VLSI DESIGN
2000, Vol. 11, No. 3, pp. 301–310
Reprints available directly from the publisher
Photocopying permitted by license only

A New 2-way Multi-level Partitioning Algorithm*

YOUSSEF SAAB†

Computer Engineering and Computer Science Department, University of Missouri-Columbia, Engineering Building West, Columbia, MO 65211

(Received 1 March 1999; In final form 10 February 2000)

Partitioning is a fundamental problem in the design of VLSI circuits. In recent years, the multi-level partitioning approach has been used with success by a number of researchers. This paper describes a new multi-level partitioning algorithm (PART) that combines a blend of iterative improvement and clustering, biasing of node gains, and local uphill climbs. PART is competitive with recent state-of-the-art partitioning algorithms. PART was able to find new lower cuts for many benchmark circuits. Under suitably mild assumptions, PART also runs in linear time.

Keywords: Hypergraph partitioning; Multi-level; Clustering; Contraction

1. INTRODUCTION

Partitioning problems have been studied by many researchers, and many algorithms have also been reported. Some of these algorithms use general techniques for combinatorial optimization such as Simulated Annealing, Mean Field Annealing, Tabu Search, genetic algorithms, and Stochastic Evolution. Other algorithms are specifically designed for the particular problem being solved, methods developed specifically for partitioning include Group migration, Clustering, and Spectral methods. For a review of partitioning methods, readers are referred to [1, 2]. This paper describes a new multi-level partitioning algorithm (PART) that combines several recent concepts in an efficient and effective way.

2. THE NETWORK BISECTION PROBLEM

A hypergraph $G(V, E)$ consists of a set of nodes V and a set of nets E. Each net $e \in E$ is a subset of 2 or more nodes in V. A node i has an integer size $S(i)$. However, all nets have unit weights, for a net of weight k can be replaced by k nets of unit weight each. An electrical circuit can be modeled as a hypergraph. A graph is a special case of a hypergraph in which each net links exactly two nodes.

* This work was supported by a 1999 Research Board grant, University of Missouri.
† e-mail: saaby@missouri.edu

A partition of a hypergraph $G(V, E)$ is an unordered pair (V_1, V_2) of subsets of V such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. The size $S(A)$ of a subset of nodes $A \subseteq V$ is the sum of the sizes of its constituent nodes. A partition (V_1, V_2) is a bisection if $|S(V_1) - S(V_2)|$ is as small as possible. A net e is said to be cut by a partition (V_1, V_2) if it links nodes in V_1 and in V_2, *i.e.*, $e \cap V_1 \neq \emptyset$ and $e \cap V_2 \neq \emptyset$. The cut (cost) of a partition (V_1, V_2) is the subset (number) of nets cut and is denoted by $cut(V_1, V_2)$ ($cost(V_1, V_2)$).

The network bisection problem (NB) seeks a bisection of minimum cost. Note that if the nets are ignored, an algorithm for NB can be used to solve the set partition problem which is NP-Complete [3]. Therefore, unless all nodes have unit sizes, it is NP-Hard to find just a bisection of arbitrary cost of a hypergraph. As a consequence, NB is usually relaxed to find a partition (V_1, V_2) such that $|S(V_1) - S(V_2)| \leq \alpha S(V)$, where α is some constant. This relaxed version of NB is also NP-Hard and is known as the α-edge separator problem [4].

3. NODE CONTRACTION

Contraction has been used by many authors to enhance the performance of iterative partitioning algorithms. It is an operation in which subsets of nodes may be clustered (contracted or compacted or coalesced) to form a single node each. When a subset of nodes X is clustered to form a single node x, the nets incident with node x in the new hypergraph are of the form $(e \cap (V - X)) \cup \{x\}$, where e is a net originally incident with some node in X, and with the provision that nets that are reduced to one node are discarded. All other nets and nodes in $V - X$ remain the same. Naturally, the size of a coalesced node is the sum of the sizes of its constituent nodes.

4. MULTI-LEVEL PARTITIONING

Under the multi-level partitioning paradigm [5, 6], An input hypergraph G is successively coarsened. This process, called the coarsening phase, produces a sequence of hypergraphs $G = G_0 \rightarrow G_1 \rightarrow \cdots \rightarrow G_{l-1} \rightarrow G_l$, where G_i is obtained by contraction of several disjoint subsets of nodes in G_{i-1} for $1 \leq i \leq l$. This is followed by the un-coarsening phase which produces a sequence of partitions $P_l \rightarrow P_{l-1} \rightarrow \cdots \rightarrow P_1 \rightarrow P_0$, where $P_i = (V_{i1}, V_{i2})$ is a partition of G_i for $0 \leq i \leq l$. Partition P_l of the coarsest G_l is computed by a direct partitioning method. Then, for $l > i \geq 0$, partition P_{i+1} is projected to a partition of G_i which is then refined to P_i by an iterative partitioning algorithm.

PART uses the multi-level paradigm. However, it differs from other multi-level algorithms in its coarsening phase which produces a sequence $(G_0, P_0) \rightarrow (G_1, P_1) \rightarrow \cdots \rightarrow (G_{l-1}, P_{l-1}) \rightarrow (G_l, P_l)$, where $G = G_0$ is the input graph, and $P_i = (V_{i1}, V_{i2})$ is a partition of G_i for $0 \leq i \leq l$. For $1 \leq i \leq l$, the pair (G_i, P_i) is obtained from (G_{i-1}, P_{i-1}) as follows. An iterative improvement algorithm is applied to refine partition P_{i-1} which along the way computes disjoint subsets of nodes for contraction such that nodes to be contracted belong to the same side of the refined partition P_{i-1}. Therefore, after contraction of G_{i-1} into G_i, partition P_{i-1} projects to a partition P_i of G_i. To get the process started, the first partition P_0 of G_0 is generated randomly. The un-coarsening phase in PART proceeds just like other multi-level algorithms, except that the partition P_l of G_l is immediately available after coarsening. Also some refinement steps are skipped during the un-coarsening phase as we will explain later.

The advantages of the coarsening scheme used by PART are:

- The best partition is refined during coarsening, so that partition P_l of the coarsest graph projects to a good partition of $G = G_0$.
- Coarsening is a by-product of iterative improvement at almost no additional computation. This helps save computation time.
- Multiple improvement phases are possible: Coarsening and un-coarsening can be repeat-

edly applied until no further improvements can be made.
- Most of the improvement in the number of nets cut is achieved during the coarsening phase. The un-coarsening phase contributes minor improvement. In fact as we will explain next, many refinement steps during un-coarsening are skipped altogether.

During the un-coarsening phase, A refinement step is skipped if it is unlikely to produce any improvement. The coarsening phase guarantees that $cost(P_{i+1}) \leq cost(P_i)$. Therefore during the un-coarsening phase, if $cost(P_{i+1}) = cost(P_i)$ then refinement is skipped. Otherwise, P_{i+1} projects to a partition of G_i that will be refined and then will replace P_i. We note that most of the refinement steps are skipped during the un-coarsening phase of PART.

The reason for skipping refinement during the un-coarsening phase is simple. During the coarsening phase, P_{i+1} was obtained from P_i using an iterative improvement algorithm. If during the un-coarsening phase we see that $cost(P_{i+1}) = cost(P_i)$, then the iterative improvement algorithm was unable to improve P_i during coarsening. This also means that it is likely that P_{i+1} has not changed since P_i was projected to P_{i+1} during the coarsening phase. This means that P_i and the projection of P_{i+1} on G_i are essentially the same partitions of G_i. This means that the iterative algorithm is unlikely to improve this partition now, and therefore the refinement step is skipped.

We use the words likely and unlikely in our discussion because in the iterative improvement algorithm we use:

- There is a minor random step which make it likely that two different runs using the same initial partition may produce different results. However the probability of such behavior is very small from what we observed from our experiments.
- The current partition replaces the best partition if it has a smaller cost or if it has the same cost but has a better size balance.

5. ITERATIVE IMPROVEMENT AND CONTRACTION

The core of PART is an iterative improvement pass that refines the current partition, and at the same time colors some of the nodes. The colors are used to contract the graph during the coarsening phase and are not used at all during the uncoarsening phase. The iterative improvement pass is a variant of the Fiduccia–Mattheyses [7] algorithm and is essentially a slightly modified version of the improvement pass used in [8].

Given an initial bisection, *i.e.*, a partition that satisfy the size balance criterion, sequences of nodes are moved from one side of the partition to the other side until every node has moved once. Several subsets of nodes in the same sequence may be colored with the same color signaling that these nodes should be clustered together. The rational for this clustering scheme is simple. Nodes that are heavily connected to each other tend to migrate together during iterative improvement. The best bisection seen during the iterative improvement pass is saved if it improves on the best bisection so far. A bisection (V_1, V_2) improves on a bisection (X_1, X_2) if $cost(V_1, V_2) < cost(X_1, X_2)$ or if $cost(V_1, V_2) = cost(X_1, X_2)$ and $|S(V_1) - S(V_2)| < |S(X_1) - S(X_2)|$.

The order in which nodes are moved from one side of the partition to the other is determined by the gain of nodes. The gain of a node is the net decrease (may be negative) in the number of nets cuts if the node is moved to the other side of the partition. The node with highest gain is moved first. To facilitate retrieval of the node of highest gain, a bucket structure as described in [7] is used with the LIFO scheme as described in [9]. To get started, the first initial bisection is randomly generated.

During the iterative pass, a node can be:

- free or locked. Only free nodes are allowed to move, a node is locked as soon as it moves, initially all nodes are free to move.
- colored or not. Initially all nodes are not colored. A node is not colored until after it is moved.

- contractable or not. A node is contractable if it is available for clustering with other nodes. Initially all nodes are not contractable. A node becomes immediately contractable after it is moved. A node ceases to be contractable for the rest of the iterative pass as soon as it is colored or as soon as its own sequence terminates. Therefore, nodes to be clustered together always belong to the same sequence. A new bisection replaces the best one so far only at the end of a complete move sequence. This is so in order to prevent nodes on different sides of the best bisection from being clustered together. Therefore at the end of the iterative improvement pass, nodes of the same color are on the same side of the best bisection. This means that after clustering, the best bisection projects to an initial bisection of the coarser graph during the coarsening phase. The color and the contractability states of a node are only used during the coarsening phase and are ignored during the uncoarsening phase.

Coloring of the nodes during the iterative pass is done as follows. Initially all nodes are not colored. A soon as a node v is moved, it is made contractable. Then v and all other contractable nodes that are connected to it *via* critical nets, are colored by a new color and their contractability is removed. If it turns out that only v gets the new color, then the color of v is removed and the contractability of v is restored. A net is critical for node v if it is removed from the cut after moving v to the other side of the partition. At the end of the iterative improvement pass, nodes of the same color are clustered together. Other nodes connected to v *via* non-critical nets are not selected for clustering with v for two reasons:

- A non-critical net may be very long. Therefore large clusters may be formed if neighboring node *via* non-critical nets are included.
- For the purpose of efficiency. If all the contractable nodes connected to node v are searched for, many nodes may be examined too many times. For example, consider the situation in which a single net connects all the nodes of the hypergraph. In this situation during the search for contractable nodes connected to v, all the nodes are examined. Since each node is moved once during the iterative improvement pass, the running time of the algorithm becomes quadratic. By restricting the search to contractable nodes connected to v *via* critical nets this situation is avoided. A net can become critical during the iterative shifting of nodes from one side of the partition to the other if at some points it becomes cut and its last node is eventually moved to the other side. Because a node is locked immediately after it is moved, a net cannot be critical more than once during the iterative improvement pass.
- Critical nets are scanned anyway in order to update node gains. This means that node coloring can be done at the same time at almost no additional computation cost.

The function $MOVE(v, A, B)$ performs the update needed to move node v from side A to side B of the current partition. Basically, $MOVE(v, A, B)$ moves v from A to B, sets $free(v) = false$, updates gains of all affected nodes, and, while doing gain updates, it colors v and all its contractable neighbors *via* critical nets with a new color and set $contractable(x) = false$ for all nodes that got the new color. If only v gets the new color then it sets $color(v) = NoColor$ and $contractable(v) = true$.

We are now ready to present the complete the description of the iterative improvement pass with contraction:

1. For each node v, set $free(v) = true$, $color(v) = NoColor$, and $contractable(v) = false$.
2. Save the initial input bisection (V_1, V_2) as the best bisection so far (B_1, B_2).
3. Partition Side Selection: for $1 \leq i \leq 2$, let h_i be the highest gain of a free node in V_i. If V_i has no free nodes then set $h_i = -\infty$. If $h_1 = h_2$ then toss a balanced coin and set $j = 1$ or $j = 2$ depending on the outcome of the toss. Otherwise set j as the index of the side with the highest gain.

4. Forward move: If V_j has no free nodes then go to Step 7. Otherwise let v be a free node of highest gain in V_j. Record cost of current partition in *OldCost*. Call $Move(v, V_j, V_{(3-j)})$. If $cost(V_1, V_2) \geq OldCost$ then repeat this step. Otherwise, for each contractable node x, set $contractable(x) = false$.
5. Restore size: Let j be the index of the largest side of the partition. As long as $S(V_j) - S(V_{(3-j)}) > \alpha S(V)$ and V_j has free nodes do: let v be a free node of highest gain in V_j. If $|S(V_j - v) - S(V_{(3-j)} \cup v)| < |S(V_j) - S(V_{(3-j)})|$ then call $Move(v, V_j, V_{(3-j)})$. Otherwise for each contractable node x, set $contractable(x) = false$.
6. Save: If $cost(V_1, V_2) < cost(B_1, B_2)$ or if $cost(V_1, V_2) = (B_1, B_2)$ and $|S(V_1) - S(V_2)| < |S(B_1) - S(B_2)|$, then replace (B_1, B_2) with (V_1, V_2). Go to Step 3.
7. Restore best bisection: replace (V_1, V_2) with (B_1, B_2).
8. Contraction: Color each remaining uncolored node with a new color. Cluster together all node with the same color.

The above pass takes an input graph G_i and a initial bisection $P_i = (V_{i1}, V_{i2})$ of G_i. It then improves P_i and contract G_i. It returns the contracted graph $G_{(i+1)}$ and an initial bisection $P_{(i+1)}$ of $G_{(i+1)}$ which is the projection of the best bisection P_i of G_i. For later reference, we will encapsulate this iterative improvement pass in the function $REFINE\text{-}AND\text{-}CONTRACT(G_i, P_i)$. This function is the main function used during the coarsening phase. The same function with the contraction operation removed is used in the uncoarsening phase. We will call this function in future reference $REFINE(G_i, P_i)$. During the uncoarsening phase the function $MOVE(v, A, B)$ can be modified to ignore coloring of nodes although this does not significantly affect the running time.

6. IMPROVEMENT MODULE

A single multi-level pass consists of a coarsening phase followed by an uncoarsening phase. We will encapsulate this pass in the function $IMPROVE(G, P)$ which takes as input a hypergraph G and a partition $P = (V_1, V_2)$ of G, and returns an improved partition of G. A pseudo-code of this function is as follows:

```
IMPROVE(G, P) {
(G0, P0) = (G, P);
i = 0;
while (true) do {/*coarsening phase */
      (Gi, Pi) = REFINE-AND-CONTRACT
      (G(i-1), P(i-1));
      if (Gi = G(i-1)) then /*No contraction*/
        break;
      else
        i = i + 1;
      }
i = i - 1;
while (i ≥ 0) do {
       if (cost(Pi) > cost(P(i+1))) then {
          replace Pi by the projection of P(i+1)
          on Gi;
          Pi = REFINE(G(i+1), Pi);
          }
       i = i - 1;
       }
return P0;
}
```

One approach to optimize an initial bisection is to call function *IMPROVE* repeatedly until no more improvements can be made. We experimentally found that this approach can find good bisections. However, we were able to obtain better results by applying node gain biasing and local bisection perturbations inspired by the work in [10].

7. NODE BIASING

Paper [10] points out that during a pass of a Fiduccia–Mattheyses [7], a net e can be in one of three states:

- Free: all nodes of net e are free.
- Loose: only one side of the partition contains locked nodes of net e. The side that contains

locked nodes of net e is called the anchor of e. The other side is called the tail of e.

- Locked: Both sides of the partition contain locked nodes of net e.

Furthermore, the state of a net changes from free to loose to locked in that order. Clearly locked nets cannot be removed from the cut for the remainder of the iterative pass. Therefore, it makes sense to increase the chances of removal of loose nets from the cut before they become locked. This is achieved by positively biasing the gains of nodes of a loose net that are in the tail of the loose net. The author of [10] use a formula for the additional bias that favors short nets, dense connectivity at the anchor, and sparse connectivity at the tail. They apply the bias every time a node of a loose net moves, and they do not remove the bias once the net is locked. Our approach to biasing is simpler. A bias of 1 is added to all nodes in the tail of a net e only immediately after the state of e changes from free to loose. Also, we remove the additional bias as soon as the loose net becomes locked.

We have experimentally observed that biasing is not always good. After all some of the loose nets may actually belong to the optimal cut. Therefore, biasing them to be removed from the cut may hinder the algorithm from finding the optimal solution. Another case where biasing is not good is when the partition being improved is close to being optimal. In this case, biasing prevent the distinction between a move that actually improves the cut (positive gain without bias) from another that does not (equal positive gain but biased). Our experiments suggest that biasing is most beneficial initially when the partition is far from optimal, and that it should be turned off once the partition is close to optimality. Furthermore sometimes biasing altogether should be avoided. This lead us to the notion of selective biasing.

In selective biasing, only a selected subset of nets can be biased. If we pose the question: which nets are harder to remove from the cut? The intuitive answer is longer nets are harder to remove from the cut than short nets. One way to increase the chances of removal of long nets from the cut is to simply apply biasing to loose nets longer than some threshold length. We found that biasing long nets was beneficial for some circuits, but it was quite bad for others. For the other circuits, doing the opposite, *i.e.*, biasing short nets, was very effective.

The conclusions that can be drawn from our preliminary experiments is that depending of the structure and connectivity of the input circuit one of the following may be beneficial:

- bias of all loose nets regardless of length ($mode = 1$).
- avoid biasing altogether ($mode = 2$).
- bias loose nets longer than some threshold length ($mode = 3$).
- bias loose nets shorter than some threshold length ($mode = 4$).

Since it is difficult to determine which of the four items above is helpful, the obvious choice is to try all of them and pick the best result. Indeed, this is exactly what we did. We ran $IMPROVE(G, P)$ four times each time with a random initial partition and one of the four biasing approach above. Furthermore, we only applied the biasing during the coarsening phase of $IMPROVE$ and turned it off during the un-coarsening phase. To apply biasing, the following sample code need to be added to the end of function $MOVE(v, A, B)$:

```
for (each net e incident to node v) do
if (net e is selected for biasing) then
   if (v is the only locked node of e) then /*free to
   loose transition*/
      for (each node w of e ∩ A) do
            gain(w) = gain(w) + 1;
   else if (e has locked nodes in A and v is the
   first locked node of e in B) then
            /*loose to locked transition*/
                  for (each node w of e ∩ B) do
                        gain(w) = gain(w) − 1;
```

We call our approach global sampling since we sample four different initial random partition to

obtain a bisection that will be subject to further refinement. We used a threshold of 5 for selective biasing. Obviously other thresholds can be used. We found that a threshold of 5 works reasonably well on all the circuits which we experimented with. This global sampling is summarized in the pseudo-code:

```
generate a random initial bisection P;
Q = P; /*save best bisection so far*/
for 1 ≤ i ≤ 4 do {
   call IMPROVE(G, P) using mode i;
   if (cost(P) < cost(Q)) then
      Q = P; /*save best bisection so far*/
   generate a random initial bisection P;
   }
return Q;
```

The bisection Q returned by the above code will be further optimized as we will explain next.

8. LOCAL SAMPLING

The partition returned by global sampling tend to be of high quality. However, it can be further improved by calling function *IMPROVE* repeatedly until no more improvements can be made. This approach, although satisfactory, can result in too many iterations with only small improvements in cost per iteration. We avoid too many iterations by preceding this step with an optimization step which we call local sampling. As we shall see, local sampling may not improve the partition at all. However in many instances local sampling improve the partition enough to significantly reduce the number of times function *IMPROVE* is called in the last refinement step of the algorithm.

Like global sampling, local sampling also perturbs the current partition then it attempts to improve it. However unlike global sampling which starts each time with a brand new random initial bisection, local sampling attempts to perturb only the nodes of nets that belong to the cut. This local hill-climbing does not increase the cost of the perturbed partition significantly. Yet it allows the removal from the cut of certain nets that are harder to remove otherwise. Our local sampling is similar to the so called stable net removal in [10].

We encapsulate the local perturbation in the function *PERTURB*(P). This function first sets the states of all nodes of cut nets free. It then scans the cut nets of bisection P in random order. For each scanned net e the following is done. If all the nodes of e on the larger side of the partition are free, then they are moved to the other side of the partition and they are locked. Otherwise if all the nodes of e on the smaller side of the partition are free, then they are moved to the other side of the partition and they are locked. Nothing is done if net e has locked nodes on both sides of the partition.

After scanning all the cut nets of the initial bisection P, the perturbed partition P may no longer be a bisection. Partition P is then restored to a bisection as follows. The nodes are scanned one at a time and the scanned node is moved to the other side of the partition if: (1) moving it improves the balance of the partition, or (2) if (1) is false and the node currently belongs to the larger side of the partition and a random coin toss yields a head. The nodes are repeatedly scanned in this fashion until the balance criterion of the bisection is restored. Although it may seem that this approach is expensive, we have observed that it does not take much time for the 45–55% balance criterion (10% deviation in size) that is typically used in the literature.

Local sampling can now be described by the following pseudo-code:

```
Q = best bisection so far;
repeat 4 times:
        P = PERTURB(Q);
        IMPROVE(G, P);
        if (cost(P) < cost(Q)) then
           Q = P; /*save best bisection so far*/
return Q;
```

We iterate 4 times in local sampling just like we did for global sampling. Off course, any other number of iterations can be used. Local sampling

can only improve its input bisection. If no improvements can be made the initial bisection is returned intact.

The full algorithm PART can now be described as follows:

1. Q = bisection returned by global sampling.
2. Let P be the best partition obtained by applying local sampling to Q.
3. Repeatedly call $IMPROVE(G, P)$ until no further improvement can be made.

In algorithm PART, node biasing is applied only during the coarsening phases of global sampling as explained before.

9. EXPERIMENTAL RESULTS

In Tables I and II we compare the result of PART to previously reported results in the literature. All the results are reported for 45–55% partitions (deviation up to 10% of exact bisection). For our algorithm PART, we report the average running time of a single run in seconds, the minimum cut found in 10 runs (cut), and the number of solutions in 10 runs that are within 3% (fr3) and within 5% (fr5) of the smallest cut. All experiments were performed on a DEC 8400 station.

Table I presents the results on 23 benchmarks from the CAD Benchmarking Laboratory (see [6] for details). The results of three other algorithms are reported as percentages in cut over the cut produced by PART. The three other algorithms are MLC(100) [6], CLIP with PROP gain calculations [11] and hMetis [5]. For CLIP, we used the results reported in [6].

Table II presents the results on the ISPD98 benchmarks [12]. The results of three other algorithms are reported as percentages in cut over the cut produced by PART. The three other algorithms are FM [7], CLIP [11] and hMetis [5]. For FM and CLIP, we used the results reported in [12] which are the minimum cuts over hundred runs of each algorithm. The results of hMetis were erroneous in [12]. The corrected results which we used are available from vlsicad.cs.ucla.edu web site.

TABLE I Results on 23 standard benchmarks

Test	# Nodes	Time(s)	cut	fr3	fr5	MLC	CLIP	hMetis
balu	801	0.4	27	9	9	0.0	0.0	0.0
prim1	833	0.4	47	3	3	0.0	8.5	6.4
bm1	882	0.4	47	6	5	0.0	0.0	8.5
test04	1515	0.8	48	4	10	0.0	8.3	6.2
test03	1607	0.8	56	4	8	0.0	1.8	3.6
test02	1663	0.9	86	5	8	3.5	1.2	2.3
test06	1752	1.0	60	3	8	0.0	0.0	0.0
struct	1952	0.7	33	4	10	0.0	0.0	0.0
test05	2595	1.5	71	5	7	0.0	8.5	0.0
l9ks	2844	1.7	104	8	9	1.9	0.0	1.9
prim2	3014	2.3	139	7	8	0.0	9.4	4.3
s9234	5866	2.0	40	9	9	0.0	5.0	0.0
biomed	6514	3.2	83	10	10	0.0	1.2	0.0
s13207	8772	2.8	52	3	3	5.8	36.5	5.8
s15850	10470	3.4	41	5	5	7.3	36.6	2.4
ind2	12637	10.9	160	2	5	2.5	20.0	4.4
ind3	15406	19.3	241	2	2	0.8	0.8	5.4
s35932	18148	6.6	40	2	3	2.5	5.0	5.0
s38584	20995	10.1	47	8	10	0.0	8.5	0.0
avqsm	21918	13.6	127	4	4	0.8	13.4	2.4
s38417	23949	8.9	49	7	10	0.0	32.7	4.1
avqlrg	25178	14.6	127	3	3	0.8	12.6	0.0
golem3	103048	178.5	1331	6	7	1.1	−100.0	8.6

TABLE II Results on ISPD98 benchmarks

Test	# Nodes	Time(s)	cut	fr3	fr5	FM	CLIP	hMetis
ibm01	12752	14.3	180	9	9	6.1	0.6	0.0
ibm02	19601	31.4	262	9	9	1.5	1.1	0.0
ibm03	23136	39.7	950	10	10	21.1	12.4	0.6
ibm04	27507	55.9	522	2	3	15.5	7.9	3.8
ibm05	29347	68.2	1672	6	10	12.1	28.3	2.6
ibm06	32498	67.4	885	7	8	9.9	10.4	0.3
ibm07	45926	108.6	824	6	6	25.8	12.7	3.5
ibm08	51309	115.6	1140	10	10	12.7	10.6	0.2
ibm09	53395	120.4	620	10	10	47.1	8.7	0.6
ibm10	69429	212.0	1249	3	3	19.3	13.7	0.6
ibm11	70558	200.7	956	2	7	52.6	11.2	0.4
ibm12	71076	265.7	1872	5	7	20.5	27.5	2.5
ibm13	84199	267.3	831	7	7	42.1	9.9	1.1
ibm14	147605	499.1	1794	7	8	65.2	41.4	2.4
ibm15	161570	981.7	2587	1	1	97.4	38.0	1.5
ibm16	183484	953.6	1710	6	7	38.2	54.3	2.6
ibm17	185495	158.9	2186	5	7	39.6	28.2	2.4
ibm18	210613	876.3	1521	6	6	12.2	49.1	1.3

In all cases PART was able to find smaller cuts. On circuit s13207 for example, PART improves the best previously reported cut by almost 6%. This is significant given that such a relatively small circuit has been used by many. This is also true for all the other circuits in Table I. The fact that the results of many state-of-the-art partitioners began to converge on the circuits in Table I, motivated the introduction of the ISPD98 benchmarks [12].

PART performed also well on the ISPD98 benchmarks, and in all cases obtained the smallest cut. As can be readily seen from Table II, PART significantly improves over the results of FM and CLIP. The improvement over the results of hMetis which is a very powerful partitioner are modest. For example on circuit ibm04, the result of hMetis is 3.8% larger than the result of PART.

Algorithm PART is robust. In many cases 7 out of 10 runs are within 5% and even 3% of the minimum as columns fr3 and fr5 in Tables I and II show. This means that only few runs are needed to get a good solution.

The running time of PART is reasonable. On the largest circuit, ibm18, with 210613 nodes and 201920 nets, a single run of PART takes about 15 minutes. Note that a single run of PART requires at least 9 coarsening/un-coarsening runs or 9 calls to function *IMPROVE*: 4 calls in global sampling, 4 calls in local sampling, and at least one call in the final improvement phase. Therefore, a single call to function *IMPROVE* takes less than 2 minutes for circuit ibm18.

10. CONCLUSION

We have presented an efficient, effective, and robust partitioning algorithm. Our algorithm effectively combines node gain biasing and local uphill climbs within the multi-level framework. For several circuits new lower cuts have been obtained than previously reported in the literature. In the future, we intend to investigate possible improvement to our algorithm which may include new ways of clustering and node biasing. We think that the running time can be improved without significant degradation in quality by limiting the number of levels during the coarsening phase (currently we do not use any preset limit), and by using the early-exit strategy in the un-coarsening phase as described in [5].

References

[1] Alpert, C. and Kahng, A. (1995). "Recent directions in netlist partitioning: A survey", *Integration – The VLSI Journal*, **19**(1–2), 1–81.

[2] Johannes, F. M., "Partitioning of VLSI circuits and systems", *Design Automation Conference*, 1996, pp. 83–87.

[3] Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-completeness.* New York, NY: W. H. Freeman and Company, 1979.

[4] Jones, C., Vertex and Edge Partitions of Graphs. *Ph.D. Thesis*, Department of Computer Science, Pennsylvania State University, May, 1992.

[5] Karypis, G., Aggarwal, R., Kumar, V. and Shekhar, S., "Multilevel hypergraph partitioning: Application in VLSI domain", *Design Automation Conference*, 1997, pp. 526–529.

[6] Alpert, C. J., Huang, J.-H. and Kahng, A. B., "Multilevel circuit partitioning", *Design Automation Conference*, 1997, pp. 530–533.

[7] Fiduccia, C. and Mattheyses, R., "A linear-time heuristics for improving network partitions", *Proceedings of the 19th Design Automation Conference*, January, 1982, pp. 175–181.

[8] Saab, Y., "A fast and robust network bisection algorithm", *IEEE Transactions on Computers*, **44**, 903–913, July, 1995.

[9] Hagen, L. W., Huang, D. J. H. and Kahng, A. B. (1997). "On implementation choices for iterative improvement partitioning algorithms", *IEEE Transactions on Computer-Aided Design*, **16**(10), 1199–1205.

[10] Cong, J., Li, H. P., Lim, S. K., Shibuya, T. and Xu, D., "Large scale circuit partitioning with loose/stable net removal and signal flow based clustering", *International Conference on Computer-Aided Design*, 1997, pp. 441–446.

[11] Dutt, S. and Deng, W., "VLSI circuit partitioning by cluster-removal using iterative improvement techniques", *International Conference on Computer-Aided Design*, 1996, pp. 194–200.

[12] Alpert, C. (1998). "The ISPD98 circuit benchmarks suite", *Physical Design Workshop*, pp. 80–85.

Author's Biography

Youssef G. Saab received the B.S. degree in Computer Engineering and the M.S. and the Ph.D. degrees in Electrical Engineering from the University of Illinois at Urbana-Champaign, Urbana, IL, in 1986, 1988 and 1990, respectively. He is currently an associate professor in the department of Computer Engineering and Computer Science at the University of Missouri-Columbia.

From January, 1986 to July, 1990, he was a research assistant at the Coordinated Science Laboratory, Urbana, IL. His research interests include computer-aided design, combinatorial optimization, computational geometry, and graph algorithms.

Dr. Saab is a member of Tau Beta Pi, Eta Kappa Nu, Phi Kappa Phi, and the Golden Key National Honor Society.