A Hybrid Multilevel/Genetic Approach for Circuit Partitioning

Charles J. Alpert, Lars W. Hagen†, and Andrew B. Kahng

UCLA Computer Science Department, Los Angeles, CA 90095-1596
† Cadence Design Systems, San Jose, CA 94135

Abstract

This work presents a genetic circuit partitioning algorithm that utilizes the Metis graph partitioning package originally designed for sparse matrix computations [16]. Metis is an extremely fast iterative partitioner that uses multilevel clustering. We have adapted Metis to partition circuit netlists and have integrated a genetic technique that uses previous Metis solutions to help construct new Metis solutions. Our experiments show that this genetic technique produces better results than Metis alone, and in addition, produces cut size that are competitive with previous methods [21] [19] [7] while using less CPU time.

1 Introduction

The *circuit bipartitioning* problem is given as follows: we are given a netlist hypergraph $H(V, E)$ containing n modules $V = \{v_1, v_2, \ldots v_n\}$; a hyperedge (or *net*) $e \in E$ is defined to be a subset of V with size greater than one. A *bipartitioning* $P = \{X, Y\}$ is a pair of disjoint *clusters* (i.e., subsets of V) X and Y such that $X \cup Y = V$. The *cut* of a bipartitioning $P = \{X, Y\}$ is the number of nets which contain modules in both X and Y, which can be written as $|\{e \mid e \cap X \neq \emptyset, e \cap Y \neq \emptyset\}|$. Given a size threshold r, the min-cut bipartitioning problem seeks a solution $P = \{X, Y\}$ with minimum cut such that $\frac{1-r}{2} \leq |X|, |Y| \leq \frac{1+r}{2}$.

The de facto industry standard for circuit bipartitioning is the iterative improvement method based on the Kernighan-Lin (KL) [17] and Fiduccia-Mattheyses (FM) [8] algorithms. The KL algorithm begins with each module v randomly assigned to either X or Y and proceeds in a series of *passes*. During a pass modules are successively swapped between X and Y until each module has been swapped exactly once. The order in which modules are swapped is determined by greedily finding the module pair $v \in X, w \in Y$ that will minimize the cut of the new solution $\{X - \{v\} + \{w\}, Y - \{w\} + \{v\}\}$ generated by the swap. The reduction in cut is called the *gain* of the swap. After each pass, the best solution observed during the pass becomes the initial solution for a new pass; the passes terminate when no further improvement is observed. The FM algorithm works similarly except single modules are moved from one cluster to the other instead of swapped between clusters; this modification permits a linear time implementation of a single pass. FM

has become widely used due to its short runtimes, ease of implementation, and high-quality results; hence, much work has sought to improve the basic FM algorithm.

One significant improvement is to have a tie-breaking mechanism to choose among multiple modules with the same gain; in this vein, Krishnamurthy [18] proposed a lookahead tie-breaking mechanism. Later, Sanchis produced a multi-way partitioner that also uses this tie-breaking mechanism, and her algorithm has since become the industry standard for multi-way partitioning. Recently, Hagen, Huang, and Kahng [9] have studied various tie-breaking schemes and concluded that using a "last-in-first-out" scheme based on the order that modules are moved in FM is significantly better than random or "first-in-first-out" tie-breaking schemes. Finally, Saab [22] has also exploited the order in which modules are moved to produce an improved FM variant.

A second significant improvement to FM has been to integrate *clustering* into a "two-phase" methodology. A k-way clustering of $H(V, E)$ is a set of disjoint clusters $P^k = \{C_1, C_2, \ldots C_k\}$ such that $C_1 \cup C_2 \cup \ldots \cup C_k = V$ where k is sufficiently large.[1] For ease of notation, we write the input netlist $H(V, E)$ as $H_0(V_0, E_0)$. A clustering $P^k = \{C_1, C_2, \ldots, C_k\}$ of H_0 induces the *coarser netlist* $H_1(V_1, E_1)$ with $V_1 = \{C_1, C_2, \ldots, C_k\}$. For every $e \in E_0$, the net e' is a member of E_1 where $e' = \{C_i \mid \exists v \in e \text{ and } v \in C_i\}$, unless $|e'| = 1$, i.e., each cluster in e' contains some module that is in e. In two-phase FM, a clustering first induces the netlist H_1 from H_0, and then FM is run on $H_1(V_1, E_1)$ to yield the bipartitioning $P_1 = \{X_1, Y_1\}$. This solution then *projects* to a new bipartitioning $P_0 = \{X_0, Y_0\}$ of H_0, where $v \in X_0(Y_0)$ if and only if for some $C_h \in V_1$, $v \in C_h$ and $C_h \in X_1(Y_1)$. Next FM is run again on $H_0(V_0, E_0)$ using P_0 as its initial solution. This second run can be classified as a *refinement* step, which refers to whenever an initially good solution is further improved via local moves and swaps.

Many clustering algorithms for two-phase FM have appeared in the literature (see [2] for an overview of clustering methods and a general netlist partitioning survey). Bui et al. [5] find a random maximal matching in the netlist and "compacts" the matched pairs of modules into $\frac{n}{2}$ clusters; the matching can then be repeated to generate clusterings of size $\frac{n}{4}, \frac{n}{8}$, etc. Hagen and Kahng [10] proposed a random-walk method in which cycles in the walk form the clusters; Cong and Smith [6] compress cliques of modules into clusters; and Alpert and Kahng [1] cluster via graph traversals and dynamic programming. All of these methods when used within two-phase FM significantly improve performance over standard FM. Further, two-phase FM (not including the time needed to cluster) is frequently faster than a single FM run because the second phase generally starts with a good initial solution, thereby converging to a local minimum more quickly.

The "two-phase" approach can be extended to include as many phases as are desired; such an approach

[1] A *partitioning* and a *clustering* are identical by definition, but the term partitioning is generally used when k is small (e.g., $k \leq 10$), and the term clustering is generally used when k is large (e.g., $k = \Theta(n)$) and the average cluster size is a constant. Hence, a bipartitioning can also be written as $P^2 = \{C_1, C_2\}$, but we use the notation $P = \{X, Y\}$ to better distinguish partitioning and clustering and to make notation less cumbersome.

is called *multilevel* and is illustrated in Figure 1 (as in [16]). In a multilevel algorithm, a clustering of the initial netlist H_0 induces the coarsened netlist H_1, then a clustering of H_1 induces H_2, etc. until the most coarsened netlist H_m is constructed ($m = 4$ in the Figure). A partitioning solution $P_m = \{X_m, Y_m\}$ is found for H_m (e.g., via FM) and this solution is projected to $P_{m-1} = \{X_{m-1}, Y_{m-1}\}$. P_{m-1} is then refined, e.g., by using it as a starting solution for an FM run; each projected solution is given by a dotted line and each refined solution is given by a solid dividing line in the Figure. This *uncoarsening* process continues until a refined partitioning of H_0 is derived.

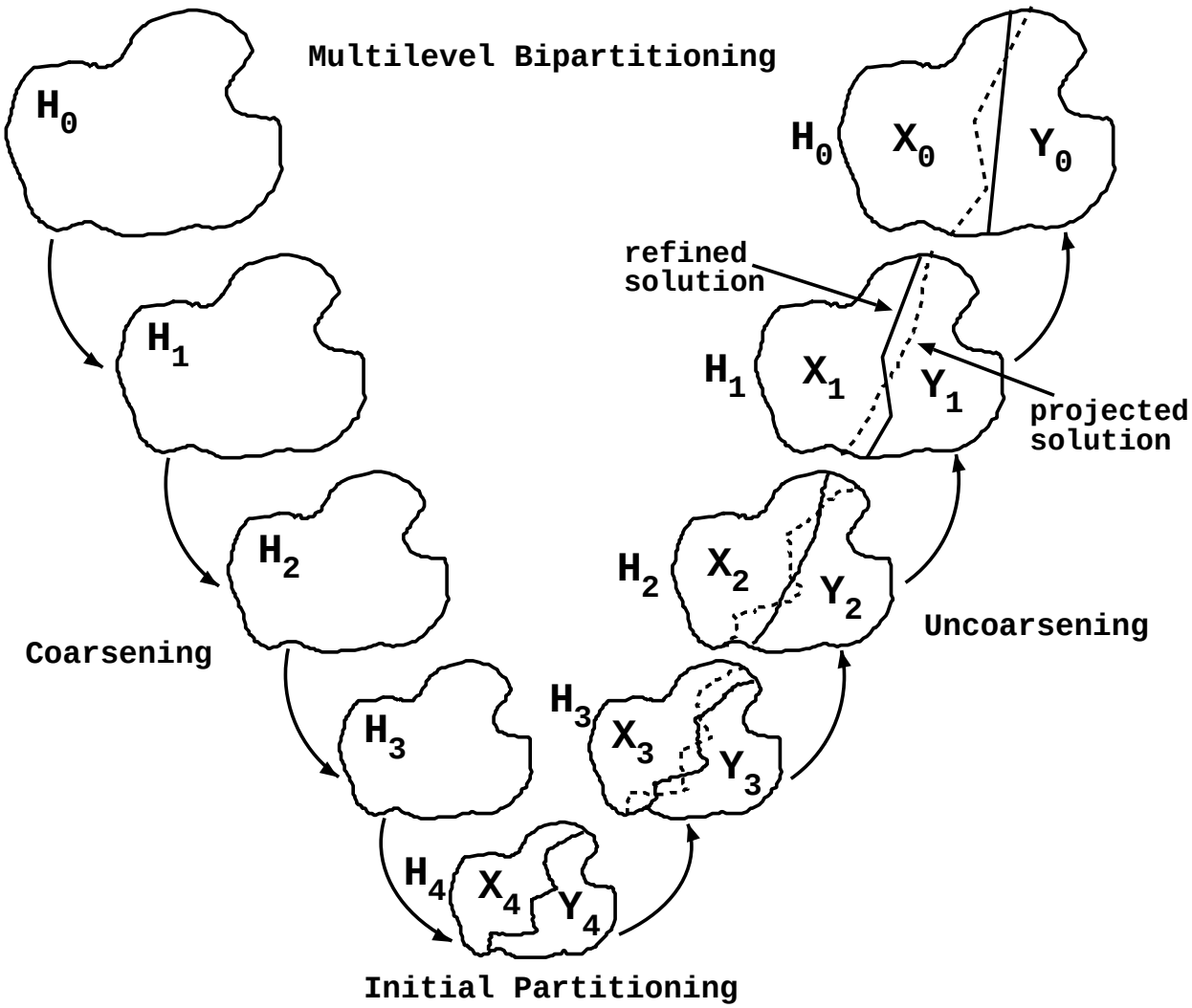

Figure 1: The multilevel bipartitioning paradigm.

Multilevel clustering methods have been virtually unexplored for netlist partitioning; the work of Hauck and Borriello [11] is the notable exception. They performed a detailed study of multilevel partitioning for FPGAs and found that simple connectivity-based clustering combined with a KL and FM multilevel approach produced excellent solutions. Unlike in the physical design literature, multilevel partitioning is

well-studied in the numerical computations literature which includes works by Hendrickson and Leland [12] [13] and Karypis and Kumar [14] [15] [16]; they have respectively developed the Chaco and Metis[2] partitioning packages.

The Metis partitioning package of [16] has produced very good results for finite element graphs and is extremely fast, requiring only 2.8 seconds of CPU time on a Sun Sparc 5 to bipartition a graph with more than 15,000 vertices and 91,000 edges. Considering that Hauck and Borriello [11] showed that this type of multilevel approach can be effective for circuit netlists, we hypothesize that Metis adapted for circuit netlists will be both better and faster than FM. Since Metis runtimes are so low, we can afford to run it over 100 times to generate a solution, but instead of simply generating 100 solutions randomly, we integrate Metis into a genetic algorithm that produces much better average and minimum cuts than Metis alone. Overall our approach generates bipartitioning solutions that are competitive with the approaches of [21] [19] [7] while only requiring a fraction of their runtimes.

The rest of our paper is as follows. Section 2 reviews the Metis partitioning package and presents our modifications for circuit netlists. Section 3 discusses the ideas behind our genetic algorithm and how Metis is integrated. Section 4 presents experimental results for 23 ACM/SIGDA benchmarks, and Section 5 presents directions for future work.

2 The Metis Graph Partitioning Package

Karypis and Kumar developed Metis with multiple options for coarsening, for the initial partitioning step, and for refinement. For example, one can choose from among eight different matching schemes to perform clustering including random, heavy-edge, light-edge, and heavy-clique matching. We follow their recommendations, although their choices are based on extensive empirical studies of finite element graphs and not circuit netlists. Before multilevel partitioning is performed, they randomly permute the adjacency lists for each module. We apply our previous notation to this discussion of graphs, noting that a weighted graph is simply a hypergraph with $|e| = 2$ for each $e \in E$ with a nonnegative weight function on the edges (as opposed to hyperedges) $w : E \rightarrow \Re^+$.

To cluster, Karypis and Kumar suggest the *Heavy-Edge Matching* (HEM) algorithm which is a variant of the random matching algorithm of [5]. A matching M is a subset of E such that no module is incident to more than one edge in M. Each edge in the matching will be contracted to form a cluster, hence the contracted edges should have the highest weights to reduce the contribution to the cut. HEM first visits the modules in random order: if a module u is not incident to an edge in M, the edge (u, v) is added to M such that $w(u, v)$ is maximum with respect to all unmatched modules v; if a module has no unmatched neighbors, it remains unmatched. This algorithm by no means guarantees the maximum sum of weights of

[2] Metis2.0 is available on the world wide web at http://www.cs.umn.edu/users/kumar.

edges in M, but it greedily seeks this objective and runs in $O(|E|)$ time.

After coarsening proceeds through m levels, the netlist H_m at the top level will have few modules; we follow the suggestion of [16] by continuing to coarsen until $|V_m| \leq 100$. The Greedy Graph Growing Algorithm (GGGP) is then applied to generate a bipartitioning of H_m. Initially, one "fixed" module v is in its own cluster X_m and the rest of the modules are in Y_m. Modules with highest gains are greedily moved from Y_m to X_m until $P_m = \{X_m, Y_m\}$ satisfies the cluster size constraints. Since the solution is extremely sensitive to the initial choice of v, the algorithm is run four times with different initial modules, and the best solution observed is retained for the next step.

For the refinement phase, we use the *Boundary Kernighan-Lin Greedy Refinement* (BGKLR) scheme, although despite its name, the heuristic actually uses the FM single-module neighborhood structure. Kumar and Karypis label the KL algorithm "greedy" when only a single pass is performed (they observe that KL typically requires 3 to 5 passes to converge), and propose a hybrid algorithm which completely performs KL when the graph is small (i.e., less than 2000 modules) but performs greedy KL for larger graphs. They believe that greedy KL is only slightly inferior to "complete" KL, but substantial CPU time can be saved by only performing one pass. In addition, they integrate a "boundary" scheme for updating gains: initially, only modules that are incident to cut edges (i.e., boundary modules) are stored in the data structure and eligible to be moved; when a module that is not in the data structure becomes incident to a moved module, it is inserted into the data structure only if it has positive gain. Thus, the cost of performing the boundary version of KL is small relative to the non-boundary version, since only the boundary modules are considered. The overall Metis methodology is presented in Figure 2.

The Metis Algorithm
Input: Graph $H_0(V_0, E_0)$ **Output:** Bipartitioning $P_0 = \{X_0, Y_0\}$
1. i = 0; Randomly permute the adjacency lists of G_0. 2. **while** $\|V_i\| < 100$ **do** 3. Use HEM to find a matching M of H_i 4. Contract each edge in M to form a clustering. Constuct the induced graph $H_{i+1}(V_{i+1}, E_{i+1})$. Set i to $i+1$. 5. Let $m = i$. Apply GGGP to H_m to derive P_m. Refine P_m using BGKLR. 6. **for** i = m-1 **downto** 0 **do** 7. Project solution P_{i+1} to the new solution P_i. Refine P_i using BGKLR. 8. **return** P_0.

Figure 2: The Metis Algorithm

To run Metis on circuit netlist hypergraphs, all that is needed is a hypergraph to graph conversion algorithm. Our goal is to have a fast converter that also retains sparsity; hence, the traditional clique net model (which adds an edge to the graph for every pair of modules in a given net) is not a good choice since large nets will destroy sparsity. Since we observed that keeping large nets generally increases the cut

size regardless of the net model, all nets with more than T modules were removed (in practice, we used $T = 50$). For each net e, our converter picks $F \cdot |e|$ random pairs of modules and add an edge with cost one into the graph for each pair. Here, F is a constant; our experiments showed that the value of F is not too significant as long as it is large enough – we use $F = 5$. Table 1 shows how the cuts of Metis vary with various values of F and T. Some advantages our converter is that it is fairly fast, sparsity is retained, and randomness is introduced. If we run Metis multiple times with different graphs generated by the converter, some of these graphs are more likely to lead to good solutions than others. We did not experiment with many net models, but we hypothesize that the net model does not significantly affect the solution quality.

T/F	1	2	3	4	5	6	8	10	12	15
10	289(238)	241(184)	239(185)	238(180)	225(176)	228(184)	230(174)	220(169)	225(178)	227(169)
15	276(231)	224(188)	239(185)	228(184)	222(178)	228(175)	228(165)	215(176)	241(181)	228(174)
20	320(261)	252(202)	259(180)	258(189)	252(173)	253(187)	261(165)	269(190)	265(176)	253(178)
25	321(243)	251(189)	250(174)	254(170)	243(169)	238(162)	255(173)	232(162)	245(176)	266(174)
35	309(243)	248(172)	239(170)	227(168)	249(173)	245(171)	240(166)	247(164)	254(176)	239(169)
50	316(233)	258(190)	250(177)	251(173)	245(169)	240(167)	255(159)	255(178)	232(162)	240(175)
100	310(231)	274(184)	256(173)	260(172)	256(173)	254(175)	248(166)	237(170)	245(180)	252(176)
200	325(252)	265(182)	266(170)	257(174)	288(184)	258(182)	261(187)	260(192)	271(181)	266(186)
500	471(366)	427(333)	418(318)	412(327)	414(294)	429(295)	399(311)	408(321)	414(296)	411(270)

Table 1: Average(minimum) cuts for avqlarge for 50 runs of Metis shown for various values of T (rows) and F (columns).

Alternatively, we could have modified the internal structures of Metis to handle hypergraphs, but we fear that this may have slow Metis down considerably; we prefer to use Metis as a black box.

3 A Genetic Version of Metis

Our experiments in Section 4 show that over the course of 100 independent runs Metis generates at least one very good solution, but that Metis' performance is not particularly stable generating average cuts much higher than minimum cuts. To try to stabilize solution quality and generate superior solutions, we have integrated Metis into a genetic framework.

An *indicator vector* $\vec{p} = \{p_1, p_2, \ldots, p_n\}$ for a bipartitioning $P = \{X, Y\}$ has entry $p_i = 0$ if $v_i \in X$ and entry $p_i = 1$ if $v_i \in Y$, for all $i = 1, 2, \ldots, n$. The *distance* between two bipartitionings P and Q with corresponding indicator vectors $\vec{p}$ and $\vec{q}$ is given by $\sum_{i=1}^{n} |p_i - q_i|$, i.e., by the number of module moves needed to derive solution Q beginning from solution P. Boese et al. [4] showed that the set of local minima generated by multiple FM runs exhibit a "big valley" neighborhood structure: solutions with smallest distance to the local minima with lowest cost also have low cost, and the best local minima are "central" with respect to the other local minima.

Thus, we seek to combine several local minima solutions generated by Metis into a central or average

solution. Given a set S of s solutions, the s-digit binary code $C(i)$ for module v_i is generated by concatenating the i^{th} entries of the indicator vectors for the i solutions. A clustering is then constructed by assigning two modules v_i and v_j to the same cluster if $C(i)$ and $C(j)$ are the same code, The idea is that since v_i and v_j appear in the same cluster in each one of the s local minima solutions, v_i and v_j should also be in the same cluster in the new central solution (since good solutions have small distance to other good solutions). Our strategy integrates this code-generated clustering into Metis.

In practice, we find that these code-generated clusterings are fairly coarse, generally consisting of between 100 and 300 clusters (with s between 9 and 15). A clustering of this size may be best suited for the top level in the clustering hierarchy, i.e., the netlist induced by this hierarchy is P_m. However, to exploit Metis' multilevel approach, many levels of clusterings must be generated before this clustering is applied at the top level. We use the same HEM clustering algorithm, but only consider matching module v_i to module v_j if $C(i) = C(j)$. This scheme ensures that every clustering generated during coarsening will be a refinement of this code-based clustering.[3] The Genetic Metis (GMetis) algorithm is shown in Figure 3.

The Genetic Metis (GMetis) Algorithm
Input: Hypergraph $H(V, E)$ with n modules **Output:** Bipartitioning $P = \{X, Y\}$
Variables: s: Number of solutions $numgen$: Number of generations $C(i)$: s-digit code for module v_i S: set of the s best solutions seen G: graph with n modules
1. Set $C(i) = 00\ldots0$ for $1 \leq i \leq n$ 2. **for** i = 0 **to** $numgen - 1$ **do** 3. **for** j = 0 **to** $s - 1$ **do** 4. **if** $((i \cdot s) + j)$ modulo $10 = 0$ **then** convert H to graph G 5. P = Metis(G) (using HEM modification with codes $C(i)$) 6. **if** $\exists Q, Q \in S$ such that Q has larger cut then P **then** $S = S + P - Q$. 7. **if** $i > 0$ **and** $(s(i-1) + j)$ modulo $5 = 0$ **then** compute $C(i)$ for $1 \leq i \leq n$ using S. 8. **return** P where P is the solution in S with lowest cut.

Figure 3: The Genetic Metis Algorithm

Step 1 initially sets all codes to $00\ldots0$ which causes GMetis to behave just like Metis until s solutions are generated. Steps 2 and 3 are loops which cause $numgen$ generations of s solutions to be computed. Next, in Step 4 a graph is constructed from the hypergraph, but this step is performed only once out of every 10 times Metis is called. This step is not performed every time so that runtimes are reduced, but it is performed more than once to ensure a diverse population of solutions; the constant 10 is fairly arbitrary. In Step 5, Metis is called using our version of HEM described above. Step 6 maintains the set of solutions

[3] A clustering P^k is a *refinement* of Q^l if some division of clusters in Q^l will yield P^k.

S; our replacement scheme replaces solution $Q \in S$ with solution P if P has smaller cut size than Q, but other replacement schemes may work just as well if not better. Step 7 computes the binary code for each module based on the current solution set, but only after the first generation has completed and every five solutions with the previous code clustering have been generated. As in Step 4, the constant 5 is fairly arbitrary. Finally, the solution with lowest cut is returned in Step 8.

4 Experimental Results

All our experiments use a subset of the benchmarks from the ACM/SIGDA suite given in Table 2; all of these circuits are available on the world wide web at http://ballade.cs.ucla.edu:8080. All our experiments assume unit module areas; the code is written in C++ and compiled with g++ (v. 2.4) on a Unix platform. Our experiments were all run on a Sun Sparc 5 and all runtimes reported are for this machine (in seconds) unless otherwise specified. We performed the following studies:

- We show that Metis compares favorably to standard and to two-phase FM, showing the effectiveness of this multilevel approach versus one or two-level approaches.
- We show that the GMetis algorithm is more effective than running Metis multiple times.
- Finally, we show that GMetis is very competitive with previous approaches, while requiring only a fraction of the runtime.

4.1 Metis vs FM and Two-phase FM

Our first set of experiments compares Metis with both FM and two-phase FM. We ran Metis 100 times with balance parameter $r = 0$ (exact bisection) and recorded the minimum cut observed in the second column of Table 3. Since there are many implementations of FM with varying data structures and tie-breaking schemes (some of which are better than others), we compare to the best FM results found in the literature.

Dutt and Deng [7] have implemented a very efficient version bucket version of FM, and they report the best results for 100 FM runs for exact bisection; these results are given in the third column of Table 3; with corresponding Sparc 5 run times given in the last column. Hagen et al. [9] have performed a detailed study analyzing tie-breaking schemes within FM and concluded that when choosing the module with highest gain a LIFO (last-in-first-out) strategy significantly outperforms FIFO (first-in-first-out) and random strategies. Further, they proposed a new lookahead function that is superior to Krishnamurthy's [18]; we report their optimized min-cut bisection results the fourth column. Finally, we compare to various two-phase FM strategies. In the fifth column, we report the best two-phase FM results observed for various clustering algorithms (including [5] [10] [1]) as reported in [1] and [9] (optimized with LIFO tie-breaking).

Test Case	# Modules	# Nets	# Pins
balu	801	735	2697
bm1	882	903	2910
primary1	833	902	2908
test04	1515	1658	5975
test03	1607	1618	5807
test02	1663	1720	6134
test06	1752	1541	6638
struct	1952	1920	5471
test05	2595	2750	10076
19ks	2844	3282	10547
primary2	3014	3029	11219
s9234	5866	5844	14065
biomed	6514	5742	21040
s13207	8772	8651	20606
s15850	10470	10383	24712
industry2	12637	13419	48404
industry3	15406	21923	65792
s35932	18148	17828	48145
s38584	20995	20717	55203
avqsmall	21918	22124	76231
s38417	23849	23843	57613
avqlarge	25178	25384	82751
golem3	103048	144949	338419

Table 2: Benchmark circuit characteristics.

We observe that Metis does not appear to be any faster than FM for circuits with 2000 modules or less, but for larger circuits with 5-12000 modules, Metis is a factor of 2-3 times faster. In terms of solution quality, again Metis is not very distinguishable from FM for the smaller benchmarks, but for the larger benchmarks Metis cuts are significantly lower, including more than a 50% reduction in cut size versus the best known FM results for avqsmall and avqlarge. We conclude that Metis' multilevel approach is likely unnecessary for small circuits, but as circuit size grow larger, the performance of FM (and two-phase FM) tends to degrade while Metis' performance does not. These large circuits require more than two levels of clustering for such an iterative approach to be effective.

4.2 Genetic Metis vs Metis

Our next set of experiments compares Metis with the GMetis. We ran GMetis for 10 generations with $s = 10$ solutions maintained per generation to ensure that the total number of solutions generated was 100 for both Metis and GMetis. We report the minimum and average cuts observed as well as total CPU time for both algorithms in Table 4.

On average, GMetis yields minimum cuts that are 2.7% lower than Metis, yet the average cut generated is uniformly better (except for golem3). However, for the larger benchmarks (7-26000 modules) the im-

Test Case	Minimum cut (100 runs)				CPU Time (s)	
	Metis	FM[7]	FM[9]	2-FM[9][1]	Metis	FM[7]
balu	34	32			23	21
bm1	53	55		52	22	24
primary1	55	57	56	53	22	24
test04	53	86		56	37	41
test03	61	72		60	39	56
test02	99	115		97	43	46
test06	94	71		68	53	50
struct	36	45	36	43	41	46
test05	107	97		93	69	81
19ks	116	142		121	59	115
primary2	158	236	171	182	90	128
s9234	49	53			72	222
biomed	83	83	83	124	134	296
s13207	84	92			111	339
s15850	62	112			123	339
industry2	218	428	275	438	349	727
industry3	292		312	328	399	
avqsmall	175		373	399	293	
avqlarge	171		406	518	355	

Table 3: Comparison of Metis with FM.

provement is 5.7% better, with significant improvements for industry2, avqsmall and avqlarge. We believe that GMetis can have the greatest impact for larger circuits. Note that golem3 is the one benchmark in which Metis outperforms GMetis in terms of the average case – the quality of GMetis solutions gradually become worse with subsequent generations instead of converging to single good solutions. We discuss this issue in the last section.

4.3 Genetic Metis vs Other Approaches

Our final set of experiments compares GMetis to other partitioning works in the literature, namely PROP [7], Paraboli [21], and GFM [19], the results of which are quoted from the original sources and presented in Table 5. All these works use $r = 0.1$, i.e., each cluster must contain between 45% and 55% of the number of modules. The CPU times in seconds given for PROP, Paraboli, and GFM are respectively for a Sun Sparc 5, a DEC 3000 Model 500 AXP, and a Sun Sparc 10. We modified GMetis to handle varying size constraints by allowing the BGKLR refinement algorithm to move modules as long as the cluster size constraints were satisfied. The size constraints were passed down throughout the hierarchy. We found that with this implementation, GMetis with $r = 0.1$ was sometimes outperformed by GMetis with $r = 0$ (exact bisection). Hence, in Table 5, we present results for GMetis with $r = 0.1$ and $r = 0$ (given in parentheses). Since for $r = 0.1$, GMetis runtimes sometimes increase by 20-50%, we report runtimes for $r = 0$ in the last column. For these experiments we used $s = \log_2 n$ ($n = |V|$) solutions and 12 generations, although these

Test Case	Metis			GMetis		
	min	avg	CPU	min	avg	CPU
balu	34	47.0	26	32	37.6	24
bm1	53	64.6	23	54	58.6	22
primary1	55	65.5	23	55	59.2	21
test04	53	67.9	37	52	58.2	37
test03	61	75.7	42	65	74.0	39
test02	99	113.3	44	96	101.0	42
test06	94	117.1	59	97	120.5	55
struct	36	52.2	41	34	40.0	39
test05	107	124.8	70	109	117.4	69
19ks	116	131.6	59	112	116.1	59
primary2	158	195.2	95	165	174.3	91
s9234	49	66.1	71	45	51.8	68
biomed	83	149.0	145	83	134.3	143
s13207	84	90.3	106	78	88.6	112
s15850	62	83.9	126	59	73.8	125
industry2	218	280.0	336	204	229.9	339
industry3	292	408.0	384	291	313.3	423
s35932	55	71.2	257	56	61.8	265
s38584	55	100.8	310	53	66.9	368
avqsmall	175	240.8	289	148	174.0	322
s38417	73	109.6	294	74	104.1	301
avqlarge	171	247.8	318	144	181.3	355
golem3	2196	2520.1	1592	2196	2647.6	1928

Table 4: Comparison of Metis with Genetic Metis. The minimum cut, average cut and CPU time for 100 runs of each algorithm are given.

parameters are somewhat arbitrary.

We observe that GMetis cuts are competitive with the other methods, especially for the larger benchmarks s15850, industry2, and avqsmall. However, the big win for GMetis is the short runtimes, which are often a factor of 2 or better than the other approaches. Indeed, generating a single solution for avqlarge and golem3 respectively takes $417/(12 \log_2 25178) = 2.5$ and $450/(2 \log_2 103048) = 15$ seconds on average. For golem3, we only ran 2 generations since the results do not improve with subsequent generations; the solution with cost 2144 was achieved after only 210 seconds of CPU time.

5 Conclusions

This work integrates the Metis multilevel partitioning algorithm of [16] into a genetic algorithm. We showed that Metis significantly outperforms previous FM-based approaches, that GMetis improves upon Metis alone, especially for large benchmarks, and that GMetis is competitive with previous approaches while using less CPU time. We conclude that multilevel approaches are a promising source of study for partitioning algorithms, especially for large circuits. There are a number of improvements to GMetis which

Test Case	cuts				CPU			
	PROP	Paraboli	GFM	GMetis(bal)	PROP	Paraboli	GFM	GMetis
balu	27	41	27	27(32)	16	16	24	14
bm1	50			48(53)	20			12
primary1	47	53	47	47(54)	19	18	16	12
test04	52			49(52)	49			21
test03	59			62(66)	51			23
test02	90			95(96)	64			26
test06	76			94(93)	75			32
struct	33	40	41	33(34)	42	35	80	27
test05	79			104(109)	97			46
19ks	105			106(110)	87			39
primary2	143	146	139	142(158)	139	137	224	53
s9234	41	74	41	43(45)	139	490	672	58
biomed	83	135	84	102(83)	250	711	1440	95
s13207	75	91	66	74(70)	177	2060	1920	102
s15850	65	91	63	53(60)	291	1731	2560	114
industry2	220	193	211	177(204)	867	1367	4320	245
industry3		267	241	243(286)		761	4000	299
s35932		62	41	57(55)		2627	10160	266
s38584		55	47	53(53)		6518	9680	397
avqsmall		224		144(145)		4099		328
s38417		49	81	69(77)		2042	11280	281
avqlarge		139		145(144)		4135		417
golem3		1629		2111(2144)		10823		450

Table 5: Comparison of GMetis with PROP, Paraboli, and GFM for min cut bipartitioning allowing 10% deviation from bisection. Exact bisection results for GMetis are given in parentheses in the 5^{th} column.

we are currently pursuing:

- Experiment with other net to graph conversion models; sparser models would yield solutions even faster, and more reliable models would yield higher quality solutions.
- Try various genetic replacement schemes. Currently, we replace the current solution with one of the solution in S if it has a lower cut. A randomized version of this scheme may yield a more diverse population of solutions.
- Tweak parameters. From Table 1, one can see the effect of varying F and T in the net model, and there are many other parameters as well, e.g., the number of times we generate a new graph (every 10 solutions), the number of times we compute clusterings (every 5 solutions), the number of generations, the number of solutions s, etc. These parameters should be tunable to generate more consistent solutions in fewer iterations which could cause speedups of a factor of 2 to 3. Most importantly, proper values of these parameters may enable solutions for golem3 to converge instead of diverge; however, this task would be easier to accomplish if circuits in the 26000-99000 module range existed in the ACM/SIGDA benchmark suite.

- Experiment with various methods to control cluster size constraints. Currently, more liberal constraints do not necessarily increase solution quality, but they do increase runtimes. A possible technique may be to have stricter size constraints for the top level of the hierarchy, and then relax these as lower levels in the hierarchy are reached.
- Finally, we are extending GMetis to multi-way partitioning, and in particular, we plan to integrate GMetis into a quadrisection-based placement algorithm.

References

[1] C. J. Alpert and A. B. Kahng, "A General Framework for Vertex Orderings, with Applications to Netlist Clustering", to appear in *IEEE Trans. on VLSI*.

[2] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration, the VLSI Journal*, 19(1-2), pp. 1-81, 1995.

[3] C. J. Alpert and S.-Z. Yao, "Spectral Partitioning: The More Eigenvectors, the Better," *Proc. ACM/IEEE Design Automation Conf.*, 1995, pp. 195-200.

[4] K. D. Boese, A. B. Kahng, and S. Muddu, "A New Adaptive Multi-Start Technique for Combinatorial Global Optimizations", *Operations Research Letters*, 16(20), pp. 101-13, 1994.

[5] T. Bui, C. Heigham, C. Jones, and T. Leighton, "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms", *Proc. ACM/IEEE Design Automation Conf.*, pp. 775-778, 1989.

[6] J. Cong and M'L. Smith, "A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design", *Proc. ACM/IEEE Design Automation Conf.*, pp. 755-760, 1993.

[7] S. Dutt and W. Deng, "A Probability-Based Approach to VLSI Circuit Partitioning", submitted to *Proc. ACM/IEEE Design Automation Conf.*, 1996.

[8] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions", *Proc. ACM/IEEE Design Automation Conf.*, pp. 175-181, 1982.

[9] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng, "On Implementation Choices for Iterative Improvement Partitioning Algorithms", *IEEE Trans. Computer-Aided Design (submitted)*, 1995.

[10] L. Hagen and A. B. Kahng, "A New Approach to Effective Circuit Clustering", *Proc. IEEE Intl. Conf. Computer-Aided Design*, pp. 422-427, 1992.

[11] S. Hauck and G. Borriello, "An Evaluation of Bipartitioning Techniques", *Proc. Chapel Hill Conf. on Adv. Research in VLSI*, 1995.

[12] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs", Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[13] B. Hendrickson and R. Leland, "The Chaco User's Guide", Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[14] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs", *Technical Report* #95-035, Department of Computer Science, University of Minnesota, 1995.

[15] G. Karypis and V. Kumar, "Multilevel k-Way Partitioning Scheme for Irregular Graphs", *Technical Report* #95-035, Department of Computer Science, University of Minnesota, 1995.

[16] G. Karypis and V. Kumar, "Unstructured Graph Partitioning and Sparse Matrix Ordering", *Technical Report*, Department of Computer Science, University of Minnesota, 1995.

[17] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell Systems Tech. J.*, 49(2), pp. 291-307, 1970.

[18] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks", *IEEE Trans. Computers*, 33(5), pp. 438-446, 1984.

[19] J. Li, J. Lillis, and C.-K. Cheng, "Linear Decomposition Algorithm for VLSI Design Applications", *Proc. IEEE Intl. Conf. Computer-Aided Design*, pp. 223-228, 1995.

[20] L.-T. Liu, M.-T. Kuo, S.-C. Huang, and C.-K. Cheng, "A Gradient Method on the Initial Partition of Fiduccia-Mattheyses Algorithm", *Proc. IEEE Intl. Conf. Computer-Aided Design*, pp. 229-234, 1995.

[21] B. M. Riess, K. Doll, and F. M. Johannes, "Partitioning Very Large Circuits Using Analytical Placement Techniques", *Proc. ACM/IEEE Design Automation Conf.*, pp. 646-651, 1994.

[22] Y. Saab, "A Fast and Robust Network Bisection Algorithm", *IEEE Trans. Computers*, 44(7), pp. 903-913, 1995.

[23] L. A. Sanchis, "Multiple-Way Network Partitioning", *IEEE Trans. Computers*, 38(1), pp. 62-81, 1989.