A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design

Jason Cong and M'Lissa Smith

Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024

Abstract

In this paper, we present a bottom-up clustering algorithm based on recursive collapsing of small cliques in a graph. The sizes of the small cliques are derived using random graph theory. This clustering algorithm leads to a natural parallel implementation in which multiple processors are used to identify clusters simultaneously. We also present a cluster-based partitioning method in which our clustering algorithm is used as a preprocessing step to both the bisection algorithm by Fiduccia and Mattheyses and a ratio-cut algorithm by Wei and Cheng. Our results show that cluster-based partitioning obtains cut sizes up to 49.6% smaller than the bisection algorithm, and obtains ratio cut sizes up to 66.8% smaller than the ratio-cut algorithm. Moreover, we show that cluster-based partitioning produces much stabler results than direct partitioning.

1 Introduction

1.1 Motivation

A *cluster* is a group of strongly connected components in a circuit. The goal of clustering algorithms is to identify the clusters in a circuit. In VLSI layout design, clustering algorithms can be used to construct the natural hierarchy of the circuit. Many existing layout algorithms generate a circuit hierarchy based on recursive top-down partitioning [16]. Not only does the time and space required by partitioning algorithms increase as circuit sizes increase, but also the stability and quality of their results deteriorate. For example, iterative improvement based partitioning algorithms [14, 8, 17] do not perform well on very large circuits. These algorithms try to avoid local optima in the solution space by allowing the cut size to temporarily increase. However, as circuit sizes increase, the number of local optima becomes very large and these algorithms often fail to discover a cut size close to the global optimum. A poor result early in the top-down partitioning process imposes an unnatural circuit hierarchy and will likely lead to a suboptimal solution.

Bottom-up clustering algorithms provide a solution to the problems encountered when partitioning very large circuits. A bottom-up clustering algorithm can be integrated into the partitioning process by using clustering as a preprocessing step to partitioning. First, clustering is performed on the circuit to obtain a condensed circuit in which each cluster of components has been collapsed to form a single component. Partitioning is then performed on the clustered circuit instead of the original circuit. Since the number of components in a clustered circuit is usually much smaller than that of the original circuit, the time and space required by the partitioning algorithm is reduced significantly. Moreover, our study shows that partitioning the clustered circuit leads to better results than direct partitioning, since strongly connected components in each cluster are not separated during the partitioning process.

1.2 Basic Concepts and Terminology

A netlist is best represented by a hypergraph with each component being represented by a node and each net represented by a hyperedge. However, many clustering and partitioning algorithms [3, 10, 14, 11, 6], including the one presented in this paper, use a graph representation of the netlist rather than a hypergraph representation. An r-terminal net is represented by an r-clique in the graph representation. An r-*clique* is a complete graph with r nodes, and the number of edges in an r-clique is $\binom{r}{2}$. The edges of the clique are usually weighted according to the size of the net. Several weighting functions have been proposed, including $\frac{2}{r-1}$, $\frac{2}{r}$, or $\frac{4}{r^2-mod(r,2)}$ [4, 13, 7]. In essence, these weighting functions assign smaller weights to edges in larger nets. Our clustering algorithm uses the weighting function $\frac{2}{r}$ for an r-terminal net. For this weighting function, the total edge weight contributed by the r-clique is equal to the number of connections required to connect r components, namely $r - 1$.

It is important to be able to determine if a cluster is *good*, i.e. whether the nodes in the cluster are strongly connected. Several clustering metrics have been proposed as follows.

Cluster Density. Given a cluster of c nodes, the cluster density is $\frac{E}{M_c}$, where $M_c = \binom{c}{2}$ and E is the total weight of the edges in the cluster. Clusters with a higher density are considered to be of higher quality. Although, the density metric is perhaps the simplest and most intuitive metric, this metric is biased toward small clusters since the value of M_c increases rapidly as c increases.

k-l-connectedness. In a graph, two nodes are k-l-

connected if and only if there exist k edge-disjoint paths connecting them such that each path has length at most l [10]. The idea is that if two nodes are connected by many short paths, they are strongly connected. However, it is not obvious what values should be assigned to k and l for a given circuit.

Degree/Separation. A more recent metric for determining the quality of clusters is the degree/separation (DS) metric [5, 12]. The cluster *degree* is the average number of nets incident to each component in the cluster. The cluster *separation* is the average length of a shortest path between two components in the cluster. Clusters with a higher DS value are of higher quality. Since this metric considers the global connectivity information, it is a very robust measurement. However, in general, it is costly to compute the DS value for large circuits since computing cluster separation requires $O(n^3)$ time, where n is the number of components in the cluster.

In this paper, we choose to use the density metric due to its simplicity. Certain adjustments are taken into consideration to correct the bias of the density metric toward small clusters. We calculate the density of a clustering solution as the weighted average of the densities of the clusters in the clustered graph, with the density of single nodes equal to zero.

1.3 Previous Work

In this section, we summarize previous work in bottom-up clustering. In contrast to the top-down clustering approaches which are based on recursive partitioning, these bottom-up clustering algorithms repeatedly find locally strongly connected clusters.

The Compaction Algorithm. This method [3] was developed to improve the results of partitioning algorithms such as the Kernighan-Lin algorithm [14] and simulated annealing [15]. Those partitioning algorithms tend to perform poorly on graphs with average degree less than or equal to three [3]. The compaction heuristic increases the average degree of a graph by finding a maximal random matching on the graph. Each edge in the matching represents a cluster, and the two nodes connected by a matching edge are collapsed to form a single node in the compacted graph. This method does not attempt to find natural clusters which are very useful in VLSI design.

The k-l-connectedness Algorithm. This is a constructive algorithm that forms clusters defined by the transitive closure of the k-l-connectedness relation [10]. For arbitrary k, the complexity of this algorithm is $O(d^{2l-1}n)$ where d is the maximum degree of the nodes. Although this approach is more likely to find natural clusters than the compaction heuristic, it is not obvious how to choose k and l for any given netlist. Moreover, when k and l are large, the computational complexity of the algorithm becomes prohibitive for large circuits.

The Random Walk Algorithms. Two bottom-up clustering algorithms that depend on random walks have been developed [5, 12]. A *random walk* begins at one node in the graph and takes n^2 steps through the graph.

The clusters are based on cycles in the node sequence of the random walk. A disadvantage of both random walk algorithms is that they have a time complexity of $O(n^3)$, where n is the total number of nodes in the circuit.

1.4 Overview of the Paper

In this paper, we present a bottom-up clustering algorithm in which clusters are formed by recursively collapsing small cliques. Once a clique is found, if it satisfies the size and density thresholds, the clique is collapsed to make a single node that represents the cluster. The collapsed node may be further clustered allowing clusters of arbitrary size to be formed. The parallel version of our algorithm allows cliques to be found simultaneously by multiple processors which reduces the time required for clustering. When applied to the FM partitioning algorithm [8] and a ratio-cut partitioning algorithm [17], our cluster-based partitioning algorithms give significantly better results.

2 The Clustering Algorithm

2.1 Theoretical Background

In our clustering algorithm, clusters are based on recursive collapsing of small cliques in a graph. In a random graph of n nodes with edge probability p (i.e. p is the probability that there is an edge connecting two nodes), let X_r be the expected number of r-cliques. Then, for most n, there exists an integer r_0, computed by the following formula, such that X_{r_0} is much larger than one and X_{r_0+1} is less than one.

$$r_0 = 2\log_b n - 2\log_b \log_b n + 2\log_b \frac{e}{2} + 1 + o(1) \quad (1)$$

where $b = \frac{1}{p}$ [2]. In other words, the value of r_0 is an approximation of the size of the largest clique in the graph.

test circuit	total nodes	total nets	total pins	r_0
8870	502	494	1541	3.555
bm1	882	902	2908	3.540
PrimGA1	733	902	2908	3.568
PrimSC1	733	902	2908	3.568
5655	921	760	2967	3.423
Test04	1515	1658	5975	3.492
Test03	1607	1618	5807	3.439
Test02	1663	1721	6135	3.425
Test06	1752	1674	6671	3.350
Test05	2595	2751	10,077	3.425
19ks	2844	3282	10,547	3.475
PrimGA2	3014	3029	11,219	3.435
PrimSC2	3014	3029	11,219	3.435
industry2	12,142	12,949	47,193	3.327

Table 1: Approximate Size of Largest Clique (r_0)

We computed r_0 for 14 test circuits taken from the MCNC Layout Synthesis Workshop (see Table 1). The probability p was replaced by the density of the graph

representation of each circuit, where the density was computed as $p = \frac{E}{M_n}$ where E is the total number of edges and $M_n = \binom{n}{2}$. As can be seen in the table, r_0 is usually slightly less than 4. However, since the formulas for X_r and r_0 are derived for random graphs and real circuits are usually more structured, our clustering algorithm always starts with searching for $(r_0 + 1)$-cliques.[1]

2.2 The Sequential Clustering Algorithm

The clustering algorithm consists of three major steps. First, the original netlist is converted to a graph representation. Since a large net contributes many low-weight edges[2] and only a very small fraction of nets have more than five terminals, our clustering algorithm only considers nets of no more five terminals. Experimental results show that removing large nets has little effect on clustering results, but decreases the runtime and memory requirements considerably. In the next step, we search for $(r_0 + 1)$-cliques and r_0-cliques in turn to form clusters. Note that an r-clique does not automatically become a cluster. It has to meet the size and density thresholds as discussed later. Finally, a post-processing step is performed to further reduce the number of *unclustered* nodes. Details of the algorithm are described in the following subsections.

2.2.1 Searching and Forming Clusters

During the clustering step, we first search the entire graph for $(r_0 + 1)$-cliques to form clusters. Then, we search for r_0-cliques to form clusters. Afterward, the value of r_0 is recomputed using the current clustered graph and (r_0+1)-cliques and r_0-cliques are searched for again. The clustering step ends when the searches produce an insufficient number of clusters.

If a clique satisfies the area, size, and density thresholds defined in the next subsection, the nodes in the clique are collapsed to form a single cluster node. The edges that are internal to the clique are removed. For any node v outside of the cluster, all edges that connect v to nodes in the cluster are *bundled* together to form a new edge which connects the node v to the newly formed cluster node. The weight of the new edge is the sum of the weights of the edges that are bundled together.

2.2.2 Cluster Thresholds

A set of nodes in an r-clique does not necessarily form a cluster. In order to qualify as a cluster, the nodes and edges in an r-clique must satisfy two criteria: the area and size thresholds and the density threshold.

Area and Size Thresholds. The *area threshold* is a percentage of the total area of the original graph that no cluster may exceed. Similarly, *size threshold* is a percentage of the total number of nodes in the original graph that no cluster may exceed. (The *size* of a cluster is the total number of single nodes it contains.) The area threshold used in our implementation was 25% of the total area of

the original graph, and the size threshold was 33% of the number of nodes in the original graph.

Density Threshold. The purpose of the density threshold is to further ensure that the nodes in a cluster are strongly connected. It also prevents cliques introduced by multi-terminal nets from automatically becoming clusters. The density of a cluster must be greater than or equal to the density threshold to be accepted. The *density threshold* is $\alpha_n \cdot D$ where α_n is a predetermined factor and D is the density of the graph representation of the original netlist. D is the ratio of the total edge weight to $\binom{n}{2}$ as described in Section 1.2 where n is the number of nodes in the graph representation of the original netlist. The value of α_n determines how much higher the density of the clusters must be than the density of the original graph. Since the density, D, is much smaller for large circuits, higher values of α_n were used for the larger test circuits. In our implementation, $\alpha_n = 4.5$ when $n \leq 1000$ and $\alpha_n = 10.0$ when $n > 1000$. Since large clusters tend to have lower density, the density threshold also helps to control the size of clusters.

2.2.3 Post Processing

After clustering, a post-processing step is executed on the clustered graph to reduce the number of single, unclustered nodes. This helps to balance the sizes of the clusters and further reduce the number of nodes in the clustered graph. In this step, a weighted matching[3] [9] is performed on the clustered graph, and each qualified pair of matched nodes is collapsed into a single node in exactly the same way as a clique. To be qualified, the pair must still satisfy the same area, size, and density thresholds as a clique. However, instead of searching for cliques one by one, the matching-based clustering collapses many pairs of nodes simultaneously. The matching algorithm is executed twice.

2.3 The Parallel Clustering Algorithm

We have developed a parallel version of the clustering algorithm to reduce the runtime for large circuits. The basic idea of the parallel algorithm is to divide the graph among multiple processors. Each processor searches for and forms clusters in its portion of the graph. The processors occasionally swap part of their data to allow cliques that are divided among processors to be found. As the size of the graph is reduced by the clustering process, the number of processors involved decreases until there is only one processor. The coordination of the processors is controlled by a driver. As in the sequential version, the processors search for and form clusters from cliques of size $r_0 + 1$, followed by cliques of size r_0. However, in the parallel version, once a processor has searched for $(r_0 + 1)$-cliques and r_0-cliques, it notifies the driver.

Once all active processors have notified the driver that they are finished clustering, a *swap* takes place. During the swap, the driver randomly pairs the processors. The

---

[1] Our clustering algorithm always rounds r_0 or $r_0 + 1$ to its nearest integer.

[2] For example, a 20-terminal net contributes 190 edges with weight 0.1.

[3] Due to the complexity of performing a weighted matching, a random matching was used during post processing for the largest test circuit, industry2.

test circuit name	Parallel Clustering								
	1 processor			2 processors			4 processors		
	size	density	time	size	density	time	size	density	time
8870	7.8	0.273	23.1	7.6	0.263	10.6	7.7	0.279	9.3
bm1	9.7	0.205	95.5	10.2	0.194	28.2	10.6	0.187	22.8
PrimGA1	12.1	0.176	126.6	12.0	0.180	36.4	11.4	0.183	42.1
PrimSC1	12.1	0.176	129.4	11.2	0.193	43.1	11.2	0.191	43.4
5655	6.7	0.303	40.3	6.7	0.301	37.0	6.2	0.317	29.5
Test04	8.1	0.258	121.7	8.0	0.266	95.5	8.1	0.264	75.8
Test03	7.1	0.290	152.2	6.9	0.297	115.3	6.8	0.304	87.5
Test02	6.4	0.331	167.6	5.7	0.374	138.4	5.7	0.373	102.0
Test06	4.3	0.423	283.3	4.2	0.431	227.7	4.4	0.415	187.7
Test05	6.4	0.318	418.1	6.2	0.326	336.5	6.2	0.324	273.2
19ks	14.7	0.127	445.3	13.1	0.149	335.0	10.8	0.186	238.2
PrimGA2	7.4	0.248	604.9	7.0	0.265	501.5	6.8	0.274	366.6
PrimSC2	7.4	0.248	605.2	7.2	0.255	441.3	6.8	0.273	380.9
industry2	13.5	0.174	9894.4	13.8	0.155	6172.4	11.8	0.180	3319.3
Total Computation Time			13,107.6			8518.9			5178.3

Table 2: Average Cluster Sizes, Clustering Densities, and Total Computation Times

driver directs each pair of processors to perform one of two types of data swaps. The first type is a *normal swap* in which each processor sends half of its nodes and the corresponding edges to the other processor. The second type of swap is a *collapsing swap* in which the processor with fewer nodes sends all of it nodes and edges to the other and becomes inactive. The type of swap to be performed depends on the number of nodes the pair of processors contains and the amount by which each processor has reduced its nodes since the time of the previous swap, i.e. the number of clusters that the processor has produced. In our implementation, a collapsing swap was performed if the pair of processors contained fewer than $\beta \times \frac{N}{P}$ nodes where $0 < \beta < 1$, N is the number of nodes in the original graph, and P is the number of active processors. A collapsing swap was also performed if a processor failed to reduce its nodes by $\gamma \times N$ nodes where $0 < \gamma < 1$.

After a swap, the active processors resume clustering in their own subgraphs. The driver repeats the process of coordinating swaps until there is only one active processor left. The driver allows this processor to finish clustering and to perform the post-processing step.

Implementation. The parallel clustering algorithm was implemented using Maisie [1], a C-based parallel language that enables the algorithm to execute in parallel in a multi-processor environment. The *driver* and each *processor* are executed as processes by Maisie. A process is represented by an **entity** which is similar to a function in C. There is one *driver* entity and one *processor* entity. Execution begins in the *driver* entity. Then, the *processor* processes are started in the *driver* entity by executing the *processor* entity P times to start P *processor* processes.

Sending data from the driver to the processors and exchanging data among the processors require large amounts of data to be sent at one time. Sending large amounts of data at one time can be very slow due to the memory requirements of the message queue. During our experimen-

tation, it was observed that sending the data in smaller pieces and allowing a number of pieces to be received (processed) before sending more data helped reduce communication time. For this reason, a limit was placed on the size of a message (40 nodes or edges in our implementation) and on the number of messages that can be sent before the sender has to wait for an acknowledgement (25 messages in our implementation).

2.4 Clustering Results

Table 2 shows the clustering results and computation times obtained using one, two, and four processors in the clustering algorithm for the 14 test circuits. The table gives the average number of nodes in the clusters (not including single nodes), the density of the clustering (see Section 1.2), and the total computation time in seconds.

The computation times were recorded when executing on a network of Sun SPARC IPC workstations connected by an ethernet. As the *driver* does not do much computation, it resides on the same Sun workstation as one of the *processor* processes. The computation times are in seconds and include time spent executing in both the user and system modes. The times do not include communication time, i.e. any time a process was sleeping while waiting to receive a message.[4] The total computation time for all test circuits is given at the bottom of the table. In general, the quality of the parallel clustering results is very similar to that of the sequential clustering results.

---

[4] Aside from being difficult to compute, communication time is not included in Table 2 because it is topology dependent. Using a network of Suns connected by an ethernet is probably the worst-case topology since all message passing shares the same communication line, and the bandwidth of an ethernet is rather limited. Other topologies, such as a hypercube or a butterfly network, are designed to support more efficient communication between processors. We expect that communication time would have decreased considerably if a multi-processor computer (such as a Sun SPARC-10) were used since the communication overhead would be reduced significantly.

758

test circuit	Best Cut		Average Cut		Best Ratio Cut		Average Ratio Cut	
	FM	FMC	FM	FMC	RFM	RFMC	RFM	RFMC
8870	18	15	27.4	17.3	3.99E-05	3.74E-05	8.23E-05	4.43E-05
bml	65	53	82.9	69.6	2.28E-05	7.96E-06	3.03E-05	8.94E-06
PrimGA1	48	49	69.2	64.7	2.04E-05	7.67E-06	2.78E-05	8.89E-06
PrimSC1	46	45	74.0	56.2	3.38E-05	1.23E-05	4.67E-05	1.57E-05
5655	54	48	67.9	62.6	8.76E-06	3.64E-06	1.10E-05	6.02E-06
Test04	44	44	46.0	48.4	9.96E-08	1.07E-07	1.15E-07	1.42E-07
Test03	93	64	137.9	80.1	8.34E-07	4.80E-07	1.10E-06	6.99E-07
Test02	121	80	181.1	105.9	5.68E-08	5.68E-08	1.83E-07	1.40E-07
Test06	60	64	79.9	74.8	1.32E-06	9.59E-07	1.79E-06	1.14E-06
Test05	42	42	61.3	57.5	3.25E-08	3.79E-08	4.22E-08	4.23E-08
19ks	151	129	173.7	156.1	5.98E-06	2.37E-06	7.37E-06	3.50E-06
PrimGA2	246	130	284.0	181.3	1.53E-05	5.34E-06	1.86E-05	6.49E-06
PrimSC2	199	130	277.2	229.1	1.83E-05	6.08E-06	2.33E-05	8.32E-06
industry2	458	315	812.6	402.3	1.34E-05	8.55E-06	1.97E-05	1.06E-05
Avg. Improvement	16.6%		21.2%		37.9%		44.8%	

Table 3: Best and Average Cut Sizes and Ratio Cut Sizes

However, the computation time for clustering is reduced considerably as the number of processors increases.

3 Cluster-based Partitioning

The cluster-based partitioning algorithm uses the clustering algorithm as a preprocessing step for partitioning. Clustering is performed on the original graph, and then partitioning is performed on the clustered graph instead of the original graph. Afterwards, the partitioned clustered graph is *unclustered* without changing the partition. The areas of the two subsets formed by partitioning may not be as close to equal as desired due to the existence of large clusters in the clustered graph. Therefore, after unclustering it is usually necessary to refine the partition in order to further balance the areas of the subsets.

Instead of completely unclustering the clusters in one step, our cluster-based partitioning method gradually unclusters the clusters following the cluster hierarchy. After partitioning the clustered graph into subsets V_1 and V_2, we replace each cluster node by the r nodes (clusters) in the r-clique used to form that cluster. The resulting subsets V_1' and V_2' are used as an initial partition for partitioning the next level of the cluster hierarchy. This process of gradual unclustering and partitioning is repeated for a predetermined number of times. Then, refinement is performed on the completely unclustered netlist. As the clusters gradually become smaller, the areas of the two partitions gradually become more balanced.

We have applied our cluster-based partitioning approach to the bisection algorithm of Fiduccia and Mattheyses (the FM algorithm) [8] and to the ratio-cut algorithm of Wei and Cheng (the RFM algorithm) [17]. The clustering portion of the cluster-based partitioning method was implemented in Maisie a C-based parallel language [1] as described in Section 2.3. The FM and RFM partitioning algorithms were implemented in C as described in [8] and [17]. Experiments were executed on a network of Sun workstations.

We have compared the results obtained by our cluster-based partitioning method to the results obtained using the partitioning algorithms directly on the 14 test circuits. FM and RFM refer to the FM and the RFM algorithms when applied directly (without clustering) to the original netlist. FMC and $RFMC$ refer to the corresponding cluster-based partitioning.

Table 3 compares the best and average cut sizes for FM and FMC and compares the best and average ratio cut sizes for RFM and RFMC. The best and average (ratio) cut sizes are obtained from 10 executions. On average, the best cut size for FMC is 16.6% lower than the best cut size for FM, and the average cut size for FMC is 21.2% lower than the average cut size for FM. In fact, for 5 of the 14 test circuits, the *average* cut size produced by FMC is lower than the *best* cut size produced by FM. On average, the best ratio cut size for RFMC is 37.9% lower than the best ratio cut size for RFM, and the average ratio cut size for RFMC is 44.8% lower than the average ratio cut size for RFM. Furthermore, for 10 of the 14 test circuits, the *average* ratio cut size produced by RFMC is lower than the *best* ratio cut size produced by RFM. These results suggest that our cluster-based partitioning algorithm produces much more stable results than the FM and RFM algorithms.

Table 4 shows the partitioning results obtained when using the parallel clustering algorithm with up to four processors. For FMC, the cut sizes produced by the 1-processor, 2-processor, and 4-processor implementations are on average lower than the cut sizes for FM by 16.6%, 11.9%, and 11.8%, respectively. For RFMC, the ratio cut sizes produced by the 1-processor, 2-processor, and 4-processor implementations are on average lower than the ratio cut sizes for RFM by 37.9%, 29.0%, and 33.7%, respectively. These results further confirm that our parallel clustering algorithm produces high-quality clusterings.

4 Future Work

Our clustering algorithm can also be applied to the large scale placement problem. The basic approach of cluster-

test circuit name	FM	FMC			RFM	RFMC		
		1	2	4		1	2	4
	cut	cut	cut	cut	ratio	ratio	ratio	ratio
8870	18	15	16	15	3.99E-05	3.74E-05	2.57E-05	2.57E-05
bm1	65	53	62	44	2.28E-05	7.96E-06	7.56E-06	8.98E-06
PrimGA1	48	49	38	65	2.04E-05	7.67E-06	1.06E-05	7.67E-06
PrimSC1	46	45	55	38	3.38E-05	1.23E-05	1.23E-05	1.16E-05
5655	54	48	54	56	8.76E-06	3.64E-06	3.64E-06	3.64E-06
Test04	44	44	44	42	9.96E-08	1.07E-07	9.96E-08	9.54E-08
Test03	93	64	62	68	8.34E-07	4.80E-07	5.66E-07	4.86E-07
Test02	121	80	95	81	5.68E-08	5.68E-08	1.24E-07	1.03E-07
Test06	60	64	68	62	1.32E-06	9.59E-07	9.17E-07	8.71E-07
Test05	42	42	42	42	3.25E-08	3.79E-08	3.95E-08	3.72E-08
19ks	151	129	119	136	5.98E-06	2.37E-06	2.37E-06	3.37E-06
PrimGA2	246	130	177	124	1.53E-05	5.34E-06	5.20E-06	5.20E-06
PrimSC2	199	130	150	219	1.83E-05	6.08E-06	6.08E-06	6.08E-06
industry2	458	315	300	335	1.34E-05	8.55E-06	1.10E-05	9.63E-06
Average Improvement		16.6%	11.9%	11.8%		37.9%	29.0%	33.7%

Table 4: Cut Sizes with Parallel Clustering

based placement would be similar to that of cluster-based partitioning. After clustering, the clusters are first placed on the layout surface. Then, as in cluster-based partitioning, the clusters are gradually unclustered, and placement is performed within each cluster on the sub-clusters that result from one level of unclustering. After placing the sub-clusters, placement is performed within the sub-clusters. This process can be repeated a number of times until every component has been placed. Placement within the clusters could take place in parallel. Finally, a global refinement can be performed to further improve the placement solution. We believe that this cluster-based placement method can handle designs of very high complexity and produce high-quality placement solutions.

Acknowledgment

References

[1] R.L. Bagrodia and W. Liao. Maisie: A language and optimizing environment for distributed simulation. In *Proc. of 1990 SCS Multiconference on Distributed Simulation*, pages 205–210, San Diego, CA, Jan. 1990.

[2] B. Bollobas. *Random Graphs*. Academic Press, London, 1985.

[3] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms. *26th ACM/IEEE DAC*, pages 775–778, 1989.

[4] H.R. Charney and D.L. Plato. Efficient Partitioning of Components. In *Proc. of the 5th Annual Design Automation Workshop*, pages 16–0 to 16–21, 1968.

[5] J. Cong, L. Hagen, and A. B. Kahng. Random Walks for Circuit Clustering. In *Proc. IEEE Intl. Conf. on ASIC*, pages 14.2.1–14.2.4, June 1991.

[6] J. Cong, L. Hagen, and A. B. Kahng. Net Partitions Yield Better Module Partitions. In *Proc. ACM/IEEE Design Automation Conf.*, 1992.

[7] W. E. Donath. Logic Partitioning. In *Physical Design Automation of VLSI Systems*, B. Preas and M. Lorenzetti, editors, pages 65–86. Benjamin/Cummings, 1988.

[8] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. 19th Design Automation Conference*, pages 175–181, 1982.

[9] H. Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*. PhD thesis, Stanford University, 1973.

[10] J. Garbers, H.J. Promel, and A. Steger. Finding Clusters in VLSI Circuits. *ICCAD'90*, pages 520–523, 1990.

[11] L. Hagen and A. B. Kahng. Fast Spectral Methods for Ratio Cut Partitioning and Clustering. In *Proc. IEEE Intl. Conf. on Computer-Aided Design*, pages 10–13, 1991.

[12] L. Hagen and A. B. Kahng. A New Approach to Effective Circuit Clustering. In *Proc. IEEE Intl. Conf. on Computer-Aided Design*, Santa Clara, Nov. 1992.

[13] M. Hanan and J.M. Kurtzberg. A Review of the Placement and Quadratic Assignment Problems. *SIAM Review*, 14:324–342, 1972.

[14] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291–307, Feb. 1970.

[15] S. Kirkpatrick, C. Gelatt Jr., and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 13 1983.

[16] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Chichester, 1990.

[17] Y.C. Wei and C.K. Cheng. Towards Efficient Hierarchical Designs by Ratio Cut Partitioning. In *Proc. IEEE Intl. Conf. on Computer-Aided Design*, pages 298–301, 1989.