# Multilevel Circuit Partitioning[1]

Charles J. Alpert†, Jen-Hsin Huang‡ and Andrew B. Kahng∗

UCLA Computer Science Department, Los Angeles, CA 90095-1596
† IBM Austin Research Laboratory, Austin, TX 78758
‡ Synopsys, Inc., Mountain View, CA 94043
∗ Cadence Design Systems, Inc., San Jose, CA 95134

**Abstract**

Recent work [2] [5] [11] [12] [14] has illustrated the promise of *multilevel* approaches for partitioning large circuits. Multilevel partitioning recursively clusters the instance until its size is smaller than a given threshold, then unclusters the instance while applying a partitioning refinement algorithm. Our multilevel partitioner uses a new technique to control the number of levels in the matching-based clustering phase and also exploits recent innovations in classic iterative partitioning [7] [10]. 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 10 runs (while also using less CPU time).

## 1 Introduction

A netlist hypergraph $H(V,E)$ has $n$ modules $V = \{v_1, v_2, \ldots v_n\}$; a *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 $Y$, i.e., $cut(P) = |\{e \mid e \cap X \neq \emptyset, e \cap Y \neq \emptyset\}|$. 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 $\frac{A(V)(1-r)}{2} \leq A(X), A(Y) \leq \frac{A(V)(1+r)}{2}$.

The standard bipartitioning approach is iterative improvement based on the Kernighan-Lin algorithm, which was later improved by Fiduccia-Mattheyses (FM) [8]. The FM algorithm begins with some initial solution $\{X,Y\}$ and proceeds in a series of *passes*. During a pass, 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* $(= cut(\{X'-v, Y'+v\}) - cut(\{X,Y\}))$ 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 the initial solution. FM has been widely adopted 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 classic FM algorithm [7] [10] [15]. Other works attempt to use iterative improvement as an engine inside other algorithmic approaches such as large-scale Markov chains [9], two-phase clustering [4] or multilevel clustering [5] [12] [11] [14] (see [3] for a survey on partitioning and clustering techniques).

This paper proposes a new multilevel circuit partitioning algorithm that is motivated by the success of multilevel partitioners [12] [14] in the scientific computing community. We have added two key ingredients to the functionality of our partitioner, which significantly improve performance: (i) we use the CLIP algorithm of [7] within our FM implementation, and (ii) we use a matching based clustering that halts prematurely so that more than $\frac{n}{2}$ clusters are generated. This causes the multilevel coarsening to proceed more slowly, a major source of our superior solution quality.

## 2 The Multilevel Partitioning Paradigm

As problem sizes grow larger, the performance of iterative improvement approaches such as FM tend to degrade. The technique of *clustering* or *coarsening* is commonly used to deal with increasing problem sizes. The netlist modules are divided into many small clusters, and these clusters form the new nodes of a smaller, coarser netlist. Iterative improvement can then be run on this coarsened netlist. Since multilevel partitioning is based on this concept, we now present some definitions to make these ideas more rigorous.

**Definition 1:** A clustering $P^k = \{C_1, C_2, \ldots, C_k\}$ of $H_i$ *induces* the *coarser netlist* $H_{i+1}(V_{i+1}, E_{i+1})$ where $V_{i+1} = \{C_1, C_2, \ldots, C_k\}$. For every $e \in E_i$, the net $e^*$ belongs to $E_{i+1}$ where $e^* = \{C_h \mid e \cap C_h \neq \emptyset\}$, unless $|e^*| = 1$. In other words, $e^*$ spans the set of clusters containing modules of $e$.

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

Clustering has often been 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 is called a *refinement* step, which refers to the improvement of an initial good solution via local moves and swaps.

The "two-phase" approach can be extended to a *multilevel* approach by using as many phases as are desired. 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

[1]This work was supported by a grant from Cadence Design Systems and a UCLA Dissertation Year Fellowship. ABK is currently Visiting Scientist (on sabbatical leave from UCLA) at Cadence.

0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

netlist $H_m$ is constructed. 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 post-processing. The uncoarsening process continues until a refined partitioning of $H_0$ is obtained.

Multilevel partitioning offers several advantages over two-phase FM. First, the single coarsening step of two-phase FM can make $H_1$ too coarse a representation of $H_0$. Multilevel partitioning enables coarsening proceeds more slowly, giving the iterative engine more opportunities for refinement. Second, multilevel partitioning can be extremely efficient if a fast clustering and refinement strategy is used. Refinement for each netlist $H_i$ typically requires only a few FM passes to converge since it begins with a high-quality initial solution. Finally, refinement proceeds with progressively larger netlists, implying that the number of modules moved during a single step of FM becomes progressively smaller. This permits the refinement algorithm to avoid bad local minima via big steps at high levels, while still being able to find a good final solution via detailed refinement at low levels.

Work in multilevel partitioning [12] [14] [18] has been especially prominent in the scientific computing literature for partitioning finite-element graphs. Hendrickson and Leland [12] developed a very efficient multilevel partitioning algorithm, included in their Chaco package. They use random matching [4] to perform clustering and multi-way FM to do refinement, with several modifications to reduce runtime: (i) the algorithm can terminate before a pass is completed if further improvement appears unlikely; (ii) 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 (iii) 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. Metis, another multilevel partitioning package targeted to finite-element graphs, was developed by Karypis and Kumar [14]. As in [12], boundary schemes and early pass termination are used, along with many different algorithms for clustering, initial partitioning and refinement which allow experiments with various combinations of options. One of the Metis coarsening schemes uses a greedy weighted matching algorithm, upon which our coarsening scheme is based.

In the VLSI CAD community, previous multilevel works include [2] [5] [11]. The authors of [2] adapt Metis to partition netlist hypergraphs and use a genetic algorithm to obtain more stable solution quality. The authors of [5] apply clique-finding clustering as the coarsening step for multilevel bipartitioning. Finally, the authors of [11] give a detailed study of multilevel FPGA partitioning, exploring various schemes for technology mapping, clustering, partitioning the coarsest graph, and uncoarsening in one or multiple steps. Their final algorithm uses simple connectivity-based clustering and iterative improvement with two or three levels of lookahead.

## 3 A New Multilevel Algorithm

Figure 1 describes ML, our multilevel algorithm for partitioning netlist hypergraphs. The algorithm accepts a netlist $H_0$ as input along with two user parameters $T$ and $R$. $T$ sets a threshold that bounds the number of modules in the smallest netlist $H_m$, and $R$ is a parameter used by the *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 four steps in Figure 1 form the coarsening phase. As long as the number of modules in $H_i$ is greater than $T$, *Match* is used to form a clustering $P^k$ of $H_i$. Following Definition 1, procedure *Induce* takes a netlist $H_i$ and a clustering $P^k$ and constructs the

```
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
            $P^k \equiv$ Intermediate clusterings
            $P_i, 1 < i \le m \equiv$ Intermediate bipartitionings
Output:     $P_0 = \{X_0, Y_0\} \equiv$ Final bipartitioning
1. while $|V_i| > T$ do
2.     $P^k = Match(H_i, R)$.
3.     $H_{i+1}(V_{i+1}, E_{i+1}) = Induce(H_i, P^k)$.
4.     Set $i = i + 1$.
5. Let $m = i$. $P_m = FMPartition(H_m, NULL)$.
6. for $i = m - 1$ down to 0 do
7.     $P_i = Project(H_{i+1}, P_{i+1})$.
8.     $P_i = FMPartition(H_i, P_i)$.
9. return $P_0$.
```

Figure 1: The ML Multilevel Algorithm.

new netlist $H_{i+1}$ induced by $P^k$ (while preserving module areas). Step 5 constructs a bipartitioning of $H_m$ using the *FMPartition* procedure, and Steps 6-8 form the uncoarsening phase. Following Definition 2, the *Project* procedure takes a netlist $H_{i+1}$ and a bipartitioning $P_{i+1}$ of $H_{i+1}$ and constructs the projection of $P_{i+1}$ onto $P_i$ of $H_i$. 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 9. We now discuss the procedures *Match* and *FMPartition* in more detail.

We coarsen via a linear time "heavy-edge" matching similar in spirit to [14]. The *Match* algorithm starts by randomly permuting the modules and then visits each in turn. 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 $\frac{1}{|e|}$ emphasizes nets with fewer modules, and the term $\frac{1}{A(v) \cdot A(w)}$ gives preference to matching modules with smaller areas to help prevent cluster sizes from becoming too unbalanced. If such a $w$ can be found then $v$ and $w$ are matched together, i.e., they form a new cluster. If no unmatched $w$ exists (i.e., all of the neighbors of $v$ are matched), then $v$ is assigned to its own cluster. When computing the *conn* function, we ignore nets with more than ten modules to reduce runtimes.

The matching algorithms of [4] [12] [14] seek maximal matchings, which generally forces the ratio of $|V_{i+1}|$ to $|V_i|$ to be $\frac{1}{2}$. We believe that maximal matching can result in too few levels; a slower coarsening gives the refinement algorithm more opportunities to construct better solutions. Further, slower coarsening reduces the differences between successively coarser netlists $H_i$ and $H_{i+1}$ which implies that iterative refinement of $H_i$ will take fewer passes to converge. To control the speed of coarsening, *Match* takes a parameter $0 \le R \le 1$, called the *matching ratio*, specifying the fraction of modules to 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).

Figure 2 shows the *Match* coarsening procedure. 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 the modules have been examined. Step 3 adds the current module $v_{\pi(j)}$ to

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

Figure 2: The Match Procedure.

the current cluster, and Step 5 also adds the module $w$ to the cluster if a matched module $w$ can be found in Step 4. When Step 7 is reached, matching is complete. Each remaining unmatched module is assigned to its own cluster in Steps 7-9, and the final clustering obtained is returned in Step 10. Assuming constant degree bounds on the modules and size bounds on the nets, *Match* can be implemented in linear time through careful exploration of the neighbors of $v_{\pi(j)}$ in Step 4.

Our refinement algorithm *FMPartition* takes a netlist $H_i$ and an initial partitioning solution $P_i$ as input, and returns a refined partitioning of $P_i$. Since large nets can slow down FM, nets with more than 200 modules are ignored (these nets are reinserted when measuring solution quality). If the initial partitioning passed in is *NULL*, as in Step 5 of Figure 1, then a random starting solution is constructed. Our partitioner uses FM with a LIFO bucket scheme [10] and may also use CLIP [7] if desired. CLIP uses the idea of infinite weight tie-breaking, e.g., 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. The authors of [7] actually propose to increase the gain by an infinite factor and accomplish this by initializing all cells to the zero gain bucket in order of their true gains. Experiments in [7] show that CLIP averages 18% improvement over FM (both using a LIFO bucket scheme).

## 4 Experimental Results

We ran our partitioner on 23 of the standard benchmarks from the CAD Benchmarking Laboratory (ftp.cbl.ncsu.edu or visit our web site at http://vlsicad.cs.ucla.edu/). The characteristics for these test cases can be found in e.g., [2] [7]. We report bipartitioning results for unit module areas with $r = 0.1$. 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 was set to $T = 35$ modules.

Our first experiments study the effects of varying the matching ratio parameter $R$: we ran ML 100 times for each test case with $R$ values 1.0, 0.5 and 0.33. Due to space limitations, Table 1 includes only the data for the 12 largest test cases (see [1], or our website http://vlsicad.cs.ucla.edu/ , for the entire data set). For all the benchmarks, the difference among minimum cuts for various values of $R$ is less than 2%, except for the largest benchmarks. However, the minimum cuts significantly improve for the very largest benchmarks (particularly golem3) for smaller values of $R$. Over all 23 benchmarks, ML$_F$ yielded 7.9% and 9.5% respective reduction of *average* cut size for $R = 0.5$ and $R = 0.33$ over $R = 1.0$, while ML$_C$ yielded 6.9% and 7.6% lower average cuts. For $R = 1.0$, the average cuts of ML$_C$ were 5.5% lower than those of ML$_F$. We choose to use ML$_C$ with $R = 0.5$ for comparing with other algorithms since the gains of CLIP over FM and $R = 0.5$ over $R = 1.0$ are significant, while reducing $R$ to 0.33 does not seem worth the extra runtime [1].

| Alg | Test | MIN | | | AVE | | |
|---|---|---|---|---|---|---|---|
| | Case | 1.0 | 0.5 | 0.33 | 1.0 | 0.5 | 0.33 |
| ML$_C$ | s9234 | 41 | 40 | 40 | 48 | 45 | 45 |
| | biomed | 83 | 83 | 83 | 92 | 91 | 91 |
| | s13207 | 60 | 55 | 58 | 76 | 71 | 68 |
| | s15850 | 43 | 44 | 43 | 59 | 56 | 57 |
| | ind2 | 174 | 164 | 167 | 197 | 196 | 292 |
| | ind3 | 248 | 243 | 244 | 274 | 276 | 276 |
| | s35932 | 40 | 41 | 42 | 46 | 45 | 46 |
| | s38584 | 48 | 47 | 47 | 58 | 52 | 52 |
| | avqsml | 139 | 133 | 132 | 194 | 159 | 156 |
| | s38417 | 53 | 50 | 50 | 82 | 72 | 68 |
| | avqlrg | 144 | 130 | 131 | 200 | 163 | 157 |
| | golem3 | 1663 | 1348 | 1347 | 2026 | 1462 | 1421 |
| ML$_F$ | s9234 | 40 | 40 | 40 | 50 | 47 | 47 |
| | biomed | 86 | 83 | 83 | 103 | 96 | 94 |
| | s13207 | 58 | 55 | 58 | 77 | 72 | 71 |
| | s15850 | 43 | 43 | 42 | 63 | 58 | 59 |
| | ind2 | 168 | 171 | 169 | 213 | 207 | 207 |
| | ind3 | 243 | 243 | 241 | 275 | 277 | 275 |
| | s35932 | 41 | 42 | 42 | 46 | 48 | 49 |
| | s38584 | 49 | 48 | 47 | 77 | 56 | 57 |
| | avqsml | 133 | 128 | 128 | 182 | 147 | 148 |
| | s38417 | 50 | 49 | 49 | 66 | 56 | 56 |
| | avqlrg | 140 | 128 | 129 | 183 | 148 | 148 |
| | golem3 | 1661 | 1346 | 1340 | 2006 | 1465 | 1413 |

Table 1: Minimum cut, average cut and total CPU time obtained for 100 runs of ML$_C$ for different values of the matching ratio $R$.

Many works which present bipartitioning results for unit module areas and size constraints corresponding to $r = 0.1$. Table 2 compares the cuts and obtained by ML$_C$ with $R = 0.5$ for 100 and 10 runs to seven such algorithms in the literature. GM [2] and HB [11] are multilevel approaches, PARABOLI (PB) [19] uses linear placement techniques, GFM$_t$ is a two-phase gradient FM approach [16], and 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) [7] are three modifications to the FM engine. More complete comparisons with other algorithms whose results are subsumed by these works can be found in [1] or our website.

The last two rows of the table respectively give the percent improvements of ML$_C$ with 100 runs, and ML$_C$ with 10 runs, over the other algorithms in terms of cut size. ML$_C$ with 100 runs averages between 6.9% and 27.9% improvement in cut sizes. Even when limiting ML$_C$ to just 10 runs, we still obtain between 3.0% and 20.6% improvement over the other algorithms.[1]

From Table 1 we see that the *average* cut obtained for golem3 was 1465, which is still significantly better than the best known result. The table also reports the total time required for 10 runs of

[1]For 10 and 100 runs of ML$_C$, we respectively averaged 19.1% and 21.9% improvement over our implementation of LSMC [9] (22 test cases), 6.5% and 11.1% improvement over GFM [16] (13 test cases), and -1.7% and 2.4% improvement over PANZA [17] (9 test cases). Note that PANZA does not report results some of the largest benchmarks (e.g., industry2, avqsmall, avqlarge, and golem3) for which our approach has been particularly successful.

|  | Cut size |  |  |  |  |  |  |  |  | CPU Time (s) |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Test Case | $ML_C$ (100) | $ML_C$ (10) | GM | HB | PB | $GFM_t$ | CL-$LA3_f$ | CD-$LA3_f$ | CL-$PR_f$ | $ML_C$ (10) | GM | PB | $GFM_t$ | CL-$LA3_f$ | CD-$LA3_f$ | CL-$PR_f$ |
| balu | 27 | 27 | 27 |  | 41 | 28 | 27 | 27 | 27 | 17 | 14 | 16 | 25 | 32 | 31 | 34 |
| bm1 | 47 | 51 | 48 |  |  |  | 51 | 47 | 47 | 18 | 12 |  |  | 37 | 47 | 36 |
| prim1 | 47 | 52 | 47 |  | 53 | 51 | 51 | 47 | 51 | 18 | 12 | 18 | 25 | 36 | 48 | 37 |
| test04 | 48 | 49 | 49 |  |  |  | 49 | 48 | 52 | 41 | 21 |  |  | 81 | 106 | 114 |
| test03 | 56 | 58 | 62 |  |  |  | 56 | 57 | 57 | 47 | 23 |  |  | 88 | 107 | 95 |
| test02 | 89 | 92 | 95 |  |  |  | 91 | 89 | 87 | 45 | 26 |  |  | 99 | 124 | 109 |
| test06 | 60 | 60 | 94 |  |  |  | 60 | 60 | 60 | 55 | 32 |  |  | 50 | 55 | 175 |
| struct | 33 | 33 | 33 |  | 40 | 36 | 33 | 36 | 33 | 35 | 27 | 35 | 32 | 45 | 54 | 75 |
| t05 | 71 | 72 | 104 |  |  |  | 80 | 74 | 77 | 74 | 46 |  |  | 141 | 162 | 188 |
| 19ks | 106 | 108 | 106 |  |  |  | 104 | 104 | 104 | 84 | 39 |  |  | 178 | 216 | 219 |
| prim2 | 139 | 145 | 142 |  | 146 | 139 | 142 | 151 | 152 | 90 | 53 | 137 | 61 | 167 | 210 | 353 |
| s9234 | 40 | 41 | 43 | 45 | 74 | 44 | 45 | 44 | 42 | 97 | 58 | 490 | 186 | 175 | 270 | 264 |
| biomed | 83 | 84 | 83 |  | 135 | 92 | 83 | 83 | 84 | 172 | 95 | 711 | 371 | 231 | 362 | 572 |
| s13207 | 55 | 55 | 70 | 62 | 91 | 61 | 66 | 69 | 71 | 155 | 102 | 2060 | 397 | 220 | 429 | 380 |
| s15850 | 44 | 56 | 53 | 46 | 91 | 46 | 71 | 59 | 56 | 189 | 114 | 1731 | 530 | 267 | 543 | 576 |
| ind2 | 164 | 174 | 177 |  | 193 | 175 | 200 | 182 | 192 | 502 | 245 | 1367 | 819 | 1129 | 1453 | 2127 |
| ind3 | 243 | 243 | 243 |  | 267 | 244 | 260 | 243 | 243 | 667 | 299 | 761 | 861 | 1419 | 1944 | 1920 |
| s35932 | 41 | 42 | 57 | 46 | 62 | 44 | 73 | 73 | 42 | 427 | 266 | 2627 | 1088 | 463 | 964 | 1085 |
| s38584 | 47 | 48 | 53 | 52 | 55 | 54 | 50 | 47 | 51 | 490 | 397 | 6518 | 3463 | 748 | 1339 | 1950 |
| avqsm | 128 | 134 | 144 |  | 224 |  | 129 | 139 | 144 | 603 | 328 | 4099 |  | 1260 | 2507 | 2082 |
| s38417 | 49 | 50 | 69 |  | 49 | 62 | 70 | 74 | 65 | 496 | 281 | 2042 | 1062 | 811 | 1733 | 1690 |
| avqlrg | 128 | 131 | 144 |  | 139 |  | 127 | 137 | 143 | 666 | 417 | 4135 |  | 1430 | 3145 | 2126 |
| golem3 | 1346 | 1374 | 2111 |  | 1629 |  |  |  |  | 10483 | 450 | 10823 |  |  |  |  |
| %imprv | x |  | 16.9 | 9.5 | 27.9 | 7.8 | 9.2 | 11.5 | 6.9 |  |  |  |  |  |  |  |
| %imprv |  | x | 8.4 | 3.0 | 20.6 | 3.6 | 6.0 | 7.9 | 5.2 |  |  |  |  |  |  |  |

Table 2: Cut size and CPU comparisons of $ML_C$ (for 100 runs and for 10 runs) with seven other bipartitioning algorithms.

$ML_C$ on a Sun Sparc 5 (85Mhz). The runtimes for GM, CL-$LA3_f$, CD-$LA3_f$ and CL-$PR_f$ 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 runtimes across different platforms are not directly comparable, the 10 runs of $ML_C$ use less runtime than any of the other algorithms except GM. It seems that if a reasonably high quality result is desired in a few seconds, then GM is appropriate; however, if a bit more CPU time can be afforded, $ML_C$ is the better choice. Overall, our multilevel algorithm with a CLIP engine and $R = 0.5$ provides excellent cut results compared to previous algorithms while requiring only moderate CPU resources.

As the golem3 data shows, multilevel partitioning is best suited for very large instances; however, the lack of public test cases with more than 25,000 modules makes this difficult to illustrate. Indeed, the algorithms have begun to converge to the same cut size for most of the smaller benchmarks; without newer, larger test cases, it will be difficult to recognize improvement or innovation from any new partitioner.

Our current efforts seek to speed up our approach (e.g., via boundary refinement schemes and propagation of gain data down the hierarchy [12] [14]) while maintaining high solution quality. We have also integrated a 4-way partitioning version of $ML_C$ to yield an excellent quadrisection-based placement tool [13].

## REFERENCES

[1] C. J. Alpert. "Multi-way Graph and Hypergraph Partitioning" PhD Thesis, UCLA, 1996.

[2] C. J. Alpert, L. W. Hagen, and A. B. Kahng. "A Hybrid Multilevel/ Genetic Approach for Circuit Partitioning." *Physical Design Workshop*, pp. 100–105, 1996.

[3] C. J. Alpert and A. B. Kahng. "Recent Directions in Netlist Partitioning: A Survey." *Integration, the VLSI Journal*, 1995.

[4] T. Bui, C. Heigham, C. Jones, and T. Leighton. "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms." *DAC*, pp. 775–778, 1989.

[5] J. Cong and M'L. Smith. "A Parallel Bottom-Up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design." *DAC*, pp. 755–760, 1993.

[6] S. Dutt and W. Deng. "A Probability-Based Approach to VLSI Circuit Partitioning." *DAC*, pp. 100–105, 1996.

[7] S. Dutt and W. Deng. "VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques." *ICCAD*, pp. 194–200, 1996.

[8] C. M. Fiduccia and R. M. Mattheyses. "A Linear Time Heuristic for Improving Network Partitions." *DAC*, pp. 175–181, 1982.

[9] A. S. Fukunaga, J.-H. Huang, and A. B. Kahng. "Large-Step Markov Chain Variants for VLSI Netlist Partitioning." *Proc. of the IEEE Intl. Symp. on Circuits and Systems*, vol. IV, pp. 496–499, May 1996.

[10] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng. "On Implementation Choices for Iterative Improvement Partitioning Algorithms." *IEEE Trans. CAD (to appear)*, 1997.

[11] S. Hauck and G. Borriello. "An Evaluation of Bipartitioning Techniques." *Proc. of the 16th Conf. on Advanced Research in VLSI*, pp. 383–402, 1995.

[12] B. Hendrickson and R. Leland. "A Multilevel Algorithm for Partitioning Graphs." *Proc. Supercomputing*, 1995. Also see, Tech. Rep. SAND93-1301, Sandia National Laboratories, 1993.

[13] D. J.-H. Huang and A. B. Kahng "Partitioning-Based Standard-Cell Global Placement with An Exact Objective." *Proc. Intl. Symp. on Physical Design*, pp. 18–25, April 1997.

[14] G. Karypis and V. Kumar. "Multilevel Graph Partitioning Schemes." P. Banerjee and P. Boca, editors, *Proc. of the 1995 Intl. Conf. on Parallel Processing*, vol. 3, pp. 113–122, 1995.

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

[16] L. T. Liu, M. T. Kuo, S.-C. Huang, and C.-K. Cheng. "A Gradient Method on the Initial Partition of Fiduccia-Mattheyses Algorithm." *ICCAD*, pp. 229–234, 1995.

[17] J. Li, J. Lillis, and C.-K. Cheng. "Linear Decomposition Algorithm for VLSI Design Applications." *ICCAD*, pp. 223–228, 1995.

[18] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox. "Graph Contraction for Mapping Data on Parallel Computers: A Quality-Cost Tradeoff." *Scientific Programming*, **3**(1):73–82, Spring 1994.

[19] B. M. Riess, K. Doll, and F. M. Johannes. "Partitioning Very Large Circuits Using Analytical Placement Techniques." *DAC*, pp. 646–651, 1994.