone의 s-t min cut 감축을 적용하여 두 개의 영역으로 분할 할 수 있다.

본 논문에서 사용하는 제어 흐름 그래프의 각 노드는 식 e의 값을 수정하는 노드 MOD와 식 e를 계산하는 노드 COMP 그리고 식 e를 계산하지도 않고 e의 값을 수정하지도 않는 노드 NULL의 3가지 경우로 구분한다. [그림 1]의 N2, N4, N5는 MOD 노드이고 N3, N7, N9는 COMP 노드이며 나머지 노드들은 NULL 노드다.

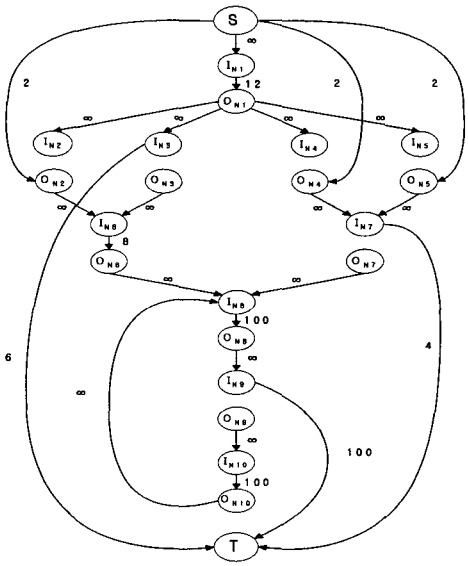
[알고리즘 1]은 s-t 네트워크를 구성하는 알고리즘으로 제어 흐름 그래프를 입력받아 s-t

네트워크를 출력한다.

```
procedure construct_network()
  begin
    add_node(S);
    add_node(T);
    weight_edge(S, node1.n) = ∞;
    for i := 1 to E_node do
          begin
weight_edge(nodei.x, (succ(nodei)).n) = ∞;
          case modification :
weight_edge(S, nodei.x) = nodei.frq;
          case computation :
weight_edge(nodei.n, T) = nodei.frq;
          case null :
weight_edge(nodei.n, nodei.x) = nodei.frq;
           end;
end
```

[알고리즘 1] 네트워크 구성 알고리즘

제어 흐름 그래프의 각 노드의 입구와 출구들은 min-cut 알고리즘에 의해 *Available*과 ¬*Available* 영역으로 분할될 것이다. 따라서 프로그램의 시작 부분에서는 계산식이 사용가능할 수 없기 때문에 시작 노드 S의 입구 레이블은 ¬*Available*의 요소가 되어야한다. 제어 흐름 그래프에 대한 s-t 네트워크를 구성하기 위해 [그림 2]와 같이 가지들을 구성한다. [알고리즘 1]에 따라 제어 흐름 그래프의 입구 노드인 N1에 대해서 가지 (S, I_{N1})에 무한대의 가중치를 부여하여 삽입하고, 제어흐름 그래프의 각 가지 (u,v)∈E와 일치하는 네트워크의 가지 (O_u, I_v)에 무한대의 가중치를 부여하여 삽입한다. 그리고 각각의 MOD 노드 u에 대해서 노드의 u의 실행 빈도 frq(u)를 가지 (S, O_u)에 부여하여 삽입하고, 각 COMP 노드 u에 대해서 frq(u)를 가지 (I_u, T)에 부여하여 삽입한다. 마지막으로 각 NULL 노드 u에 대해서 frq(u)를 가지 (I_u, O_u)에 부여하여 삽입한다.



[그림 2] s-t 네트워크

3.2 네트워크 분할 알고리즘

3.2.1 비용함수 알고리즘과 비용계산 알고리즘

[알고리즘 1]에 의해 구성된 s-t 네트워크의 노드들의 가지 가중치를 이용하여 각 노드의 실행 비용을 계산하기 위한 비용 함수와 비용 계산 함수는 [알고리즘 2]와 [알고리즘 3]과 같다.

```
function cost_func(cut_edge_line)
  begin
for i := 1 to |cut_edge_line| do
  begin  if (α> 0 && β> 0) then
cost_value = cost_value +
      (cost(cut_edgei) *
(frequency(cut_edgei) +
          byte_instruction(exp)));
   else if (α== 0) then
    cost_value = cost_value +
 (cost(cut_edgei) * byte_instruction(exp));
   else (β== 0) then
    cost_value = cost_value +
(cost(cut_edgei) * frequency(cut_edgei));
  end;
 end
```

[알고리즘 2] 비용 함수 알고리즘

```
function cost(edge_set)
  begin
for i := 1 to |edge_set| do
  begin
            node_i = edge_set.node_i;
            case modification :
if (node_i.x == Available)
   then return_cost = 1;
   else return_cost = 0;
case computation :
if (node_i.n == ¬Available)
    then return_cost = 1;
    else return_cost = 0;
case null :
if (node_i.n ==¬Available)&&(node_i.x == Available)
   then return_cost = 1;
   else return_cost = 0;
  end;
 end
```

[알고리즘 3] 비용 계산 함수 알고리즘

3.2.2 Min-cut 알고리즘

비용 함수에 의한 실행 비용 계산 결과를 이용하여 s-t 네트워크의 각 노드를 *Available*과 ¬*Available* 영역으로 분할하기 위한 [알고리즘 4]를 제안한다. [알고리즘 4]는 s-t 네트워크를 입력받아 min-cut 네트워크를 출력한다.

```
function partitioning()
begin
for i := 1 to E_node do
  net_node = read(s_t_network);
while (net_node != S || net_node != T) do
  begin
            s_t_node_i = net_node_i.n;i++;
            s_t_node_i = net_node_i.x;i++;  j++;
  end;
source_nodes = s_t_node_1;
Best_value = ∞;
while (source_nodes != s_t_node) do
  begin
sink_nodes = s_t_node - source_nodes;
cut_edge_line=s_t_cut(source_nodes, sink_nodes);
Unit_value = cost_func(cut_edge_line);
if (Unit_value < Best_value) then
 begin
            min_cut_line = cut_edge_line;
            source_node = s_t_node_next  end;
              end
end
```

[알고리즘 4] 네트워크 분할 알고리즘

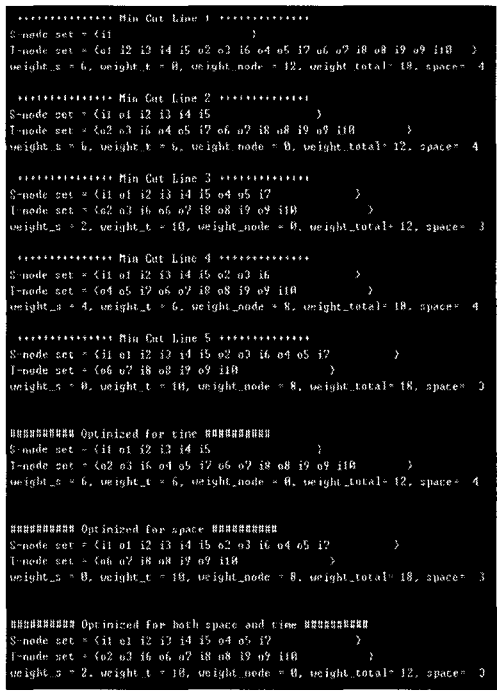
[알고리즘 4]는 [알고리즘 2]와 [알고리즘 3]을 이용하여 네트워크 가지의 가중치를 계산한다. [알고리즘 5]는 cut_edge_line의 각 가지들의 비용을 계산하기 위해 해당 가지 집합을 구하는 함수다.

```
procedure s_t_cut(s_nodes, t_nodes)
 begin
cut_edges = edge(S, t_nodes);
cut_edges = cut_edges + edge(s_nodes, T);
cut_edges = cut_edges + edge(net_node);
 end
```

[알고리즘 5] 가지 집합 알고리즘

[그림 3]은 제안한 알고리즘을 구현하여 [그림 1]에 적용한 결과 화면이다.

[그림 1]의 프로그램에서 가지의 가중치가 무한대인 경우를 제외한 가능한 min-cut은 5개이며 최적화 분야별 min-cut은 [그림 3]의 프로그램 실행 결과와 같다.

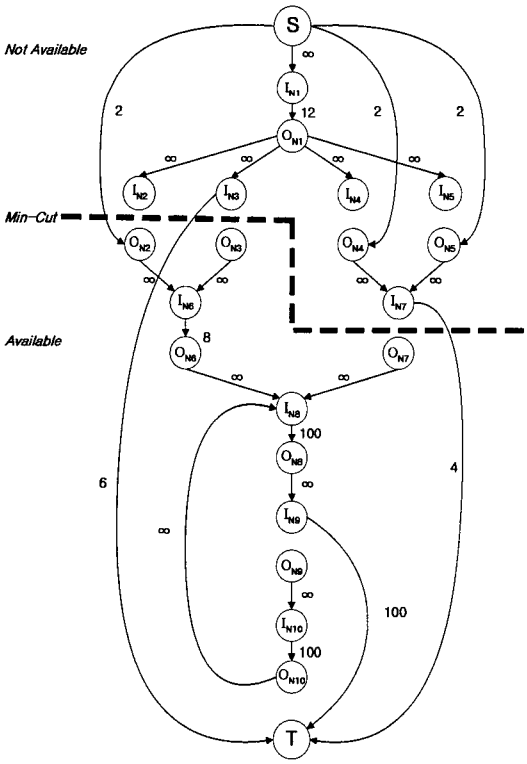


[그림 3] 프로그램 실행 결과

[그림 3]의 결과에서 실행속도/메모리 최적화 (Optimized for both space and time)로 분할된 네트워크의 *Available* 영역은 {S, I_{N1}, O_{N1}, I_{N2}, I_{N3}, I_{N4}, O_{N4}, I_{N5}, O_{N5}, I_{N7}}이고 ¬*Available* 영역은 나머지 노드들 {O_{N2}, O_{N3}, I_{N6}, O_{N6}, O_{N7}, I_{N8}, O_{N8}, I_{N9}, O_{N9}, I_{N10}, O_{N10}, T}이다. [그림 4]는 [그림 2]에 [그림 3]의 결과를 적용한 결과이다.

3.3 부분 중복 제거

[알고리즘 6]은 [알고리즘 4]에 의해 구축된 min-cut 네트워크를 이용하여 부분 중복 제거를 수행하는 알고리즘이다.
[그림 5]는 [그림 4]에 [알고리즘 6]을 적용하여 실행속도/메모리 최적화를 수행한 결과를 나타낸다.



[그림 4] 네트워크 분할 결과

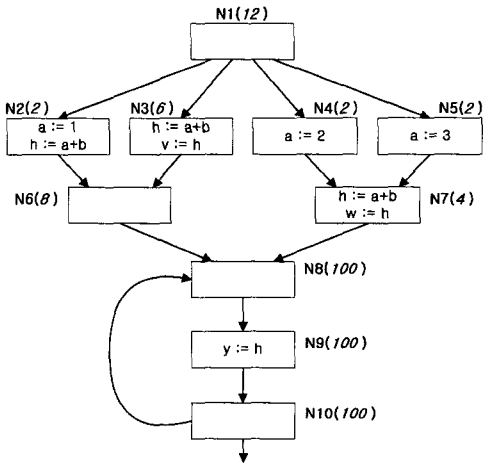
[그림 5]의 흐름 그래프는 식 a+b에 대한 3개의 계산과 12회의 수행 횟수를 갖는다. 이 결과는 [그림 5]의 프로그램에 비해 속도와 메모리 공간 모두에서 향상되었고 메모리 사용이 중요한 임베디드 시스템 응용 프로그램을 위한 적합한 최적화이다.

4. 성능 분석

이 장에서는 제안한 알고리즘에 대한 성능 평가를 수행하여 최적화 수행 전 프로그램과 비교하고 Knoop의 알고리즘과 제안한 알고리즘의 적용 결과를 비교 분석한다. 성능 분석을 위해 Windows 시스템에서 Visual Studio .NET 2003과 Visual C++ 6.0을 이용하였으며 요구 메모리 측정을 위해 개발한 프로그램을 이용했다. [그림 6]은 예제 프로그램에 대한 요구 메모리를 측정하기 위해 구현한 프로그램의 실행 결과다.

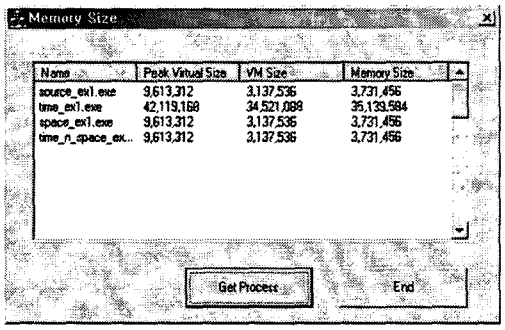
```
procedure SPRE_action()
begin
while ((node = read(min_cut_network) != Null)
do
begin      case modification :
 if (node.x == Available)
       then Insert_x(node);
case computation :
 if (node.n == ¬Available) && (node.x ==
Available) then
  begin Insert_n(node);
       Replace(node); end;
else if (node.n == Available) then
Replace(node);
case null :
if (node.n == ¬Available) && (node.x ==
Available) then
Insert(node); end end;
```

[알고리즘 6] 부분 중복 제거 알고리즘



[그림 5] 최적화 수행 결과

본 논문에서 제안한 SPRE의 개선된 알고리즘을 적용시킨 결과를 나타내면 <표 2>과 같다. <표 2>은 실행 속도 최적화, 메모리 최적화와 실행 속도/메모리 최적화로 구성되며 각 최적화 분야에 따른 실행 시간과 요구 메모리를 측정하였다. 실행 시간의 경우는 각 최적화 분야 전체에서 개선된 것으로 측정되었다. <표 2>의 측정 결과를 실행 시간과 요구 메모리로 구분하여 그래프로 나타내면 [그림 7]와 [그림 8]과 같다.



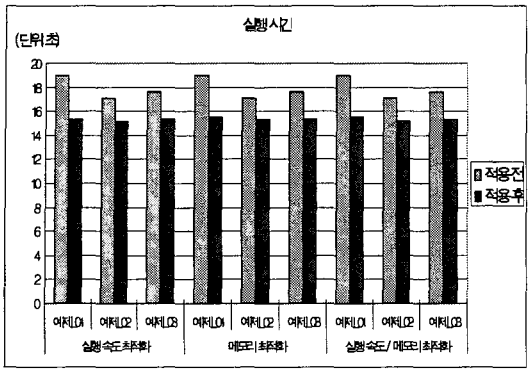
[그림 6] 메모리 측정 프로그램

요구 메모리의 경우는 최적화 분야에 따라 큰 차이를 보였다. 예제_01의 실행 속도 최적화 경우, 알고리즘을 수행하기 전 3,644KB이던 요구 메모리가 48,672KB로 증가하였다. 속도만을 고려한 최적화는 메모리 요구가 크게 증가할 수 있기 때문에 메모리 감소에 대한 고려도 중요하다.

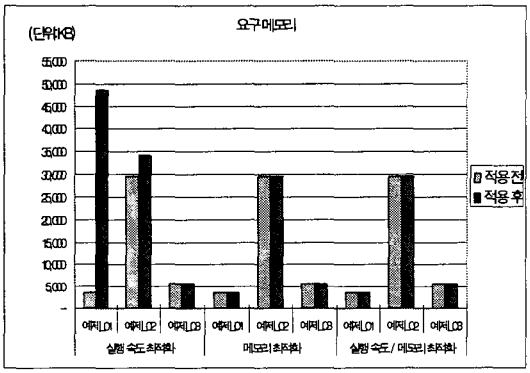
<표 2> 알고리즘 수행 결과

(단위 : 시간(초), 메모리(KB))

제안한 알고리즘 \ 예제			예제_01	예제_02	예제_03
실행 속도 최적화	실행 시간	적용 전	17.14	19.01	17.63
		적용 후	15.21	15.32	15.38
		비율	0.887	0.806	0.872
	요구 메모리	적용 전	3,644	29,524	5,576
		적용 후	48,672	34,316	5,576
		비율	13.357	1.162	1.000
메모리 최적화	실행 시간	적용 전	17.14	19.01	17.63
		적용 후	15.32	15.55	15.38
		비율	0.894	0.818	0.872
	요구 메모리	적용 전	3,644	29,524	5,576
		적용 후	3,644	29,524	5,576
		비율	1.000	1.000	1.000
실행 속도 /메모리 최적화	실행 시간	적용 전	17.14	19.01	17.63
		적용 후	15.21	15.55	15.38
		비율	0.887	0.818	0.872
	요구 메모리	적용 전	3,644	29,524	5,576
		적용 후	3,644	29,524	5,576
		비율	1.000	1.000	1.000



[그림 7] 실행 시간 분석 결과



[그림 8] 요구 메모리 분석 결과

메모리 최적화를 수행한 경우는 상대적으로 실행 시간에 대한 효율이 낮아 질 수 있다. 예제_01의 실행 속도 최적화 결과는 실행 시간 비율이 0.887이고 메모리 최적화 결과는 0.894로 높아 졌다. 예제_03의 경우, 요구 메모리에 대한 최적화 분야별 알고리즘 적용의 결과가 동일하며 메모리 요구의 증가도 발생하지 않았다. 실행 속도/메모리 최적화의 경우 실행 속도 최적화에 비해 실행 속도가 느릴 수 있지만 메모리에 대한 요구는 더 적다. 예제_02의 경우, 실행 속도 최적화에서 15.32초인 실행 시간이 실행 속도/메모리 최적화에서는 15.55초로 증가했지만 메모리 요구는 34,316KB에서 29,524KB로 낮아졌다.

5. 결론

본 논문에서는 임베디드 응용 프로그램의 최적화를 위해서 네트워크 분할 알고리즘을 이용한 추론 부분 중복 제거 알고리즘을 제안했다. 제안한 네트워크 분할 알고리즘은 네트워크로 구성된 제어 흐름 그래프를 실행 속도 최적화, 메모리 최적화, 실행 속도/메모리 최적화의 최적화 분야별로 분할하여 부분 중복 제거를 수행한다.

제안한 알고리즘의 첫 번째 목적은 프로그램 실행 시 요구되는 메모리의 감소이며 두 번째는 실행 시간을 감소시키는 것이다. 단지 프로그램의 실행 속도만을 고려하는 경우에는 메모리 요구가 크게 증가하기 때문에 메모리 감소에 대한 고려도 중요하다.

성능 분석을 해본 결과 실행 속도 / 메모리 최적화의 경우 실행 속도 최적화에 비해 실행 속도가 느릴 수 있지만 메모리에 대한 요구는 더 적을 수 있다. 예제_02의 경우 실행 속도 최적화 적용 결과 15.32초인 실행 시간이 실행 속도/메모리 최적화 적용에서는 15.55초로 증가했지만 메모리에 대한 요구는 34,316KB에서 29,524KB로 약 14% 낮아졌다. 이것은 프로그램을 실행하는데 요구되는 메모리의 크기가 실행 속도에 비해 더 중요한 임베디드 시스템에 적합한 최적화 기법이다.

참 고 문 헌

[1] 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.

[2] Cai, Q. and Xue. J., "Optimal and efficient speculation-based partial redundancy elimination", *Proceedings of the 1st IEEE/A*

CM International Symposium on Code Generation and Optimization, pp. 91-102, 2003.

[3] Gupta, R., Berson, D. A. and Fang, J. Z., "Path profile guided partial redundancy elimination using speculation", *Proceedings of the 1998 International Conference on Computer Languages*, pp. 230-239, 1998.

[5] Hailperin, M., "Cost-optimal code motion", *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 6, pp. 1297-1322, November 1998.

[7] Horspool, R. N. and Ho, H. C., "Partial redundancy elimination driven by a cost-benefit analysis", *Proceedings of 8th Israeli Conference on Computer Systems and Software Engineering*, pp. 111-118, June 1997.

[8] 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.

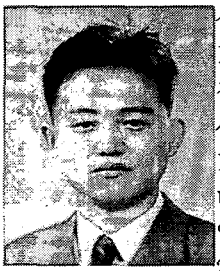
[9] Knoop, J., Rüthing, O. and Steffen, B., "Partial dead code elimination", In Proc. *ACM SIGPLAN Conference on Programming Language Design and Implementation'94*, of *ACM SIGPLAN Notices*, Vol. 29, No. 6, pp. 147-158, Orlando, FL, June 1994.

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

[11] 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.

[13] Stone, H., "Multiprocessor scheduling with the aid of network flow algorithms", *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, pp. 85-93, January 1977.

최강희
1988.3-1993.2 관동대학교 정보처리과 이학사
1993.3-1995.2 관동대학교 전자계산공학과 공학석사
1996.9-2000.2 관동대학교 전자계산공학과 공학박사
2001.3-2006.4 현재 대원과학대학 컴퓨터정보계열 교수
관심분야 :Mobile DB, 전자상거래



신현덕
1993.3-1998.2 관동대학교 전자계산공학과 졸업(공학사)
1998.3-2000.2 관동대학교 대학원 전자계산공학과 졸업(공학석사)
2000.3-2006.2 관동대학교 대학원 전자계산공학과 졸업(공학박사)
2006.4 현재 관동대학교 컴퓨터공학과 겸임교수
관심분야 : 컴파일러, 프로그래밍 언어, 코드 최적화, 임베디드 시스템