Multilevel Circuit Partitioning

Charles J. Alpert, *Member, IEEE,* Jen-Hsin Huang, *Member, IEEE,* and Andrew B. Kahng, *Associate Member, IEEE*

***Abstract*— Many previous works in partitioning have used some underlying clustering algorithm to improve performance. As problem sizes reach new levels of complexity, a single application of a clustering algorithm is insufficient to produce excellent solutions. Recent work has illustrated the promise of *multilevel* approaches. A multilevel partitioning algorithm recursively clusters the instance until its size is smaller than a given threshold, then unclusters the instance while applying a partitioning refinement algorithm. In this paper, we propose a new multilevel partitioning algorithm that exploits some of the latest innovations of classical iterative partitioning approaches. Our method also uses a new technique to control the number of levels in our matching-based clustering algorithm. Experimental results show that our heuristic outperforms numerous existing bipartitioning heuristics with improvements ranging from 6.9 to 27.9% for 100 runs and 3.0 to 20.6% for just ten runs (while also using less CPU time). Further, our algorithm generates solutions better than the best known mincut bipartitionings for seven of the ACM/SIGDA benchmark circuits, including golem3 (which has over 100 000 cells). We also present *quadrisection* results which compare favorably to the partitionings obtained by the GORDIAN cell placement tool. Our work in multilevel quadrisection has been used as the basis for an effective cell placement package.**

***Index Terms*— Optimization, partitioning, physical design, placement.**

I. Introduction

A netlist hypergraph $H(V,E)$ has n modules $V = \{v_1, v_2, \cdots v_n\}$; a *net* $e \in E$ is defined to be a subset of V with size greater than 1. 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, i.e., $cut(P) = |\{e | e \cap X \neq \emptyset, e \cap Y \neq \emptyset\}|$. Let $A(v)$ denote the area of $v \in V$ and let $A(S) = \sum_{v \in S} A(v)$ denote the area of a subset $S \subseteq V$. Given a balance tolerance r, the *min-cut bipartitioning problem* seeks a solution $P = \{X,Y\}$ that minimizes $cut(P)$ subject to $(A(V)(1-r)/2 \leq A(X), A(Y) \leq (A(V)(1+r)/2$.

The standard bipartitioning approach is iterative improvement based on the Kernighan–Lin (KL) [29] algorithm, which was later improved by Fiduccia–Mattheyses (FM) [15]. The FM algorithm proceeds in a series of *passes*. A pass begins with some initial solution $\{X,Y\}$; modules are successively moved between X and Y until each module has been moved exactly once. Given a current solution $\{X',Y'\}$, the previously unmoved module $v \in X'$ (or Y') with highest *gain* $(= cut(\{X'-v, Y'+v\}) - cut(\{X,Y\}))$ is moved from X' to Y'. After each pass, the best solution $\{X',Y'\}$ observed during the pass becomes the initial solution for a new pass, and the passes terminate when a pass does not improve upon the most recent solution. FM has been widely adopted by the physical design community due to its short runtimes and ease of implementation.

Iterative approaches dominate both the VLSI CAD literature and industry practice for several reasons. They are generally intuitive (the obvious way to improve a given solution is to repeatedly make it better via small changes), easy to describe and implement, and relatively fast. Hence, much work has sought to improve upon the basic FM algorithm by introducing module tie-breaking schemes [19], [31], by modifying the module locking and unlocking mechanism [11], [23], or by using different formulas for computing the gain [13], [14]. Other works attempt to use iterative improvement inside other algorithmic approaches such as genetic algorithms [9], tabu search [5], large-scale Markov chains [16], two-phase clustering [7], [17], [33], [40], or multilevel clustering [3], [10], [22], [21], [27].

This paper proposes a new multilevel circuit partitioning algorithm. Our work is motivated by the multilevel partitioners of Hendrickson and Leland [22] and Karypis and Kumar [27] which have been very successful in the scientific computing community for partitioning finite-element graphs. In addition to the implementation differences between graphs and netlist hypergraphs, we have added two key ingredients which significantly improves performance.

- We utilize a LIFO bucket scheme for storing module gains [19] and the CLIP algorithm of [14] within our FM implementation.
- We cluster based on the matching algorithms of [7], [22], [27]. However, instead of constructing $(n/2)$ clusters from a set of n modules, we stop the clustering prematurely so that more than $(n/2)$ clusters are generated. This causes the multilevel coarsening to proceed more slowly, which allows the partitioner to explore more levels of the partitioning hierarachy.

The rest of our paper is as follows. Section II surveys the latest innovations in iterative partitioning and discusses our adoption of the CLIP and LIFO improvements within our algorithm. Section III describes our multilevel algorithm, and Section IV describes the matching-based clustering used within the multilevel algorithm. We present extensive exper-

Manuscript received May 2, 1997. This work was supported by a grant from Cadence Design Systems, Inc. This paper was recommended by Associate Editor R. H. J. M. Otten.

C. J. Alpert is with the IBM Austin Research Laboratory, Austin, TX 78758 USA.

J.-H. Huang is with the Physical Product Division, Avant! Corporation, Fremont, CA 94555 USA.

A. B. Kahng is with the Department of Computer Science, University of California, Los Angeles, CA 90024 USA.

Publisher Item Identifier S 0278-0070(98)05825-4.

TABLE I
BENCHMARK CIRCUIT CHARACTERISTICS

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

imental results in Section V that show that our algorithm outperforms numerous other circuit bipartitioning algorithms. Section VI concludes with directions for future work.

II. Innovations in Iterative Partitioning

We now review selected works in iterative partitioning which have provided new innovation (see the survey of Alpert and Kahng [2] for a broader view of previous work in partitioning). In our discussion of the algorithms below, we include some comparisons of these methods (using our implementations) for 23 of the standard benchmarks from the CAD Benchmarking Laboratory (ftp to ftp.cbl.ncsu.edu). Table I shows the characteristics for these test cases, and we assume unit cell area for all test cases. Our experiments were all run on a Sun Sparc 5 (85 MHz), and all runtimes reported are for this machine (in seconds).

A. Tie-Breaking Strategies

One potential problem with the FM algorithm is that many modules in the top bucket may potentially have the same gain; hence, various tie-breaking strategies have been proposed to choose among alternate moves that have the same gain. Krishnamurthy [31] proposed using *lookahead* gain vectors, and Sanchis [39] extended this approach to multiway partitioning. Even when gain vectors are used, ties may still occur in the first- through rth-level gains. Thus, it is the implementation of the gain bucket data structure that determines which module is selected. The original FM algorithm uses a linked list for each bucket; we may infer that modules are probably removed and inserted at the head of the list, i.e., that the bucket organization corresponds to a last-in first-out (LIFO) stack. The authors of [15] do not specifically mention a LIFO organization; one can speculate that LIFO was an “obvious” choice. However, a first-in first-out (FIFO) organization which supports the same update efficiency could have been implemented just as easily. One might even use a random organization, possibly at the cost of increased run times or a more complex bucket structure. The authors of [19] observe that Sanchis [39], and most likely Krishnamurthy [31], used random bucket selection schemes.

In experiments with both the FM and Krishnamurthy algorithms, the authors of [19] found that the LIFO bucket organization is distinctly superior to FIFO and random bucket organizations. Reference [19] ascribes the success of LIFO to its enforcement of “locality” in the choice of modules to move, i.e., modules that are naturally clustered together will tend to move sequentially. Hagen *et al.* [19] use this idea of locality to propose an alternative formula for higher level gains, which also improves performance. That LIFO outperforms FIFO was also observed by Dutt and Deng [14] who, like [19], noted that lookahead tie breaking does not improve the performance of FM when LIFO buckets are used (in other words, using LIFO instead of FIFO negates the advantage of lookahead tie-breaking).

Table II presents our own comparisons of LIFO with random (RND) and FIFO bucket schemes, allowing 10% deviation from exact bisection. Our implementations actually significantly outperform those of [19], perhaps because their implementations were adapted from Sanchis’ original partitioning code (and also because they perform exact bisection). For each of the test cases in the table, we ran FM 100 times for all three bucket schemes; we report the minimum cut, average cut, and standard deviation observed. Like [19] and [14], the table shows that LIFO significantly outperforms FIFO. However, we do not observe any improvement of LIFO over random selection (it appears that random selection may even be the best scheme of the three). In our work below, we use a LIFO scheme since it is much faster than a random scheme within the context of our implementation. Clearly, the discrepancy between these results and those of [19] are a source of concern and need to be further explored.

Recently, Dutt and Deng [13] proposed a different kind of tie-breaking approach, based on probabilistic techniques. Instead of using a gain value that reflects only the immediate change in cut from moving a single vertex, their PROP algorithm uses a more global gain computation. Each vertex has an associated probability for the event that the vertex will actually be moved to the other cluster. PROP begins by assigning each vertex an initial probability of 0.95, and then gains are recomputed based on a function of the current solution and the vertex probabilities. As vertices are moved, probabilities and gains are updated for neighboring vertices. Experiments in [13] show that this gain computation significantly outperforms classic FM. However, since its gain values are nondiscrete, PROP cannot exploit the FM bucket structure; run times thus increase by a factor of 4–8. The heuristic is nevertheless still fairly efficient, and future work on probabilistic gain computations is certainly promising.

B. Modifying the Basic FM Structure

Saab [38] observes that in an iterative improvement algorithm, when a vertex is moved, it tends to “drag” with it its

TABLE II
MINIMUM CUT, AVERAGE CUT, AND STANDARD DEVIATION FOR 100 RUNS OF FM USING THE LIFO, RANDOM (RND), AND FIFO TIE-BREAKING SCHEMES

Test Case	MIN			AVG			STD		
	LIFO	FIFO	RND	LIFO	FIFO	RND	LIFO	FIFO	RND
balu	27	75	27	39	107	39	10	15	10
bm1	47	64	51	76	107	76	14	17	13
primary1	49	57	47	74	111	76	13	18	13
test04	71	139	66	138	208	135	27	26	25
test03	64	112	69	109	184	118	22	32	26
test02	109	185	122	172	169	243	28	18	23
test06	66	146	60	90	196	90	12	19	14
struct	38	131	42	54	184	42	9	16	6
test05	104	251	93	175	335	175	33	29	37
19ks	121	261	120	175	332	180	27	33	28
primary2	215	310	177	285	428	278	44	44	38
s9234	50	246	49	95	335	90	27	28	26
biomed	83	392	83	134	445	130	50	25	42
s13207	87	278	88	129	340	125	20	32	20
s15850	108	416	98	184	506	177	31	32	35
industry2	319	667	304	623	1192	603	171	262	196
industry3	241	408	259	497	2225	491	205	806	187
s35932	113	719	103	230	953	230	61	78	61
s38584	59	1474	54	251	1641	258	106	111	109
avqsmall	319	1415	295	597	1667	624	129	85	122
s38417	167	1120	132	383	1194	381	95	39	102
avqlarge	262	1839	345	787	2024	772	163	78	151

adjacent vertices. His algorithm first performs a sequence of consecutive moves from X to Y, and then clusters the first k vertices moved, reasoning that vertices that are dragged across the cut line together should belong to the same cluster. Like the LIFO bucket scheme, this strategy recognizes that adjacent vertices should be moved sequentially. Saab uses clusters identified in this manner to coarsen the graph, then runs a two-phase FM variant (see the two-phase FM discussion below).

The CLIP algorithm of Dutt and Deng [14] builds upon this idea further by tie breaking based on the adjacency to the most recently moved modules. For example, suppose that moving module v_i increases the gain of v_j by one. Instead of increasing the gain by just one, it could be increased by two, five, ten, etc., which would greatly increase the chance that v_j is moved next. Instead of increasing the gain by some constant factor, the authors of [14] actually propose to increase the gain by an infinite factor. Since the magnitude of the bucket indexes in FM are bounded by a constant, a different implementation is required: 1) the FM buckets are rearranged immediately after the initial gains are computed to start a pass, and 2) all of the buckets in each bucket structure are concatenated into a single linked list starting with the bucket with the largest index. This entire list is then inserted into the bucket with index zero, and all other buckets are made empty. This single preprocessing step has the effect of multiplying the gain change of the most recently moved modules by an infinite factor. The only other modification required is that the range of bucket indexes must double.

Experiments in [14] show that CLIP averages 18% improvement over FM (both using a LIFO bucket scheme).

TABLE III
MINIMUM CUT, AVERAGE CUT, STANDARD DEVIATION, AND CPU TIMES FOR 100 RUNS OF THE FM AND CLIP ALGORITHMS

Test Case	MIN		AVG		STD		CPU	
	FM	CLIP	FM	CLIP	FM	CLIP	FM	CLIP
balu	27	27	39	35	10	10	26	26
bm1	47	47	76	63	14	9	27	29
primary1	49	47	74	62	13	8	27	30
test04	71	55	38	80	27	12	45	63
test03	64	57	109	74	22	14	61	67
test02	109	88	172	112	28	15	49	73
test06	66	60	90	72	12	6	61	65
struct	38	34	54	46	9	7	55	55
test05	104	72	175	72	33	10	92	116
19ks	121	110	175	151	27	18	134	144
primary2	215	143	285	215	44	31	142	168
s9234	50	45	95	74	27	23	273	237
biomed	83	84	134	109	50	26	326	267
s13207	87	78	129	125	20	20	423	370
s15850	108	79	184	143	31	29	435	505
industry2	319	203	623	342	171	89	838	991
industry3	241	242	497	406	205	142	974	1199
s35932	113	45	230	118	61	30	1075	935
s38584	59	48	251	101	106	57	1523	1363
avqsmall	319	204	597	340	129	83	1447	1538
s38417	167	72	383	140	95	33	1595	1423
avqlarge	262	224	787	352	163	79	1662	1896
golem3	2847	2276	3500	3403	296	510	38028	146301

We have implemented the CLIP algorithm and made the same comparisons of CLIP versus FM for bipartitioning with balance tolerance $r = 0.1$. Table III reports the minimum cut, average cut, standard deviation of cut, and total CPU time (Sun Sparc 5) for 100 runs of CLIP and FM on the suite of test cases. We also report significant improvement for CLIP, especially for some of the larger test cases. Interestingly, the run times for CLIP are not much higher than those of FM, except one drastic increase for the very large circuit golem3. The run times decrease for some of the larger test cases for which CLIP requires fewer passes to converge.

Many other works have proposed modifications to the basic FM structure. Observing that each module can be locked only once during a pass, Hoffman [23] proposed an unlocking mechanism that allows modules to move if they have been locked in the "wrong" cluster. Dasdan and Aykanat [11] have proposed a multiway variant of FM that allows a small constant number (e.g., three or four) of module moves per pass. In a similar spirit, Dutt and Deng [14] also propose a promising method called CDIP which allows the iterative improvement algorithm to reverse a sequence of bad moves, and then try some different sequence. Backing up in this manner prevents continuing an entire pass in which positive gain is unlikely to be realized. Yeh *et al.* [44] proposed an extension of Sanchis' multiway partitioning algorithm that alternates "primal" passes of module moves with "dual" passes of net moves; however, run times for dual passes are a factor of 9–10 higher than for a primal pass. In their study on circuit partitioning algorithms, the authors of [21] conclude that dual passes "are not worthwhile." Park and Park [34] propose to integrate size constraints into the cut objective, and Shin and Kim [40] propose to gradually tighten size constraints between FM passes.

These are just some of the many proposed modifications to the basic FM structure. We chose to adopt only CLIP and LIFO within our algorithm because neither of these modifications increases run time significantly, while both enhance solution quality. Whether the run time sacrifices for dual passes, CDIP, or lookahead are worthwhile remains an open direction for future work.

C. Using an Iterative Improvement Engine

As problem sizes grow larger, the performance of iterative improvement approaches such as FM tend to degrade [20]. Hence, many heuristics have utilized iterative improvement within a different paradigm. For example, the genetic partitioning algorithm of Bui and Moon [9] uses FM as a postprocessing step to each crossover operation. (A similar approach was proposed by [25].) FM postprocessing has also been utilized within tabu search-based approaches [4], [5]. Fukunaga *et al.* [16] proposed a large-step Markov Chain (LSMC) algorithm which generates new solutions by making big "jumps" from low-cost local minima. These solutions are then used as starting solutions in FM to generate new local minima (also see Isomoto *et al.* [26]). Liu *et al.* [32] proposed a gradient Fiduccia–Mattheyses algorithm (GFM) that alternates FM refinements with gradient descents. They also propose a variant (GFM$_t$) which uses the two-phase FM technique described below.

Another technique typically used to handle increasing problem sizes is *clustering* or, equivalently, *coarsening*. The modules of the circuit are grouped into many small clusters, and these clusters form the new nodes of a smaller coarser netlist. Iterative improvement is then run on (some of) the clustered netlists. Since our multilevel approach is based on this concept, we now give some formal definitions.

Definition 1: A clustering[1] $P^k = \{C_1, C_2, \cdots, C_k\}$ of H_i *induces* the *coarser netlist* $H_{i+1}(V_{i+1}, E_{i+1})$ with $V_{i+1} = \{C_1, C_2, \cdots, C_k\}$. For every $e \in E_i$, the net e^* is a member of E_{i+1} where $e^* = \{C_h | e \cap C_h \neq \emptyset\}$, unless $|e^*| = 1$, i.e., e^* spans the set of clusters containing modules of e

Definition 2: Suppose that H_{i+1} was induced from H_i by the clustering $P^k = \{C_1, C_2, \cdots, C_k\}$. The *projection* of the bipartitioning solution $P_{i+1} = \{X_{i+1}, Y_{i+1}\}$ of H_{i+1} onto H_i is the solution $P_i = \{X_i, Y_i\}$ where $X_i = \{v \in V_i | \exists C_h \in P^k, v \in C_h, C_h \in X_{i+1}\}$ and $Y_i = \{v \in V_i | \exists C_h \in P^k, v \in C_h, C_h \in Y_{i+1}\}$. The process of projecting P_{i+1} to P_i is called *uncoarsening*.

Clustering has been commonly applied within a "two-phase" methodology. First, a clustering P^k of H_0 is generated, then this clustering is used to induce the coarser netlist H_1 from H_0. FM is then run once on H_1 to yield the bipartitioning P_1, and this solution P_1 is projected to a new bipartitioning P_0 of H_0. Finally, FM is run a second time on H_0 using P_0 as its initial solution. This second FM run can be classified as a *refinement* step, which refers to when an initially good solution is improved via local moves and swaps. The primary difference among two-phase algorithms is the clustering method used to generate P^k. Some common clustering approaches which have been applied to two-phase FM include spectral [3], random walks [17], random matching [7], and bottom-up connectivity-based [33], [40] (see [2] for a survey of circuit clustering techniques).

The "two-phase" approach can be extended to a *multilevel* approach by allowing as many phases as are desired. Fig. 1 illustrates the multilevel partitioning paradigm with five phases or *levels* (as in [27]). In a multilevel algorithm, a clustering of H_0 is used to induce the coarser 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 bipartitioning solution $P_m = \{X_m, Y_m\}$ is found for H_m (e.g., via FM), and this solution is then projected to $P_{m-1} = \{X_{m-1}, Y_{m-1}\}$. P_{m-1} is then refined, e.g., by FM postprocessing (in the figure, the projected and refined solutions are, respectively, denoted by dotted and solid lines). The uncoarsening process continues until a refined partitioning of H_0 is obtained.

Multilevel partitioning offers several advantages over pure iterative partitioning two-phase FM.

- In two-phase FM, coarsening occurs in a single step which may mean that H_1 is too coarse a representation of H_0. Multilevel partitioning allows coarsening to proceed more slowly, which gives the iterative engine more opportunities for refinement.
- If a fast clustering and refinement strategy is used, the approach can be extremely efficient. One can afford to perform a careful partitioning on H_m since this netlist will have very few modules.
- Refinement progresses with progressively larger netlists, which implies that number of modules moved during an FM "move" become progressively smaller. This permits

[1] A k-way clustering P^k of the netlist $H(V, E)$ is a set of disjoint subsets $C_1, \cdots, C_k$ of V such that $C_1 \cup C_2 \cup \cdots \cup C_k = V$. Since a clustering and a partitioning are actually equivalent, we use the superscript k to distinguish between a clustering P^k and a bipartitioning P.

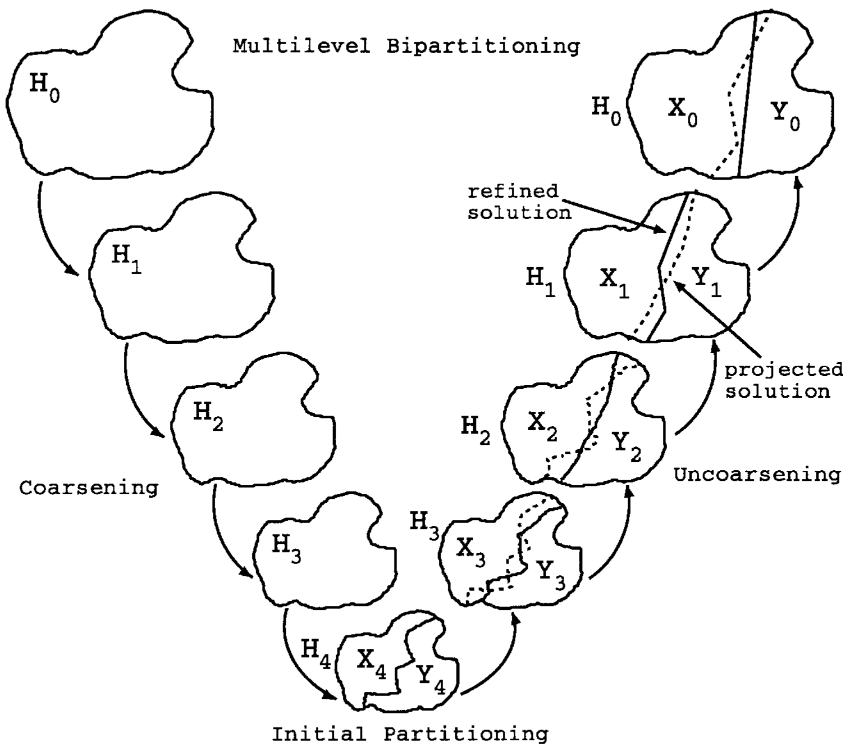


Fig. 1. Multilevel bipartitioning paradigm.

the refinement algorithm to avoid bad local minima via big steps at high levels, but at the same time find a good final solution via refinement at the low levels.

Multilevel partitioning approaches have been especially prominent in the scientific computing literature. Barnard and Simon [6] have used multilevel techniques not directly for partitioning, but rather to compute the Fiedler vector for spectral bisection. Inspired by this work, Hendrickson and Leland [22] developed a very efficient multilevel partitioning algorithm which is included in the Chaco partitioning package. The coarsening step finds a random maximal matching as in [7] and [8], and merges pairs of modules to reduce the instance size by a factor of 2. The refinement step uses multiway FM with a LIFO bucket scheme, but with several modifications to improve run times: 1) the algorithm can terminate before a pass is completed if further improvement appears unlikely, 2) gains are saved after a pass is completed so that only moved modules and their neighbors need to have their gains recomputed before the next pass, and 3) an efficient boundary refinement scheme is used wherein only vertices incident to cut edges are inserted into the data structure, with gains for other vertices computed only on an "as needed" basis. The authors of [35] also proposed a multilevel algorithm but without refinement, i.e., a partition of the coarsest graph is uncoarsened in one step to form the final solution.

Karypis and Kumar [27], [28] recently developed the Metis multilevel graph partitioning package. Like [22], they use boundary schemes and early pass termination. They also allow the user to set options for the clustering scheme, the initial partitioning algorithm, and the refinement scheme. One of their coarsening schemes uses a greedy weighted matching algorithm, upon which our coarsening scheme is based. The work of [1] adapted Metis to partition netlist hypergraphs while integrating the genetic approach of [20] to obtain more stable solution quality.

Cong and Smith [10] proposed applying their clique finding clustering algorithm as the coarsening step in a multilevel circuit bipartitioning algorithm. More recently, Hauck and Borriello [21] performed a detailed study of multilevel FPGA partitioning. They studied many variations of the basic paradigm, including 1) partitioning before and after technology mapping, 2) clustering via shortest paths, pairwise connectivity, random matching, etc., 3) partitioning of the coarsest graph via searches, spectral, and iterative techniques, and 4) uncoarsening in one or multiple steps. Their final algorithm uses simple connectivity-based clustering and iterative improvement with two or three levels of lookahead.

III. A New Multilevel Algorithm

Motivated by the high-solution quality and fast run times of the Chaco and Metis multilevel partitioners as well as new improvements in FM [14], we have implemented our own multilevel partitioner for netlist hypergraphs.[2] One main difference between our multilevel algorithm and previous multilevel partitioners [10], [21], [22], [27] is that a mechanism is provided to control the speed of coarsening, and hence the total number of levels in the netlist hierarchy. We can obtain more levels in the hierarchy than previous approaches by allowing coarsening to proceed more slowly. The advantage

[2] Note that the approach of [3] has to transform the netlist hypergraph to a weighted graph before calling the Metis algorithm [27]. Our implementation coarsens and partitions the hypergraph directly as in [21].

```
ML Multilevel Algorithm
Input:      $H_0(V_0, E_0) \equiv$ Netlist hypergraph
            $T \equiv$ Coarsening threshold
            $R \equiv$ Matching ratio
Variables:  $m \equiv$ Number of levels
            $P^k, \equiv$ Interim clusterings
            $P_i, 1 < i \le m \equiv$ Interim bipartitionings
Output:     $P_0 = \{X_0, Y_0\} \equiv$ Final bipartitioning
1.  $i = 0$.
2.  while $|V_i| > T$ do
3.      $P^k = Match(H_i, R)$.
4.      $H_{i+1}(V_{i+1}, E_{i+1}) = Induce(H_i, P^k)$.
5.      Set $i = i + 1$.
6.  Let $m = i$. $P_m = FMPartition(H_m, NULL)$.
7.  for i = m-1 downto 0 do
8.      $P_i = Project(H_{i+1}, P_{i+1})$.
9.      $P_i = FMPartition(H_i, P_i)$.
10. return $P_0$.
```

Fig. 2. ML multilevel algorithm.

is that more levels allow more opportunities to refine the current solution at the various levels. The result is an efficient partitioner that produces the lowest cost solutions in the literature.

Fig. 2 describes ML, our new multilevel algorithm (which follows the same structure as [22]) for partitioning netlist hypergraphs. The algorithm accepts a netlist H_0 as input along with two user parameters T and R. T specifies that coarsening should proceed as long as the number of modules in the current netlist H_i is greater than T, and R is a parameter used by our *Match* coarsening algorithm explained below. The variable m denotes the number of levels used during coarsening, and the variables P^k and P_i denote intermediate clustering and bipartitioning solutions respectively.

The first five steps in Fig. 2 form the coarsening phase. As long as the number of modules in H_i is more than T, $Match$ is used to form a clustering P^k of H_i. Procedure *Induce* takes a netlist H_i and a clustering P^k and constructs the new netlist H_{i+1} induced by P^k. Note that module areas are preserved, e.g., if P^k contains a cluster with two modules with areas 4 and 7, the module corresponding to this cluster in V_{i+1} will have area 11. The functionality of *Induce* exactly follows Definition 1. Step 5 constructs a bipartitioning of H_m using the *FMPartition* procedure, which takes a netlist and an initial solution as input and returns a refined bipartitioning. If no initial solution is specified, the parameter *NULL* is passed which causes *FMPartition* to start with a random initial solution. Steps 7–9 form the uncoarsening phase. The *Project* procedure takes a netlist H_{i+1} as input and a bipartitioning P_{i+1} of H_{i+1}, then constructs the projection of P_i of P_{i+1} onto P_i of H_i (following Definition 2). The projected solution is then refined via *FMPartition*, and uncoarsening proceeds until a refined partitioning P_0 of H_0 is obtained; this solution is returned in Step 10. The procedures *Match* and *FMPartition* are now discussed in more detail.

A. The Match Coarsening Algorithm

The Chaco [22] and Metis [27] and multilevel algorithms respectively use linear time "random" and "heavy-edge" matching algorithms to construct a clustering. The partitioning study of [21] explored numerous coarsening schemes with varying complexity, yet the authors chose a simple connectivity-based scheme for their multilevel algorithm. Based on the intuitions afforded by of these works, we have also chosen to coarsen via a matching algorithm which loosely follows the heavy-edge matching algorithm used in Metis. In addition, a matching-based approach allows us to control the total number of levels in the netlist hierarchy (as opposed to other approaches, e.g., random walks [17], shortest path clustering [43], and clique compression [10], which automatically determine the number of clusters).

The *Match* algorithm starts by randomly permuting the module indexes, and then visits each module in turn. A *permutation* of $[1 \cdots n]$ is a one-to-one mapping π: $[1 \cdots n] \rightarrow [1 \cdots n]$. For a given module $v = v_{\pi(j)}$, *Match* tries to find the unmatched module w (i.e., a module that has not yet been assigned to a cluster) with highest connectivity to v, where the connectivity between v and w is defined as

$$conn(v, w) = \frac{1}{A(v) \cdot A(w)} \sum_{e \in \{e | v \in e, w \in e\}} \frac{1}{|e|}.$$

The term $(1/|e|)$ emphasizes nets with fewer modules, and the term $(1/A(v) \cdot A(w))$ gives preference to matching modules with smaller areas to help prevent cluster sizes from becoming unbalanced. If such a w can be found, then v and w are matched together to form a new cluster $\{v, w\}$. If no unmatched w exists (i.e., all of the neighbors of v are matched), then the singleton cluster $\{v\}$ is created. When computing the *conn* function, nets with more than ten modules are ignored to reduce runtimes.

The matching algorithms of [22], [27] both seek maximal matchings which will generally force the ratio of $|V_i|$ to $|V_{i+1}|$ to be $(1/2)$. For example, if the parameter T is set to 100, then a netlist with 3000 modules will likely generate five coarser netlists during partitioning. We believe that reducing the problem instance by a factor of 2 may result in an insufficient number of levels, i.e., the coarsening proceeds too quickly. A slower coarsening that results in more levels can give the refinement algorithm more opportunities to find solutions, and in addition, will reduce the differences between successively coarser netlists H_i and H_{i+1}. To control the speed of coarsening, *Match* takes a parameter $0 \le R \le 1$, called the *matching ratio*, that indicates the fraction of modules that should be matched. For example, when $R = 1$, a maximal matching is sought, but when $R = 0.5$, the matching continues only until half of the modules are matched (each remaining unmatched module is assigned to its own cluster).

Fig. 3 shows the *Match* coarsening procedure. Step 1 initializes the permutation π and the variables $nMatch, k$ and j. The while loop in Step 2 continues as long as the ratio of matched modules to the total number of modules is less than R or until all of the modules have been examined. Step 3 checks if the current module is unassigned $v_{\pi(j)}$, and if so, Step 4 adds it to the current cluster. Step 6 also adds the module w to the cluster if a matching module w can be found for $v_{\pi(j)}$ in Step 5. Step 7 increments j to consider the next module in the permutation. When Step 8 is reached, matching

```
Procedure Match
Input:      $H_i(V_i, E_i) \equiv$ Netlist hypergraph
            $R \equiv$ Matching ratio
            $\pi \equiv$ Permutation of $V_i$
Variables:  $k \equiv$ Number of clusters
            $nMatch \equiv$ Number of matched modules
            $j \equiv$ Current module index
            $w \equiv$ Matched module
Output:     $P^k \equiv$ Clustering of $H_i$
1. Construct random permutation $\pi$ of $[1..n]$.
   Set $nMatch = 0$, $k = 0$, $j = 1$.
2. while $\frac{nMatch}{|V_i|} < R$ and $j < |V_i|$ do
3.     if $v_{\pi(j)}$ is unmatched then
4.         Set $k = k + 1$. Add $v_{\pi(j)}$ to cluster $C_k$.
5.         Find unmatched $w \in V_i$ that maximizes
             $conn(v_{\pi(j)}, w)$.
6.         if such a $w$ exists then
               add $w$ to cluster $C_k$ and
               set $nMatch = nMatch + 2$.
7.     Set $j = j + 1$.
8. while $j < |V_i|$ do
9.     if $v_{\pi(j)}$ is unmatched then
           Set $k = k + 1$. Assign $v_{\pi(j)}$ to cluster $C_k$.
10.     Set $j = j + 1$.
11. return $P^k = \{C_1, C_2, \ldots, C_k$.
```

Fig. 3. Match procedure.

is complete; each remaining unmatched module is assigned to its own cluster in Steps 8–10. The final clustering obtained is returned in Step 11.

The best module w in Step 5 is found by using an array *Conn* indexed over the modules and a set S which stores the neighbors of $v_{\pi(j)}$. First, each net e incident to $v_{\pi(j)}$ is considered, and every module $w \in e$ is then visited. If w is unmatched, then $conn(v_{\pi(j)}, w)$ is computed for the net e; this value is added to $Conn[w]$, and w is added to S. After all neighboring modules of $v_{\pi(j)}$ have been visited, each module in S is considered in turn, and its connectivity is looked up in the *Conn* array. The module w that maximizes $Conn[w]$ is returned, and all of the entries in the *Conn* array are then reset to zero. This reinitialization can be done efficiently by resetting entries indexed by modules in S. Assuming constant degree bounds on the modules and that nets with more than ten modules are ignored, *Match* has linear time complexity.

B. The FM Partition Refinement Algorithm

Our refinement algorithm $FMPartition$ takes a netlist H_i and an initial partitioning solution P_i as input, and returns a refined partitioning of P_i. If the initial partitioning passed in is NULL, as in Step 5 of Fig. 2, then a random starting solution is generated. Our partitioner uses FM with a LIFO bucket scheme, and may also use CLIP [14] if desired. Since large nets can significantly slow down an iterative partitioner, $FMPartition$ ignores nets with more than 200 modules; these nets are reinserted when measuring solution quality.

Cluster size bounds can be set via the parameter r, i.e., the areas of X_i and Y_i are bounded below by $A(V_i)/2 - \max(A(v^*), r \cdot A(V_i)$ and above by $(A(V_i)/2) + \max(A(v^*), r \cdot A(V_i)$. where v^* is the module in V_i with the largest area. The solution P_{i+1} may satisfy the balance constraints for H_{i+1}, but the projected solution P_i may not satisfy the constraints for H_i (since $A(v^*)$ may decrease during uncoarsening). In this case, the solution is rebalanced by randomly moving modules from the larger cluster to the smaller one.

C. Other Implementation Details

Our code was written in C++ and compiled with g++ (v. 2.4) on a Unix platform. We utilize LEDA abstract data types (anonymous ftp to ftp.cs.uni-sb.de) for sets, queues, and doubly linked lists. We have also implemented a database which can perform numerous netlist and clustering functions and which handles the memory management of the primary data structures. The database also contains implementations for the *Project* and *Induce* subroutines.

We have also extended our multilevel code to *quadrisection*, i.e., four-way partitioning. We use the quadrisection algorithm of Sanchis [39], but without lookahead. We have implemented the sum of cluster degrees, net cut, and generic gain computations [24]; our quadrisection results are reported for the sum of degrees gain computation. To utilize our quadrisection algorithm within a placement tool, the user can preassign some modules (e.g., I/O pads) to clusters. In addition, the user has flexibility in defining terminal propagation models to partition sub-regions of the layout.

IV. Experimental Results

We ran our experiments on the 23 circuit benchmarks listed in Table I, with all CPU times are reported for a Sun Sparc 5 (85 MHz) unless indicated otherwise. We report bipartitioning results for unit module areas, allowing cluster sizes to vary 10% from exact bisection (so $r = 0.1$). The FM- and CLIP-based implementations for our ML algorithm are denoted by ML_F and ML_C, respectively. For all experiments, the coarsening threshold is set to $T = 35$ modules. We have performed the following studies.

- We compare ML to CLIP, which is a superior iterative improvement engine to FM (as seen in Table III).
- We study the effects of modifying the matching ratio parameter R, and find that slower coarsening yields more stable solution quality.
- We show that ML yields solutions with smaller cut sizes than any existing two-way partitioner.
- Finally, we show that ML yields excellent results for quadrisection, illustrating its ability to serve as the core of a top-down placement tool.

A. Comparisons with CLIP Bipartitioning

Our first set of experiments compares both the FM and CLIP variants of ML with the CLIP iterative algorithm [14]. We set the matching parameter R to 1, which forces ML to find a complete matching in the coarsening phase. Table IV reports the minimum cut, average cut, and total CPU time obtained from 100 runs of each algorithm on each test case. The results are similar for the smaller test cases in terms of the min cuts, but both implementations of ML are significantly better for circuits with more than 6000 modules. In terms of average

TABLE IV
MINIMUM CUT, AVERAGE CUT, AND TOTAL CPU TIME OBTAINED FOR 100 RUNS OF THE CLIP, ML$_F$, AND ML$_C$ ALGORITHMS

Test Case	MIN			AVG			CPU		
	CLIP	ML$_F$	ML$_C$	CLIP	ML$_F$	ML$_C$	CLIP	ML$_F$	ML$_C$
balu	27	27	27	35	35	33	26	100	110
bm1	47	47	47	63	57	55	29	93	107
primary1	47	47	47	62	56	55	30	93	106
test04	55	48	48	80	64	56	63	219	263
test03	57	56	57	74	64	61	67	258	294
test02	88	89	89	112	101	100	73	243	288
test06	60	60	60	72	77	71	65	309	354
struct	34	33	33	46	39	38	55	199	233
test05	72	75	71	72	91	83	116	386	459
19ks	110	104	106	151	114	114	144	447	510
primary2	143	139	139	215	158	156	168	414	522
s9234	45	40	41	74	50	48	237	542	582
biomed	84	86	83	109	103	92	267	909	1036
s13207	78	58	60	125	77	76	370	857	950
s15850	79	43	43	143	63	59	505	997	1126
industry2	203	168	174	342	213	197	991	2360	3015
industry3	242	243	248	406	275	274	1199	2932	3931
s35932	45	41	40	118	46	46	935	2108	2351
s38584	48	49	48	101	77	58	1363	2574	3106
avqsmall	204	139	133	340	194	182	1538	3022	3811
s38417	72	53	50	140	82	66	1423	2544	3032
avqlarge	224	144	140	352	200	183	1896	3338	4230
golem3	2276	1663	1661	3403	2026	2006	146301	48495	89800

cut sizes obtained, the results are clearer: ML$_C$ easily obtains the lowest averages, followed by ML$_F$ and CLIP. Indeed, for seven of the test cases, the *average* cut for ML$_C$ is better than the *minimum* cut obtained by CLIP. A low average cut is attractive for users who may wish to run an algorithm only a few times. The run times are higher for both versions of ML than for CLIP, with ML$_C$ using slightly more time than ML$_F$. Note that as the instance sizes increase, the ratios of ML run times to CLIP run times decrease.

B. The Matching Ratio Parameter R

Our next set of experiments varied the matching ratio parameter R: we ran ML 100 times for each test case with R values 1.0, 0.5, and 0.33. Recall that the number of levels of coarsening increases as R decreases. Tables V and VI, respectively, show how the solution quality varies as a function of R for ML$_F$ and ML$_C$.

We observe that in both tables, the minimum cuts do not vary much as R changes, except with the larger benchmarks. In both tables, the minimum cuts are significantly smaller for the largest four benchmarks (particularly golem3) for $R = 0.5$ and $R = 0.33$. Slower coarsening also reduces the average cut value, albeit with a noticeable runtime penalty. The cuts for $R = 0.5$ and $R = 0.33$ appear virtually indistinguishable, but the slower coarsening for $R = 0.33$ may start paying off for very large test cases (e.g., the averages for golem3 are 1421 for ML$_F$ and 1413 for ML$_C$ as compared to 1462 and 1465 for $R = 0.5$). This small gain does not seem to be worth the extra run time, however.

Observe that for small values of R, the differences between ML$_F$ and ML$_C$ are not nearly as pronounced as for $R = 1$. This may be because that extra levels allow an inferior iterative improvement engine extra opportunities to find a better solution. Although ML$_C$ does not yield lower minimum cuts than ML$_F$, it more consistently produces solutions with lower cuts.

In general, as R decreases toward zero, the quality of the partitioning solution should improve. This phenomenon may not necessarily hold due to randomness introduced by matching-based clustering. Of course, as R decreases, both memory and runtimes increase as well. Fig. 4 illustrates the tradeoff between R and solution quality. Results are presented for the average cut obtained by 40 runs of ML$_C$ on avqsmall and avqlarge.

C. Comparisons with Other Bipartitioning Algorithms

There are many works which present bipartitioning results for unit module areas and size constraints corresponding to $r = 0.1$. Table VII compares the cuts obtained by ML$_C$ with $R = 0.5$ for 100 and 10 runs to nine of the best and most recent algorithms in the literature. Many of these nine algorithms outperform or subsume other older algorithms, so we simply give pointers to these older works.

- GMet [1] combines an adaptation of the Metis multilevel partitioning algorithm of [27] to netlist hypergraphs with the genetic method of [20]. This algorithm is very fast since it exploits the efficiency of Metis, yet its cut sizes are somewhat inferior since it was a graph partitioning rather than a netlist hypergraph partitioning engine.
- HB is the multilevel partitioning algorithm of Hauck and Borriello [21]. They actually set the module area to be equal to its degree (for FPGA applications), yet their

TABLE V
MINIMUM CUT, AVERAGE CUT, AND TOTAL CPU TIME OBTAINED FOR 100 RUNS OF ML_F FOR DIFFERENT VALUES OF THE MATCHING RATIO R

Test Case	MIN			AVG			CPU		
	1.0	0.5	0.33	1.0	0.5	0.33	1.0	0.5	0.33
balu	27	27	27	35	32	30	100	166	234
bm1	47	47	47	57	55	55	93	166	236
primary1	47	47	47	56	54	54	93	171	231
test04	48	48	48	64	61	57	219	394	543
test03	56	58	58	64	61	61	258	543	625
test02	89	88	88	101	98	97	243	435	601
test06	60	60	60	77	68	66	309	534	732
struct	33	33	34	39	37	38	199	346	493
test05	75	72	71	91	80	79	386	696	946
19ks	104	105	105	114	118	116	447	783	1077
primary2	139	141	139	158	161	157	414	771	1089
s9234	40	40	40	50	47	47	542	939	1386
biomed	86	83	83	103	96	94	909	1604	2199
s13207	58	55	58	77	72	71	857	1472	2150
s15850	43	43	42	63	58	59	997	1793	2596
industry2	168	171	169	213	207	207	2360	4232	5885
industry3	243	243	241	275	277	275	2932	5393	7859
s35932	41	42	42	46	48	49	2108	3978	5586
s38584	49	48	47	77	56	57	2574	4530	6535
avqsmall	139	133	132	194	159	156	3022	5184	7476
s38417	53	50	50	82	72	68	2544	4649	6536
avqlarge	144	130	131	200	163	157	3338	5799	8407
golem3	1663	1348	1347	2026	1462	1421	48495	68154	99124

TABLE VI
MINIMUM CUT, AVERAGE CUT, AND TOTAL CPU TIME OBTAINED FOR 100 RUNS OF ML_C FOR DIFFERENT VALUES OF THE MATCHING RATIO R

Test Case	MIN			AVG			CPU		
	1.0	0.5	0.33	1.0	0.5	0.33	1.0	0.5	0.33
balu	27	27	27	33	29	29	110	171	234
bm1	47	47	47	55	55	54	107	177	248
primary1	47	47	47	55	54	54	106	179	243
test04	48	48	48	66	56	55	263	414	561
test03	57	56	57	61	60	60	294	469	622
test02	89	89	88	100	98	97	288	452	619
test06	60	60	60	71	65	65	354	546	720
struct	33	33	33	38	37	37	333	351	506
test05	71	71	71	83	77	76	459	735	984
19ks	106	106	105	114	114	116	510	839	1137
primary2	139	139	139	156	156	156	522	900	1234
s9234	41	40	40	48	45	45	582	968	1406
biomed	83	83	83	92	91	91	1036	1723	2300
s13207	60	55	58	76	71	68	950	1552	2183
s15850	43	44	43	59	56	57	1126	1894	2635
industry2	174	164	167	197	196	292	3016	5023	6893
industry3	248	243	244	274	276	276	3932	6670	9353
s35932	40	41	42	46	45	46	2351	4266	5921
s38584	48	47	47	58	52	52	3106	4898	6814
avqsmall	133	128	128	182	147	148	3811	6031	8228
s38417	50	49	49	66	56	56	3032	4960	6782
avqlarge	140	128	129	183	148	148	4230	6657	9276
golem3	1661	1346	1340	2006	1465	1413	89800	104828	141704

resulting bipartitionings still fall within the required size constraints even for unit areas. They report results for ten runs of HB, and show that it outperforms the flow-based algorithm of Yang and Wong [42] and spectral bipartitioning [18].

- The PARABOLI (PB) algorithm of Riess *et al.* [36] was widely considered to be the state-of-the-art partitioner in 1994, and has been the subject of numerous comparisons since [21], [42], [32], [14], [13]. The authors of [36] report cuts that are 50% better than spectral bipartitioning.

TABLE VII
CUT SIZE COMPARISONS OF ML_C (FOR 100 RUNS AND FOR TEN RUNS) WITH NINE OTHER BIPARTITIONING ALGORITHMS

Test Case	ML_C (100)	ML_C (10)	GMet	HB	PB	GFM	GFM_t	CL-$LA3_f$	CD-$LA3_f$	CL-PR_f	LSMC
balu	27	27	27		41	27	28	27	27	27	27
bm1	47	51	48					51	47	47	49
primary1	47	52	47		53	47	51	51	47	51	49
test04	48	49	49					49	48	52	69
test03	56	58	62					56	57	57	63
test02	89	92	95					91	89	87	102
test06	60	60	94					60	60	60	60
struct	33	33	33		40	41	36	33	36	33	43
test05	71	72	104					80	74	77	97
19ks	106	108	106					104	104	104	123
primary2	139	145	142		146	139	139	142	151	152	163
s9234	40	41	43	45	74	41	44	45	44	42	44
biomed	83	84	83		135	84	92	83	83	84	83
s13207	55	55	70	62	91	66	61	66	69	71	68
s15850	44	56	53	46	91	63	46	71	59	56	91
industry2	164	174	177		193	211	175	200	182	192	246
industry3	243	243	243		267	241	244	260	243	243	242
s35932	41	42	57	46	62	41	44	73	73	42	97
s38584	47	48	53	52	55	47	54	50	47	51	51
avqsmall	128	134	144		224			129	139	144	270
s38417	49	50	69		49	81	62	70	74	65	116
avqlarge	128	131	144		139			127	137	143	255
golem3	1346	1374	2111		1629						
% imprv	x		16.9	9.5	27.9	11.1	7.8	9.2	11.5	6.9	21.9
% imprv		x	8.4	3.0	20.6	6.5	3.6	6.0	7.9	5.2	19.1

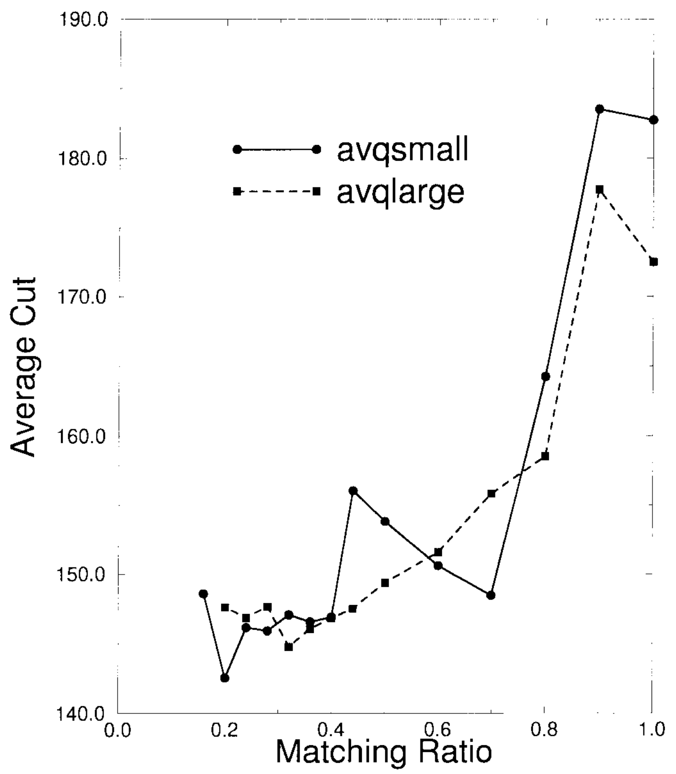


Fig. 4. Tradeoff between solution quality and the matching ratio R. Each data point represents the average cut obtained by 40 runs of ML_C.

- The GFM results are for 80 runs of the gradient Fiduccia–Mattheyses algorithm of [32], and the GFM_t results are for a single run of a "two-phase" variation of GFM.
- Dutt and Deng [14] show how CLIP and CDIP (see above discussion) can be used within any partitioner. We quote the best results for 20 runs of their three best algorithms: CL-$LA3_f$ (CLIP with lookahead level 3), CD-$LA3_f$ (CDIP with lookahead level 3) and CL-PR_f (CLIP with PROP gain calculation). The f subscript implies that standard FM was run as a refinement step after the original algorithm terminated. CL-PR_f subsumes the results for PROP reported in [13].
- Finally, we compare to the LSMC algorithm of [16] which we reimplemented. The results are reported for 100 descents, with the kick move performed on the best partitioning solution observed so far (temperature $= 0$ in the LSMC algorithm).

The last two rows of the table, respectively, give the percent improvements of ML_C with 100 runs, and ML_C with ten runs, over the other algorithms. We observe that ML_C with 100 runs averages between 7.8 and 27.9% improvement in cut sizes, yielding the best cuts ever reported for seven of the test cases. Even when limiting ML_C to just ten runs, we still obtain between 3.0 and 20.6% improvement over the other algorithms. For 100 runs of ML_C, we obtained the best known results for the benchmarks test05, s9234, s13207, s15850, industry2, avqsmall, and golem3. From Table IV, we see that the *average* cut obtained for golem3 was 1465, which is still significantly better than the best known result.

Table VIII compares the CPU times for the algorithms. We report the total time required for ten runs of ML_C on a Sun Sparc 5. The run times for GMetis, CL-$LA3_f$, CD-$LA3_f$, CL-PR_f, and LSMC are also given for this machine. PB and GFM(GFM_t) runtimes are reported for a DEC 3000 Model 500 AXP and a Sun Sparc 10, respectively. Although run times across different platforms are not directly comparable, we observe that ten runs of ML_C use less runtime than any

TABLE VIII
CPU COMPARISONS OF ML_C WITH OTHER BIPARTITIONING ALGORITHMS

Test Case	ML_C (10)	GMet	PB	GFM	GFM_t	CL-$LA3_f$	CD-$LA3_f$	CL-PR_f	LSMC
balu	17	14	16	24	25	32	31	34	41
bm1	18	12				37	47	36	43
primary1	18	12	18	16	25	36	48	37	42
test04	41	21				81	106	114	89
test03	47	23				88	107	95	92
test02	45	26				99	124	109	94
test06	55	32				50	55	175	99
struct	35	27	35	80	32	45	54	75	83
test05	74	46				141	162	188	148
19ks	84	39				178	216	219	279
primary2	90	53	137	224	61	167	210	353	176
s9234	97	58	490	672	186	175	270	264	326
biomed	172	95	711	1440	371	231	362	572	342
s13207	155	102	2060	1920	397	220	429	380	505
s15850	189	114	1731	2560	530	267	543	576	598
industry2	502	245	1367	4320	819	1129	1453	2127	944
industry3	667	299	761	4000	861	1419	1944	1920	1192
s35932	427	266	2627	10160	1088	463	964	1085	1191
s38584	490	397	6518	9680	3463	748	1339	1950	1586
avqsmall	603	328	4099			1260	2507	2082	1600
s38417	496	281	2042	11280	1062	811	1733	1690	1676
avqlarge	666	417	4135			1430	3145	2126	1742
golem3	10483	450	10823						

of the other algorithms except GMetis. It seems that if a reasonably high-quality result is desired in only a few seconds, then GMetis is appropriate; however, if a bit more CPU time can be afforded, ML_C is the better choice.

We conclude that for bipartitioning, our multilevel algorithm with a CLIP engine provides excellent cut results compared to previous algorithms while requiring a reasonable amount of CPU resources.

D. Quadrisection Comparisons

Our final set of experiments compares ML for four-way partitioning against the GORDIAN [30] standard cell placement program. In GORDIAN, the I/O pads are initially preplaced, then a system of equations is solved to find the locations of the unfixed modules such that either a squared wire-length [30] or a linear wire-length objective [41] (GORDIAN-L) is optimized. The solution to this system induces an ordering of the modules in the horizontal direction which is then split into a bipartitioning.[3] Then, another optimization induces a vertical ordering of the modules which is split to yield a four-way partitioning. The algorithm continues to perform optimization in order to spread out the cells (i.e., prevent overlapping), but this initial four-way partitioning is preserved in the final solution.

[3]GORDIAN finds a bipartitioning by finding the single split that evenly divides the area into a left and right half. GORDIAN-L uses a more complicated scheme whereby the ordering is split into five clusters and the system of equations is resolved with new constraints. The ordering induced by this second solution is then split into a bipartitioning using the same technique as GORDIAN.

TABLE IX
FOUR-WAY PARTITIONING COMPARISONS

Test Case	# Cut Nets					
	ML_F	GORDIAN	FM	CLIP	$LSMC_F$	$LSMC_C$
primary1	126 (153)	157	135	169	118	129
primary2	346 (378)	502	591	535	495	428
biomed	311 (390)	479	933	697	859	567
s13207	472 (503)	590	653	819	337	359
s15850	547 (594)	678	774	958	487	392
industry2	398 (1369)	1179	2200	1505	1695	1246
industry3	830 (1049)	1965	3005	2223	1605	1572
avqsmall	408 (505)	646	2877	1728	2098	1324
avqlarge	481 (519)	661	3131	1890	2511	1435

We obtained GORDIAN-L placement solutions [37] for some of the test cases in Table IX. For each placement solution, we split the placement into four equal-sized clusters and measured the total cut obtained. The best cut obtained by either GORDIAN or GORDIAN-L is reported in the table. We also compare to the best cut obtained for 100 runs for four-way implementations of FM, CLIP, and LSMC with both FM and CLIP partitioning engines. The first column contains min-cut results obtained by ML_F (with $R = 1.0$ and $T = 100$), with average cut sizes in parentheses. Here, ML_F outperforms ML_C in terms of cuts and run times; this may be due to CLIP being relatively ineffective at the top levels of the hierarchy. Table IX illustrates that both the minimum and average cuts obtained by ML_F are better than those obtained by GORDIAN. Our multilevel-based quadrisection algorithm has recently been integrated into a top-down hierarchical placement tool [24]. The authors of [24] report an average of 14 and 11% wire-length savings versus GORDIAN-L and GORDIAN-L + DOMINO, respectively.

V. Conclusions

We have presented a new multilevel circuit partitioner based on the paradigm of [22]. The success of our algorithm relies on exploiting new innovations in the iterative improvement engine and our ability to control the number of coarsening levels during clustering. We obtain excellent bipartitioning results compared to previous works in the literature while using less CPU time. There are several improvements that we plan to make to address the runtimes, performance and functionality of our multilevel tool.

- We plan to implement a "boundary" version of FM in which only modules incident to cut nets are initially inserted into the data structure [22]. This will significantly reduce CPU time, and may even enhance solution quality.
- Run times may be further reduced via faster reinitialization of the FM buckets at the beginning of a pass [22]. If only a few modules were moved during a pass, then only these modules and their neighbors need to be updated for the new pass. Currently, before each pass, the entire bucket structure is reinitialized.
- At the top few levels, (coarser) netlists have fewer (e.g., <500) modules so partitioning solutions can be obtained very quickly. It may be worthwhile to spend more CPU time partitioning at these levels, e.g., by calling FM multiple times or using LSMC.
- Dutt and Deng [13] showed that lookahead schemes [31] do not work very well with FM when using a LIFO bucket scheme; however, their impact increases dramatically when using CLIP. We would like to explore the use of lookahead in our iterative improvement engine even though the increases in run times may be significant.
- Finally, we have successfully integrated our quadrisection algorithm into a timing-driven placement package [24]. Our ongoing work seeks to integrate additional partitioning objectives that accommodate congestion, density, and routability considerations.

References

[1] C. J. Alpert, L. W. Hagen, and A. B. Kahng, "A hybrid multilevel/genetic approach for circuit partitioning," in *Proc. ACM/SIGDA Physical Design Workshop*, 1996, pp. 100–105.

[2] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *Integration, VLSI J.*, 1995.

[3] ——, "A general framework for vertex orderings, with applications to circuit clustering," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 240–246, June 1996.

[4] S. Areibi and A. Vannelli, "Advanced search techniques for circuit partitioning," in *DIMACS Ser. in Discrete Math. and Theoretical Comput. Sci.*, 1993, pp. 77–98.

[5] ——, "An efficient clustering technique for circuit partitioning," in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. IV, May 1996, pp. 671–674.

[6] S. T. Barnard and H. D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice Exp.*, vol. 6, pp. 101–117, Apr. 1994.

[7] T. Bui, C. Heigham, C. Jones, and T. Leighton, "Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms," in *Proc. ACM/IEEE Design Automation Conf.*, 1989, pp. 775–778.

[8] T. Bui and C. Jones, "A heuristic for reducing fill in sparse matrix factorization," in *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, 1993, pp. 445–452.

[9] T. N. Bui and B. R. Moon, "A fast and stable hybrid genetic algorithm for the ratio-cut partitioning problem on hypergraphs," in *Proceedings of the ACM/IEEE Design Automation Conference*, 1994, pp. 664–669.

[10] J. Cong and M'L. Smith, "A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design," in *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 755–760.

[11] A. Dasdan and C. Aykanat, "Improved multiple-way circuit partitioning algorithms," in *Proc. ACM/SIGDA Int. Workshop Field-Programmable Gate Arrays*, 1994.

[12] K. Doll, F. M. Johannes, and K. J. Antreich, "Iterative improvement by network flow methods," *IEEE Trans. Computer-Aided Design*, vol. 4, no. 1, pp. 1189–1199, 1994.

[13] S. Dutt and W. Deng, "A probability-based approach to VLSI circuit partitioning," in *Proc. ACM/IEEE Design Automation Conf.*, 1996, pp. 100–105.

[14] ——, "VLSI circuit partitioning by cluster-removal using iterative improvement techniques," in *IEEE/ACM Int. Conf. Computer Aided Design*, 1996, pp. 194–200. Also see corresponding Technical Report, Dep. Elect. Eng., Univ. Minnesota.

[15] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions," in *Proc. ACM/IEEE Design Automation Conf.*, 1982, pp. 175–181.

[16] A. S. Fukunaga, J.-H. Huang, and A. B. Kahng. "Large-step markov chain variants for VLSI netlist partitioning," in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. IV, May 1996, pp. 496–499, .

[17] L. Hagen and A. B. Kahng, "A new approach to effective circuit clustering," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1992, pp. 422–427.

[18] ——, "New spectral methods for ratio cut partitioning and clustering," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 9, pp. 1074–1085, 1992.

[19] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng, "On implementation choices for iterative improvement partitioning algorithms," in *Proc. European Design Automation Conf.*, 1995, pp. 144–149.

[20] L. Hagen and A. B. Kahng, "Combining problem reduction and adaptive multi-start: A new technique for superior iterative partitioning," *IEEE Trans. Computer-Aided Design*, 1996.

[21] S. Hauck and G. Borriello, "An evaluation of bipartitioning techniques," in *Proc. 16th Conf. Advanced Research in VLSI*, 1995, pp. 383–402.

[22] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proc. Supercomputing*, 1995. Also see Tech. Rep. SAND93-1301, Sandia Nat. Lab., 1993.

[23] A. G. Hoffman, "The dynamic locking heuristic–A new graph partitioning algorithm," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1994, pp. 173–176.

[24] D. J.-H. Huang and A. B. Kahng "Partitioning-based standard-cell global placement with an exact objective," in *Int. Symp. Physical Design*, 1997.

[25] H. Inayoshi and B. Manderick, "The weighted graph bi-partitioning problem: A look at GA performance," in *Parallel Problem Solving from Nature*, 1992, pp. 617–625.

[26] K. Isomoto, Y. Mimasa, S. Wakabayashi, T. Koide, and N. Yoshida, "A graph bisection algorithms based on subgraph migration," *IEICE Trans. Fundamentals*, vol. E77-A, pp. 2039–2044, Dec. 1994.

[27] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in P. Banerjee and P. Boca, Eds., *Proc. 1995 Int. Conf. Parallel Processing*, vol. 3, 1995, pp. 113–122.

[28] ——, "Parallel multilevel graph partitioning," in *Proc. 10th Int. Parallel Processing Symp.*, 1996, pp. 314–319.

[29] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.

[30] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 3, pp. 356–365, 1991.

[31] B. Krishnamurthy, "An improved min-cut algorithm for partitioning {VLSI} networks," *IEEE Trans. Comput.*, vol. C-33, pp. 438–446, May 1984.

[32] L. T. Liu, M. T. Kuo, S.-C. Huang, and C.-K. Cheng, "A gradient method on the initial partition of Fiduccia-Mattheyses algorithm," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1995, pp. 229–234.

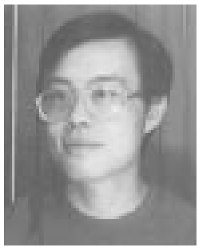
[33] T.-K. Ng, J. Oldfield, and V. Pitchumani, "Improvements of a mincut partition algorithm," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1987, pp. 479–473.

[34] C.-I. Park and Y.-B. Park, "An efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1686–1694, Nov. 1993.

[35] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox, "Graph contraction for mapping data on parallel computers: A quality-cost tradeoff," *Sci. Programming*, vol. 3, pp. 73–82, Spring 1994.

[36] B. M. Riess, K. Doll, and F. M. Johannes, "Partitioning very large circuits using analytical placement techniques," in *Proc. ACM/IEEE Design Automation Conf.*, 1994, pp. 646–651.
[37] B. M. Riess, Personal communication, Feb. 1995.
[38] Y. Saab, "A fast and robust network bisection algorithm," *IEEE Trans. Comput.*, vol. 44, pp. 903–913, July 1995.
[39] L. A. Sanchis, "Multiple-way network partitioning," *IEEE Trans. Comput.*, vol. 38, pp. 62–81, Jan. 1989.
[40] H. Shin and C. Kim, "A simple yet effective technique for partitioning," *IEEE Trans. VLSI Syst.*, vol. 1, Sept. 1993.
[41] G. Sigl, K. Doll, and F. M. Johannes, "Analytical placement: A linear or a quadratic objective function?" In *Proc. ACM/IEEE Design Automation Conf.*, 1991, pp. 427–432.
[42] H. Yang and D. F. Wong, "Efficient network flow based min-cut balanced partitioning," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1994, pp. 50–55.
[43] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin, "A probabilistic multicommodity flow solution to circuit clustering problems," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1992, pp. 428–431. Also *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 154–162, Feb. 1995, for extended version.
[44] ——, "A general purpose, multiple-way partitioning algorithm," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1480–1487, Dec. 1994.

Charles J. Alpert (S'93–M'98), for a photograph and biography, see p. 12 of the January 1998 issue of this TRANSACTIONS.



Jen-Hsin Huang (S'93–M'98) received the B.S. degree from the National Taiwan University, Taipei, Taiwan, R.O.C., the M.S. degree from State University of New York, Stony Brook, in 1991, and the Ph.D. degree from the University of California, Los Angeles, in 1996, all in computer science.

In 1991, he was a Software Developer at the IBM T. J. Watson Research Center, Yorktown Heights, NY. In 1992, he was a Research Assistant in the Department of Computer Science, University of California, Los Angeles. In the summer of 1995, he was with High Level Design Systems, Santa Clara, CA. In 1997, he was with the Design Planning Group, Synopsys, Mountain View, CA. Currently, he is with the Physical Product Division, Avant! Corporation, Fremont, CA. His research interests are in the areas of physical design on VLSI CAD, with emphasis on circuit partitioning, placement, and clock routing.

Andrew B. Kahng (A'89), for a photograph and biography, see p. 1 of the January 1998 issue of this TRANSACTIONS.