論文 91-28B-7-1

효율적인 그래프를 이용한 이차원 레이아웃 컴팩숀 알고리듬

(An Efficient Graph-Based Two-Dimensional Layout Compaction Algorithm)

申 鉉 哲*

(Hyun Chul Shin)

要 約

새로운 휴리스틱에 의존하는 2차원적인 심볼릭 레이아웃 컴팩션 방법이 개발되었다. 기존의 1차원적인 컴팩션 과정을 거친 후에, 레이아웃의 폭 또는 높이를 줄이기 위해 최장경로상에 있는 소자들이 재배치 된다. 이러한 과정에서 x와 y 두 방향의 위치 제약들이 동시에 고려되며, 계층적 컴팩션에서 설계의 데이타양을 줄이기 위해, 같은 종류의 cell은 컴팩션 후에도 같은 형태를 유지하도록 제한조건을 줄 수 있다. 이 알고리듬은 실험한 몇 개의 예에서, 다른 발표된 결과와 비교할 때, 최소의 면적을 보여주었다. 예상되는 수행시간은 1차원적인 알고리듬의 수행시간을 T_1이라할 때 $O(T_1)$이다.

Abstract

A new heuristic two-dimensional symbolic layout compaction approach is developed. After conventional one-dimensional compaction steps, all the components on the critical paths that define the height or width of the given layout are found and rearranged to reduce layout size. During this process, constraints in both x and y directions are considered and pitch-matching of ports for hierarchical compaction can be achieved to reduce the amount of the design data. This approach generated the smallest area for several examples we have tried when compared with other published results. The expected run time can be bouned by $O(T_1)$, where T_1 is the run time of a typical one-dimensional compactor.

I. 서 론

레이아웃 컴팩션은 물리적 설계(physical design)에 있어서 중요한 단계로, 회로의 동작과 성공적인 제작에 대한 소자간의 거리제한을 만족하면서 레이아웃의 면적을 줄여준다.

1차원 컴팩션 방법은 쉽고 효율적이라는 점 때문에 널리 이용되어 왔다[1,2,3,4]. 1차원 컴팩션 알고리듬들은 레이아웃 요소(component)들을 한번에 한 방향으로만 움직인다. 그러나, 소자들을 수평(x)과 수직(y)방향으로 움직임으로써 상당한 면적의 감소를 얻을 수 있는 경우가 많다. 하지만, 전역적(global) 2차원 컴팩션 문제는 NP-hard 문제[5,6]로 알려져 있으며, 매우 작은 문제가 아니고서는 최적의 해를 찾기가 불가능하다. 몇몇 저자들은 2차원 컴팩션 문제를 NP-hard 문제로 수식화하였으며, 탐색 공간을 줄이기 위한 휴리스틱한 방법들을 제안하였다[6,7]. 다른 방법으로는 1차원 컴팩션을 쓰고나서 레이아웃의 가로×세로비(aspect ration)를 바꾸기 위하여 최장경로(critical-path)

* 正會員, 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)
接受日字 : 1990年 3月 26日

(레이아웃의 넓이나 높이를 결정하는 경로)를 끊는 방법이 있다[8]. 하지만 이러한 방법은 직교방향에 대한 제한조건(예를들면, port pitch-matching)을 고려하지 않는다. 최근에는 레이아웃에서의 불필요한 영역을 '밀어내기(sweep)' 위하여 zone-refining 개념이 이용되고 있다[9]. 이 방법은 조금 느리기는 하지만 대부분의 벤치마크(benchmark)에 대하여 최소의 레이아웃을 생성하였다.[10] Mosteller[11]등은 임의의 곡선배선을 다룰 수 있는 simulated annealing에 기초한 컴팩션을 제안하였다. 이 방법을 실용화하기 위해서는 컴팩션 수행시간을 크게 줄여야할 것이다.

위에서 간략히 검토한 바와 같이, 그동안 발표된 컴팩션 알고리듬은 순수한 1차원적인 방법[1]으로부터 비교적 2차원적인 방법에 가까운것[11]등에 이르는 스펙트럼을 갖는다. 본 논문에서 제안한 컴팩션 방법도 엄밀한 의미에서의 2차원적인 방법은 아니며 1차원과 2차원 방법의 중간에 위치하지만, 컴팩션 중에 x와 y 두 방향의 거리제한이 동시에 고려되며 소자가 x와 y 양 방향으로 이동할 수 있다는 점에서 2차원 방법으로 명명하였다.

본 논문에서는 소자간의 거리 제한을 2차원 평면에서의 소자의 이동으로 만족시키며 최소면적의 레이아웃을 효율적으로 만들어내는 새로운 컴팩션 방법을 설명한다. 이 방법에서는 요소들을 x, y 두 방향으로 움직여서 한 방향에 대한 레이아웃의 크기가 최소가 되도록 하는데, 그렇게 함으로써 효율적인 알고리듬을 개발할 수 있다. 1차원 컴팩션이 행해지고 나면 모든 최장경로가 구해지고 이 경로들위의 요소들은 컴팩션하는 방향으로의 레이아웃의 크기를 줄이도록 재배열된다. 이 단계에서 컴팩션 방향에 수직인 방향의 제한조건도 함께 고려한다는 점은 주목할만하다. 이는 수직방향의 레이아웃 크기를 최소화(또는 증가시키지 않도록) 하며, 계층적 컴팩션동안 pitch-matching이 계층구조를 유지하도록 한다. 이점때문에, 계층적 레이아웃에서의 같은 리프셀(leaf cell) 내부의 모든 소자(instance)들이 컴팩션을 수행하는 동안 같은 형태를 유지하며, 따라서 설계 데이타가 컴팩션 후에 크게 증가하지 않는다. 이 방법은 MACS[12] 컴팩터에 이용되었으며, 앞서 Design Automation Confreence[13]에서 발표되었다.

2절에서는 이 방법의 개략적인 알고리듬을, 3절에서는 자세한 설명을 보였다. 4절에서는 계층적 컴팩션에 이 방법을 어떻게 적용하였는가를 설명하였고, 5절에서는 여러가지 실험결과를 기술하였다. 마지막으로 6절에서 결론을 맺는다.

Ⅱ. 개략적 알고리듬

보통 2차원 컴팩션은 일반적인 1차원 컴팩션 단계 이후에 행하게 된다. 특별히 언급하지 않는 한, y 방향으로 2차원 컴팩션이 수행된다고 가정한다. 여기서는 개략적인 알고리듬을 설명한다. x방향으로의 컴팩션은 y방향과 마찬가지이며, x와 y를 바꿈으로써 얻을 수 있다.

알고리듬의 설명에서, $G_y(N_y, E_y)$ ($G_x(N_x, E_x)$)는 y(x)방향에서의 제한 그래프를 나타내며, 여기서 N_y (N_x)는 y(x)방향 제한그래프 $G_y(G_x)$의 노드(node)의 집합을, $E_y(E_x)$는 y(x)방향의 거리 제한조건을 나타내는 arc의 집합을 나타낸다. 이와 유사하게 G_{yc} (N_{yc}, E_{yc})는 G_y의 임계(critical) 부분그래프(subgraph)이며, 여기서 E_{yc}는 레이아웃의 높이를 정의하게 되는 임계 arc들의 집합이다. 이들에 대해서는 다음 절에서 자세히 설명한다.

다음은 2차원 컴팩션 알고리듬에 대한 1회의 수행을 보여준다. 실제로는 이 알고리듬은 원하는 결과가 얻어지거나 더이상의 개선이 없을 때까지 반복해서 호출된다.

알고리듬 1. y방향의 2차원 컴팩션

Algorithm 1. Two-dimensional compaction in y direction.

```
{
  generate constraint graph Gy(Ny, Ey)in y;
  generate critical subgraph Gyc(Nyc, Eyc)in y;
  generate constraint graph Gx(Nx, Ex)in x;
  if(jogging is allowed)
    get relaxed graph Grx from Gx by removing
      tight constraints;
  else
    Grx=Gx;
  for(each arc in Eyc)
    find the inital "difficulty"value of removing it;
  * while a breakable cutset is found or
      there is not a feasible cutset  */
  cut_found=FALSE;
  while (cut_found= =FALSE){
      if(there is not a feasible cutset of Gyc)break;
      else {
        cut found=TRUE;
        find the minimum-difficulty feasible cutset;
        remove or relax the constraints in the cutset
          by moving components in  x  satisfying all the
          constraints in Grx;
        if (jogging is allowed){
          it (tight constraints of Gx can be satisfied
            by jogging)
```

```
            do jogging to satisfy all tight constraints;
          else{
            increase "difficulty"values of arcs in Gyc
              to infinity if corresponding constraint
              in Gx can not be satisfied;
            cut_found=FALSE;
          }
        }
      }
    }
    if (cut_found = = TSUE){
      do one-dimensional compaction in y;
      report area reduction;
    }
}
```

필요한 1차원 컴팩션 단계가 끝나면 알고리듬은 사용자가 정의한 'max-itr'만큼 반복하여 (정의하지 않은 경우는 9회) 위의 알고리듬을 호출함으로써 최장경로를 끊으려고 시도한다. 이 시도가 실패하는 경우는 두 가지가 있다. 첫번째는 면적의 감소가 없을 때이고, 두번째는 arc들에 breakable cutset이 없을 때이다. 위의 알고리듬에서 while 루프가 끝난 뒤에 'cut_found'가 'TRUE'이면, breakable cutset 이 찾아진 것이며 최장 경로를 끊기위한 소자의 수직방향으로의 이동이 행해진다. 즉, y방향으로 컴팩션하기위해 breakable cutset에 연결된 소자들이 x방향으로 이동된다.

하나의 cutset이 feasible 하다는 것은 G_{rx} 에서의 어떠한 거리 제한조건에도 위배되지 않으면서, G_{yc}로부터 cutset에 속하는 각 arc를 한 번에 한 개씩 제거할 수 있다는 것을 말한다. 만약 cutset의 모든 arc들이 G_x에서의 어떠한 제한조건도 위배하지 않는 x방향으로의 이동으로 G_{yc}에서 동시에 제거될 수 있다면, 이 cutset은 breakable하다고 한다. 예를 들어 그림1에서 $\{e_{13}, e_{23}\}$는 feasible cutset이지만 breakable cutset은 아니다. 또한 이 그림에서 v_1과 v_3는 x 또는 y방향으로 적어도 4단위(unit)만큼 떨어져야 한다. 이제 X_1를 v_1의 x좌표값이라 하자. $X_1=0, X_2=7, X_3=3.5$이면, e_{13}와 e_{23}는 G_{yc}에서 최소 거리 경계값(lower bound)4를 갖는 임계 arc들이다. 또 e_{12}와 e_{21}이 사용자에 의해 주어진 G_x에서의 제한조건이라 하자. Arc e_{12}는 v_1에서 v_2로의 최소 경계값 7을, e_{21}은 최대 거리 경계값(upper bound)7을 나타낸다. 즉, v_1에서 v_2 사이의 거리는 x방향으로 정확히 7이어야 함을 말한다. 여기서 v_3를 v_1의 오른쪽으로 $X_3-X_1 \geq 4$가 되도록 이동시키면 제한조건 e_{13}는 제거된다. 반면에 v_3를 v_2의 왼쪽으로 $X_2-X_3 \geq 4$가 되도록 이동시키면 제한조건 e_{23}가 제거된다. 그러므로 $\{e_{13}, e_{23}\}$는 feasible cutset이나, e_{13}와 e_{23}가 동시에 제거될 수는 없으므로 $\{e_{13}, e_{23}\}$는 breakable cutset은 아니다.

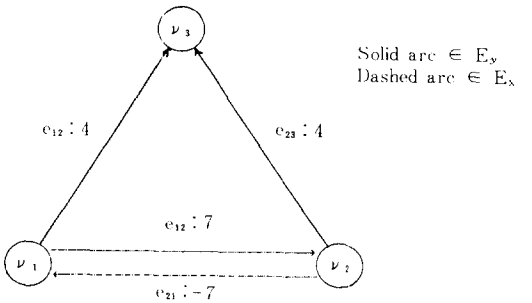


그림 1. Feasible이기는 하지만 breakable은 아닌 cutset $\{e_{13}, e_{23}\}$

Fig. 1. An example of a feasible but unbreakable cutset $\{e_{13}, e_{23}\}$.

위의 알고리듬에서는 먼저 cutset을 찾은 후 wire의 jogging을 써서 breakable 한가를 본다. feasible cutset이 breakable cutset이 아니면, unbreakable arc의 difficulty value가 증가하게 된다. 위의 예에서 e_{13}나 e_{23}중의 하나의 difficulty value가 증가하여 다음 단계에서의 feasible cutset은 e_{13}와 e_{23}중 하나만을 포함할 수 있고, 둘 다를 포함하지는 못하도록 한다. 여기에 대해서는 다음 절에서 좀 더 자세하게 설명한다.

Ⅲ. 알고리듬의 자세한 설명

본 절에서는 앞의 2절에서 설명된 2차원 컴팩션 알고리듬의 주요 부분들을 설명한다.

1. 거리 제한 그래프(Constraint Graph)

거리 제한 그래프에 대해서는 잘 알려져 있으므로[1,2,4,12], 여기에서는 간단히 설명한다. 레이아웃 요소는 제한 그래프에서는 노드로 표현되고, 두 요소 사이의 거리 제한조건은 상응하는 노드들간의 arc로 표현된다. 여기서는 2차원 컴팩션동안 두 방향의 제한조건을 고려하므로 두개의 제한그래프 $G_x(N_x, E_x)$와 $G_y(N_y, E_y)$가 유지된다. N_x는 모든 고정된(rigid) 요소들과 수직 배선 및 레이아웃의 좌 우 경계를 나타내는 두개의 특별한 노드(x_source와 x_sink)의 집합을 나타낸다. 마찬가지로, N_y는 모든 고정된(rigid) 요소들과 수평 배선 및 레이아웃의 상 하 경계를 나

타내는 두개의 특별한 노드(y_source와 y_sink)의 집합을 나타낸다. $E_x(E_y)$는 거리 제한을 나타내는 N_x (N_y)에서의 노드들간의 arc의 집합이다.

2. 임계 부분 그래프(Critical Subgraph)

임계 arc는 레이아웃의 크기를 증가시키지 않고는 거리 제한조건을 증가시킬 수 없는 arc이다. y(x)방향에서의 제한 그래프에서 모든 임계 arc들과 그에 접하는 노드들은 $G_y(G_x)$의 임계 부분 그래프인 G_{yc} (G_{xc})를 형성한다. 최장 경로는 $G_{yc}(G_{xc})$에서 y_source(x_source)로 부터 y_sink)까지의 경로이다.

만약 y_source(x_source)에서 y_sink(x_sink) 로의 최장 경로가 없어지도록 임계 arc들의 거리 제한조건들을 제거하거나 완화시킬 수 있다면, 레이아웃의 높이(폭)를 줄일 수 있다.

3. 완화된 제한 그래프(Relaxed Constraint Graph)

집적회로의 제조 기술에서는 주로 소자간의 최소 거리(lower bound)가 주어지지만 경우에 따라서는 소자간의 최대거리(upper bound)를 제한할 필요가 있다.

port pitch-matching에 의한 (또는 사용자가 정의한) 최대거리의 제한조건이 많으면 수평방향의 많은 제한조건에 의해 feasible cutset을 찾기가 어렵다. 예를 들어 어떤 port가 고정된 제한조건을 가지고 있으면, 그 port에 굳게(rigidly) 접해있는 모든 요소들은 전혀 움직일 수가 없다. 하지만 jogging wire를 쓰면 좀 더 자유롭게 붙어있는 요소를 움직일 수 있다. 따라서 우리는 feasible cutset을 찾을 때까지 최소 거리 제한조건만을 고려한다. 완화된 제한 그래프는 제한 그래프에서 모든 최대 거리 제한을 제거함으로써 얻어진다.

4. 어려운 정도의 결정(Difficulty Values)

한가지 중요한 결정사항은 완화시킬 제한조건의 집합을 어떻게 찾을 것인가 하는 것이다. 'difficulty value'는 제한 조건을 끊을 때 예상되는 끊기 어려운 정도를 나타낸다. 초기의 difficulty 값은 임계 제한조건을 제거하기 위해 요소가 수평방향으로 (여기서는 x방향) 움직여야 하는 거리로 정한다. 임계 그래프에서의 각 arc는 연관되는 소자들을 G_{rx} 에서의 거리 제한조건에 위배되지 않는 한 제한조건을 일으키는 요소의 왼쪽 또는 오른쪽으로 움직임으로써 제거할 수 있다. 예를 들면, 그림2에서 제한조건a를 제거하기 위해서는 요소B를 A의 오른쪽으로 d_1만큼, 또는 왼쪽으로 d_2만큼 이동시켜야 한다. 따라서 arc a를 끊기 위한 difficulty value는 d_1 또는 d_2 중 적은 값이 된다.

이러한 difficulty value들은 G_x에서의 거리 제한조건을 모두 만족시킬 수 없을 때에는 증가하게 된다.

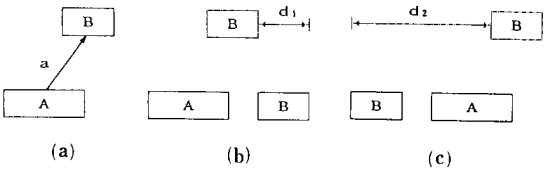


그림 2. Difficulty value의 계산
Fig. 2. Calculation of difficulty values.

5. Feasible cutset 찾기

임계 그래프에서의 cutset은 그것이 제거되었을 때 y_source와 y_sink 사이의 경로가 없어지도록 임계 그래프 G_{yc}를 분리시키는 arc의 집합이다. 레이아웃 면적을 줄이기 위해서는 cutset의 모든 arc (제한조건)들이 제거되거나 완화되어야 한다. arc들의 difficulty value의 합이 최소가 되는 최적 cutset을 찾는 문제는 잘 알려진 'max flow mincut theorem'을 이용함으로써 network flow 문제로 변환된다. max flow 문제를 풀이하는 complexity는 Sleator-Tarjan알고리듬[14]을 이용하면 0 (n m log n)이된다. 여기서 n은 그래프의 노드의 수, m은 arc의 수이다. 이 알고리듬을 사용하면 컴팩션 문제를 푸는데 너무 많은 CPU-time이 소요된다. 따라서 우리는 모든 가능한 cutset 중에서 그것을 끊는 difficulty의 최대값을 최소화하는 bottleneck mincut을 찾는다(앞으로 mincut이라 함은 이것을 의미한다). difficulty value는 단지 예상되는 값일 뿐만 아니라, difficulty value가 큰 한개의 arc를 제거하기보다 difficulty value가 작은 여러개의 arc를 제거하기가 더 쉬운 경우도 있으므로 bottleneck mincut을 이용하는 것을 전체 알고리듬에 나쁜 영향을 주지는 않는다.

mincut을 0 (m log n)시간내에 알아내는 새로운 알고리듬을 개발하였다. 이 알고리듬에서 노드는 그것이 소스쪽에 속하는가 싱크쪽에 속하는가를 표시하기 위한 플래그(flage)를 갖는다. 만약 어떤 노드가 소스쪽에 속하면, cutset의 arc를 지나지 않고 y_source에서 그 노드로 가는 경로가 존재한다는 것을 의미한다. 마찬가지로, 어떤 노드가 싱크쪽에 속하면 임계 그래프 G_{yc}에서 이 노드로부터 y_sink로의 어떤 경로가 존재한다. 어떤 arc는 '부유상태(floating)'임을 나타내는 2진수인 플래그를 갖는다. 처음에는

모든 arc가 '부유상태'가 아닌데, arc가 'check_fail'에서 처리되고, 그 arc에 인접한 'from' 노드와 'to' 노드가 소스쪽에 속하는지 싱크쪽에 속하는지 결정되지 않았으면, 그 arc는 '부유상태'가 된다. 만약 feasible cutset이 발견되면, 알고리듬은 그 cutset을 되돌려주고(return), 그렇지 않으면 공집합(empty set)을 되돌려 준다.

알고리듬 2. G_{yc}에서 최소의 difficulty 값을 갖는 feasible cutset 찾기

Algorithm 2. Finding a minimum-diffculty feasible cutset in G_{yc}.

```
{
  /* initialization */
  cutset=source_side=sink_side={ };
  source_side={y_source};
  sink_side={y_sink};

  for(each arc)
    mark that arc is not floating;

  sort critical arcs in decreasing order of difficulty;
  for(each arc in the order given above)
    if(check_fail (arc) = =TRUE) {
      cutset={ };
      return(cutset);
    }
  return(cutset);
}
check_fail(e_ij)
{
  /* e_ij is the arc from node n_i to node n_j */
  if (n_i ∈ source_side){
    if (n_j ∈ sink_side){
      if (difficulty of e_ij is INFINITY)
        return (TRUE);
      else
        cutset=cutset U {e_ij};
    }
    else {
      if (n_j ∈ source_side)
        connect (source_side, n_j);
    }
  }
  else if (n_i ∉ sink_side) {
    if(n_i ∉ sink_side)
      connect(sink_side, n_i);
  }
  else{
    mark that e_ij is floating;
  }
  return(FALSE);
}
connect(side, n_k)
{
  add n_k to side;
  for(each arc e_kj from n_k)
  {
    if(e_kj is not floating)
      continue;
    mark that e_kj is not floating;
    if(n_j ∉ side)
      connect(side, n_j);
  }
  for(each arc e_ik to n_k)
  {
    if(e_ik is not floating)
      continue;
    mark that e_ik is not floating;
  if(n_i ∉ side)
    connect(side, n_i);
  }
}
```

위의 알고리듬에서, n은 노드의 수이고 m은 arc의 수라고 할때, 0 (m log n)의 시간이 소요됨을 보이는 것은 쉽다. 여기서 주목할 점들은 다음과 같다.

1. 알고리듬3의 주 과정에서 각 노드와 arc가 0(1)시간에 처리되므로 임계 arc들을 정렬(sorting)하는데 0 (m log n)시간이 소요되고, 나머지에 0(n+m)시간이 소요된다.

2. 'check_fail' 루틴은 m번 호출되며, 이 루틴에서는 매번 0 (1)동작(operation)이 수행된다.

3. 'connect' 루틴은 노드가 '소스쪽'이나'싱크쪽'의 어느 한 쪽에만 속하므로 최대n번 호출된다. arc e_{ij}는 'for' 루프에서 많아야 두 번 처리된다. 즉, n_i와 n_j가 이 루틴에서 다루어질 때 처리된다.

4. 따라서 정렬과정을 제외한 feasible cutset 찾기의 complexity는 0 (n+m)이고, 전체의 complexity는 0 (n+m log n)이 된다.

6. Breakable Cutset 찾기

feasible cutset이 구해지면, 이 cutset의 각 arc는 G_x에서의 event-driven graph-solving을 써서 적당히 수평 이동에 의해 끊어진다. 배선의 jogging이 허용된다면 필요에 따라 이동될 수 있다. 어떤 arc가 break되면, G_x에 적당한 제한조건이 더해져서 나중에 요소의 이동이 이전의 breakage를 흐트리지 않게 한다. 이 cutset의 모든 arc들이 끊어지면 이 cutset은 breakable하게 된다.

7. Difficulty Value의 수정

feasible cutset이 완화된 제한 그래프를 써서 구해

진 것이므로, cutset에 break할 수 없는 제한조건이 있으면, 새로이 찾는 cutset이 이러한 arc를 포함하지 않도록하기 위하여 이들 arc의 difficulty value 를 무한대로 증가시킨다.

8. Jogging의 효율적 이용

2차원 컴팩션동안 수평 배선과 수직 배선 모두 면적의 효율적인 이용을 위해 구부려질(jogging) 수 있다. 그림3은 레이아웃의 높이를 줄이기 위해 수직 배선이 구부러진 예를 보여준다. 참고로 기존의 1차원 컴팩터들은 y(x)방향의 컴팩션동안 수평(수직)배선만을 구부렸었다는 점을 밝혀둔다.

9. 전체 알고리듬의 Complexity

CPU-time의 대부분 (80%이상)이 제한 그래프 생성과 그 해를 찾는데 (event-driven 그래프 풀이와 배선 길이 최소화 및 jogging)소비되므로, 알고리듬의 최악의 경우에 대한 complexity는 분석하지 않았다. 한 번의 2차원 컴팩션 과정은 2번의 전역적 그래프 생성과 풀이를 필요로하므로 보통 1차원 컴팩션의 두배의 시간이 소요된다. 2차원 컴팩션을 수행하는 최대 반복 횟수를 지정함으로써 사용자는 예상되는 CPU-time을 조정할 수 있다.

Ⅳ. 계층적 컴팩션

본 논문에서 이용한 전체적인 계층적 방법은[12]의 것과 유사하다. 가장 큰 유일한 차이는 각 leafcell을 컴팩션 하는데 2차원 컴팩션 방법을 이용하였다는 점이다. 이는 위에 설명한 2차원 컴팩션 방법이 모든 port pitch-matching 제한조건을 만족하도록 요소들을 움직이기 때문이다.

계층적 컴팩션은 모든 leafcell들을 컴팩션하는 것으로 시작된다. 1차원 또는 2차원 어느 방법이나 이용할 수 있으며, 여러 컴팩션 단계를 연속적으로 leafcell에 적용할 수 있다. 상위 레벨 컴팩션은 '모듈 그래프(module graph)'[12]를 이용하여 수행하게 된다. 모듈 그래프는 port에 해당하는 노드 N_P와 두 port 간의 거리 제한을 나타내는 arc E_P로 이루어진 요약된 거리 제한 그래프이다. 다음 알고리듬은 계층적 컴팩션 과정의 예를 보여준다.

알고리듬 3. 전형적인 계층적 컴팩션
Algorithm 3. A typical hierarchical compaction.

```
do one-or two-dimensional compaction in x for all leafcells;
generate module graph in x ;
do pitch-matching in x ;
do two-dimensional compaction in y for all leafcells
    maintaining all port constraints in x ;
generate module graph in y;
do pitch-matching in y;
}
```

Ⅴ. 실험 결과

이 절에서는 본 논문에 기술한 2차원 컴팩션 방법이 일반적인 레이아웃에 실용할 수 있음을 보이기 위해 실험한 결과를 설명한다. 모든 결과는 'UNIX system V'를 운영 체제로 갖는 'VAX 8650'에서 얻어진 것이다.

1. 채널(channel)예

그림3(a)는 어떤 채널의 1차원 컴팩션 결과이다. 2차원적인 이동에 의해 contact의 두 수직 배선이 그림3(b)에서 처럼 배열된다. MACS. 1d는 그림3 (a)를 얻는데 1.3초가 소요되었고, MACS. 2d는 그림3(b)를 얻는데 2초가 소요되었다. 이 경우의 면적 감소는 12%이다.

그림4는 다른 채널 예를 보여준다. 그림4(a)는 1차원 컴팩션 결과이다. 2차원 컴팩션 동안 그림 4(b)에 보인 결과에서와 같이 수직 배선이 jog되었다 (구부려졌다). 여기서 면적은 14%가 감소되었으며, MACS. 1d는 그림4(a)를 얻는데 1.8초가, MACS. 2d는 그림 4(b)를 얻는데 3.7초가 소요되었다.

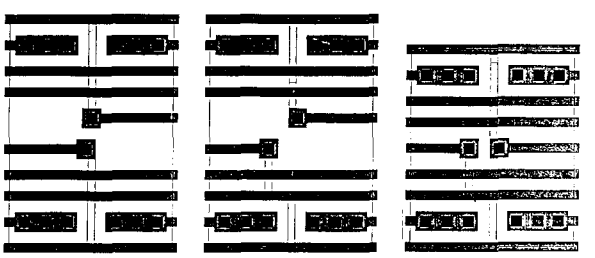
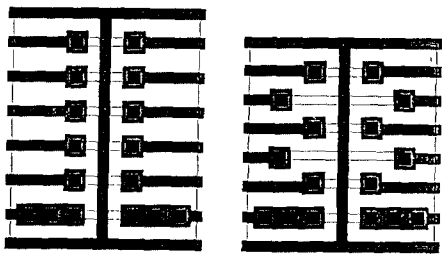


그림 3. 채널의 예 : jogging의 이용
(a) MACS. 1d의 결과
(b) jogging의 이용
(c) MACS. 2d의 결과

Fig. 3. A channel example : use of jogging
(a) result of MACS. 1d and,
(b) being jogged,
(c) result of MACS. 2d.



(a) (b)

그림 4. 채널의 예 : contact의 재배열
(a) MACS. 1d의 결과
(b) MACS. 2d의 결과

Fig. 4. A channel example : contact rearrangements.
(a) result of MACS. 1d and,
(b) result of MACS. 2d.

2. Leafcell 예

표1은 두가지의 leafcell 벤치마크 예제[10]에 대한 결과이다. MACS는 1987년의 international conference on computer design(ICCD)에서 1차원적인 컴팩션의 결과를 발표하였다. MACS. 1d는 1차원 컴팩션 알고리듬을 이용한 MACS의 현재의 결과이다. MACS 컴팩터의 세부적인 구현 방법(jogging과 배선 길이 최소화 등)들이 사용자의 요구에 따라 변하므로 이전의 결과들은 현재의 결과와는 약간 다를 수 있다. 또한 MACS의 메모리 필요량도 매우 감소되었다. MACS. 2d가 본 논문에서 설명한 2차원 방법을 이용한 MACS 컴팩션 프로그램의 결과이다. ICCD에 발표된 결과들은 D. Boyer[10]가 검토하고 정리한 것이다.

MACS. 1d도 다른 결과와 비교하면 좋은 결과를 나타내는데, 이는 jogging과 배선길이 최소화 및 요소의 대각선 거리처리(대각선 거리에 대해서 MACS는 manhattan 거리를 쓰지않고 euclidean 거리를 이용) 등에 대해 효율적인 알고리듬을 이용한 때문이다.

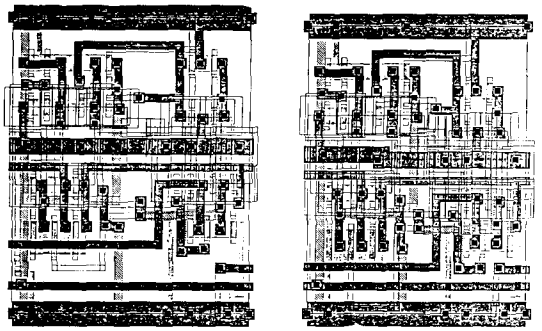
1차원 컴팩션에 비해 MACS. 2d는 5배 정도의 CPU time 소요에 의해 10%까지의 면적 감소를 얻는다. 표에서의 면적은 'PWELL'의 면적까지도 포함한 것이다.

그림5에서의 afa cell에 대한 MACS. 1d와 MACS. 2d의 결과는 1차원 컴팩션에 비교하여 2차원 컴팩션이 효과적임을 보여준다. 그림에서 원으로 표시한 부분은 1차원과 2차원 컴팩션 결과에 있어서 중요한

표 1. 레이아웃 컴팩션의 결과
Table 1. Results of the layout compaction.

Example : afa						
compactor (sequence)	width	height	area	(area %)	mem. (kbyte)	CPU (sec)
MACS	143	166	23738	(107.3 %)	1450	9
SPARCS	157	180	28260	(127.8%)	356	11
Symbolics	160	189	30240	(136.7 %)	164	5
Zorro	140.5	171	24025.5	(108.6 %)	647	430
MACS. 1d (xjyj)	144	174	25056	(113.3 %)	524	9
MACS. 2d (xjyj2)	141.75	169.5	24026.6	(108.6 %)	655	20
MACS. 1d (xjyjxj)	141	174	24534	(110.9 %)	524	11
MACS. 2d (xjyjxj2)	130.5	169.5	22119.8	(100.0 %)	655	34

Example afakr						
compaction sequence	width	heigth	area	(area %)	mem. (kbyte)	CPU (sec)
MACS	142	145	20590	(109.6 %)	1390	5
SPARCS	157	151	23707	(126.2 %)	372	8
Symbolics	154	154	23716	(126.3 %)	160	5
Zorro	128.5	151	19403.5	(103.3 %)	598	524
MACS. 1d (xjyj)	138.75	152.25	21125	(112.5 %)	524	9
MACS. 2d (xjyj2)	137.75	143.25	19876	(105.8 %)	983	50
MACS. 1d (yjxj)	135	146.25	19743.8	(105.1 %)	524	9
MACS. 2d (yjxj2)	129.75	144.75	18781.3	(100.0 %)	918	48



(a) (b)

그림 5. leafcell의 예 : afa
(a) MACS. 1d의 결과
(b) MACS. 2d의 결과

Fig. 5. A leafcell example : afa
(a) result of MACS. 1d and,
(b) result of MACS. 2d.

차이를 보여준다.

이 예에서 MACS. 2d는 최소의 레이아웃을 생성하였다. 컴팩션 순서는 사용자에 의해 주어지며, 컴팩션 결과는 이용된 컴팩션 순서에 따라 달라진다.

3. 계층적 컴팩션의 예

표 2는 곱셈기의 예[10]에 대한 결과이다. 이 예는 6종류의 부분 cell을 가지고 있다. 계층구조로부터의 추가되는 제한조건을 이용하지 않고 컴팩션을 하게 되면, 모든 부분 cell (mul4×4의 경우 16개, mum16×16의 경우 256개)들이 컴팩션 후에 다른 모양을 가질 수 있게된다. 같은 모양을 갖도록 하기 위하여 같은 종류의 부분 cell임을 지정하면, 컴팩션 후에 6가지종류의 부분 cell만을 갖게된다. 이렇게 함으로써 컴팩션하는 동안 설계 데이타가 갑자기 증가하는 것을 방지할 수 있다.

특히 mul 16×16의 경우, MACS는 symbolics 컴팩터보다 적은 CPU 시간과 적은 메모리 양으로 더 작은 면적의 레이아웃을 생성해냈다.

Ⅵ. 결 론

새로운 2차원 컴팩션 방법이 개발되었으며, MACS에 구현되었다. 이 방법은 임계 경로를 반복적으로 break함으로써 밀집된 레이아웃을 생성한 수 있다. break될 제한조건들은 이들을 break할 때의 예상되는 어려운 정도(difficulty value)를 이용하여 선택된다. 예상되는 difficulty 값은 다른 방향에서의 제한조건을 만족시킬 수 있는지의 여부에 따라 동적으로 수정되므로 이 방법은 융통성이 있다. 실험 결과는 이 방법이 만족스러운 결과를 효율적으로 생성한다는 것을 보여준다.

본 방법에서는 컴팩션동안 양쪽 방향에서의 제한조건이 고려되므로, 컴팩션동안 같은 cell내의 요소들이 같은 모양으로 유지되도록 하는 계층적 컴팩션이 가능하다.

參 考 文 獻

[1] M.Y. Hsueh, "Symbolic Layout and Compaction," *Y.C. Berkeley, UCB/ERL Report M79/80,* 1979.

표 2. 레이아웃 컴팩션의 결과 : 곱셈기
Table 2. Results of the layout compaction : multipliers.

Example : mul4×4					
compaction sequence	width	heighi	area (area %)	mem. (kbyte)	CPU (sec)
SPARCS	649	601	390049 (126.5 %)	754	66
Symbolics	654	638	417252 (135.3 %)	840	54
Zorro	577	577.8	333217.5 (108.1 %)	741	1904
MACS. 1d (yjxj)	570	566.25	322763 (104.7 %)	1180	32
MACS. 2d (yjxj2)	544.5	566.25	308323 (100.0 %)	1573	96

Example : mul16×16					
compaction sequence	width	height	area (area %)	mem. (kbyte)	CPU (sec)
Symbolics	2524	2780	7016720 (132.3 %)	14400	1073
MACS. 1d (yjxj)	2271	2492.25	5659900 (106.7 %)	4784	347
MACS. 2d (yjxj2)	2128.5	2492.25	5304754 (100.0 %)	4915	412

[2] Y.E. Cho, "A Subjective Review of Compaction," *Proceedings of 22nd Design Automation Conference,* pp. 396-404, June 1985.

[3] D. Tan and N. Weste, "Virtual Grid Symbolic Layout 1987," *International Conference on Computer Design,* pp. 192-196, October 1987.

[4] J. Bums and A. Newton, "Efficient Constraint Generation for Hierarchical Compaction," *International Conference on Computer Design,* pp. 197-200, October 1987.

[5] S. Sastry and A. Parker, "The Complexity of Two-Dimensional Compaction of VLSI Layouts," *International Conference on Circuits and Computers,* pp. 402-406, 1982.

[6] M. Schlag, Y.Z. Liao, and C.K. Wong, "An Algorithm for Optimal Two-Dimensional Compaction of VLSI Layouts," *Integration,* pp. 179-209, 1983.

[7] G. Kedem and H. Watanabe, "graph-Optimization Techniques for IC Layout and Compaction," *IEEE Transactions on CAD of ICAS* vol. 3, no. 1, January 1984.

[8] W. Wolf, R. Mathews, J. Newkirk, and R. Dutton, "Two-Dimensional Compaction Strategies," Proceedings of Int. Conf. on CAD, pp. 90-91, 1983.

[9] H. Shin, A. Sangiovanni-Vincentelli, and C. Sequin, "Two-Dimensional Compaction by Zone -Refining," *Proceedings of 23rd Design Automation Conference,* pp. 115-122, June 1986.

[10] D. Boyer, "Symoblic Layout Compaction Benchmarks," *International Conference on Computer Design,* pp. 186-191, 209-217, October 1987.

[11] R.C. Mosteller, A. Frey and R. Suaya, "2D Compaction-A Monte Carlo Method," *Proceedings, the 1987 Stanford Conference,* pp. 173-197, 1987.

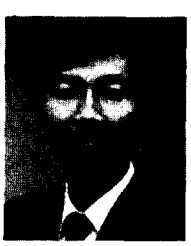
[12] W. Crocker, R. Varadarjan, and C. Lo, "MACS: A Module Assembly and Compaction System," International Conference on Computer Design, pp. 205-208, October, 1987.

[13] H. Shin and C. Lo, "An Efficient Two-Dimensional Layout Compaction Algorithm," *Proceedings of 26th Design Automation Conference,* pp. 290-295, June, 1989.

[14] R.E. Tarjan, "Data Structures and Network Algorithms," CBMS 44, SIAM, pp. 97-111, 1983.

[15] Y. Liao and C.K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *IEEE Transactions on CAD of ICAS* vol. 2, no. 2, April 1983.

著 者 紹 介



申 鉉 哲(正會員)

1955年 9月 12日生. 1978年 2月 서울대학교 전자공학과 졸업. 1980年 2月 한국과학 기술원 전기 및 전자공학과 공학석사. 1987年 미국 U.C Berkeley전기 및 컴퓨터 공학과 공학박사. 1980年～1983年 금오공과대학 전자공학과 전임강사 조교수. 1987年～1989年 미국 AT & T Bell Laboratories, Murray Hill, 연구원. 1989年 9月～현재 한양대학교 전자공학과 조교수. 주관심분야는 집적회로 설계 자동화, 디지탈 신호처리 시스템설계 등임.