Iterative Partitioning with Varying Node Weights*

Andrew E. Caldwell, Andrew B. Kahng and Igor L. Markov
UCLA Computer Science Dept., Los Angeles, CA 90095-1596 USA

{caldwell,abk,imarkov}@cs.ucla.edu.

Abstract

The balanced partitioning problem divides the nodes of a [hyper]graph into groups of approximately equal weight (i.e., satisfying balance constraints) while minimizing the number of [hyper]edges that are cut (i.e., adjacent to nodes in different groups). Classic iterative algorithms use the *pass* paradigm [24] in performing single-node moves [16, 13] to improve the initial solution. To satisfy particular balance constraints, it is usual to require that intermediate solutions satisfy the constraints. Hence, many possible moves are rejected.

Hypergraph partitioning heuristics have been traditionally proposed for and evaluated on hypergraphs with *unit* node weights only. Nevertheless, many real-world applications entail varying node weights, e.g., *VLSI circuit partitioning* where node weight typically represents cell area. Even when *multilevel* partitioning [3] is performed on unit-node-weight hypergraphs, intermediate clustered hypergraphs have varying node weights. Nothing prevents the use of conventional move-based heuristics when node weights vary, but their performance deteriorates, as shown by our analysis of partitioning results in [1].

We describe two effects that cause this deterioration and propose simple modifications of well-known algorithms to address them. Our baseline implementations achieve dramatic improvements over previously reported results (by factors of up to 25); explicitly addressing the described harmful effects provides further improvement. Overall results are superior to those of the PROP-REX$_{est}$ algorithm reported in [14], which addresses similar problems.

1 Introduction

Given a hyperedge- and node-weighted hypergraph $H = (V, E)$, a *k-way partitioning* P^k assigns the nodes in V to k disjoint nonempty partitions. The *k-way partitioning problem* seeks to minimize a given objective function $c(P^k)$ whose argument is a partitioning. A standard objective function is *net cut*, i.e., the number of hyperedges (signal nets) whose nodes are not all in a single partition. Constraints are typically imposed on the partitioning solution, and make the problem difficult. For example, limits on the total node weight in each partition (*balance constraints*) result in an NP-hard formulation [17]; certain nodes can also be fixed in particular partitions (*fixed constraints*).

*This research was supported by a grant from Cadence Design Systems, Inc.

A key driver for hypergraph partitioning research in VLSI CAD has been the top-down global placement of standard-cell designs. Key attributes of real-world instances include:

- size: number of nodes up to one million or more (all instance sizes equally important)
- sparsity: number of hyperedges very close to the number of nodes, and average node degrees typically between 3 and 5 in gate- and cell-level netlists
- average hyperedge degrees typically between 3 and 5
- small number of extremely large nets (e.g., clock, reset)
- wide variation in node weights (cell areas) due to the drive range of deep-submicron cell libraries and the presence of complex cells and large macros in the netlist
- tight balance tolerances, i.e., the sum of actual cell areas assigned to each partition must be very close (e.g., within 2%) to the requested target area.

In this application, scalability, speed and solution quality are all important criteria. To achieve speed and flexibility in addressing variant formulations, move-based heuristics are typically used, notably the Fiduccia-Mattheyses (FM) heuristic [16, 8].[1]

We note that reporting in the research literature has centered on hypergraphs with unit node weights, in particular, the original works of Kernighan and Lin [24], Fiduccia and Mattheyses [16] as well as many others evaluate new partitioning heuristics on such graphs. Prior works that address variable node weights have typically used ACM/SIGDA benchmarks [6] where hypergraph node weights vary little compared, e.g., to the size variance in modern VLSI cell libraries, and the netlist topology has relatively low node degrees (up to 10). Alpert [2] noted that many of these circuits no longer reflect the complexity of modern partitioning instances. Accordingly, the ISPD98 Circuit Benchmark Suite, consisting of 18 larger benchmarks arising in the physical implementation flow of internal IBM designs, was released in early 1998 [2, 1]. Many of the ISPD98 benchmarks have nodes with area bigger than 10% of the total and node degrees in the several hundreds; however, these instances have no large nets. By contrast, ACM/SIGDA benchmarks have only low-degree nodes with nearly uniform areas, but can have nets of degree greater than 1,000.

[1] Effective move-based heuristics for k-way hypergraph partitioning have been pioneered in [24, 16, 7], with refinements to FM given by [25, 27, 19, 26, 13, 3, 11, 18, 22, 8] and many others. Comprehensive surveys of VLSI partitioning formulations and algorithms are given in [4] [20]; a recent update on balanced partitioning in VLSI physical design is given by [21].

Akin to [14], this work addresses the differences between partitioning with varying node weights and unit node weights. Section 2 critically reviews iterative partitioning heuristics, including the popular LIFO and CLIP algorithms, and demonstrates using partitioning results published in [1] that varying node areas indeed cause performance deterioration of those heuristics. Section 3 describes a particular effect caused by heavy nodes that affects iterative partitioners, especially CLIP. The best of the proposed "fixes" to LIFO and CLIP appear to be quite effective.

In Section 4, we develop a type of temporary tolerance relaxation to counter the immobility of heavy nodes. Our technique is somewhat different than that in [14] and easier to implement. Calibration of runtimes to results reported in [14] and subsequent "best of n" tests suggest that our approach is more effective. Section 5 concludes with closing remarks.

2 Move-based partitioning

Today, competitive partitioning algorithms (e.g. [22, 3]) are overwhelmingly based on iterative heuristics [24, 16, 13] that perform single-node moves in passes in order to improve the initial solution. It is typically the case that improvements in these *classic* heuristics will also improve leading-edge heuristics. Furthermore, advances in classic heuristics often provide very immediate returns since there is a large base of users in real-world settings, as well as a more comprehensive body of results and implementations available for calibration.

2.1 Satisfying balance constraints

The need to satisfy tight balance constraints is motivated by applications in, e.g. top-down VLSI placement, where hypergraph partitioning is used to reduce large problems to smaller ones. Physical layout considerations for sub-problems translate into size/area constraints for partitioning (see [14, 9] for more details).

Turning to [1], we compare results in Table 5 for unit-weight partitioning with 2% tolerance to those in Table 6 where nodes are assigned varying (actual) weights. While lowest solution costs are comparable for both cases (e.g. 274 vs 297 for IBM01), the average performance of FM and CLIP on IBM benchmarks differs by factors of least 5-10. Moreover, comparing average cuts in the FM and CLIP columns of Table 6 against the "#Nets" column of Table 2, we see that the two iterative heuristics essentially failed on many benchmarks — more than 50% nets are cut on average in IBM02-IBM04 and IBM06-IBM13 (11 out of

18) and over 25% in several others, whereas solutions exist with only several percent of nets cut. This motivates further analysis of how balance constraints are treated in move-based partitioners.

To satisfy particular balance constraints, it is common to generate an initial solution that satisfies the constraints[2] (is "legal") and require that all intermediate solutions be legal as well. Thus moves leading to illegal solutions are rejected regardless of the gain they provide. Nodes that are heavier than the balance tolerance can never move in a typical implementation, even though such nodes often have very large degrees and the solution cost strongly depends on their assignment. Given an "unfortunate" initial assignment of several heavy nodes, a move-based partitioner is never able to recover low-cost solutions. For many instances, e.g. ISPD-98 benchmarks, heavy nodes are assigned similarly in most low-cost solutions, which means that a random assignment of heavy nodes is most likely "unfortunate".

In particular algorithms such as FM and CLIP, immobile nodes may impair the ability of other nodes to move, trapping FM- and CLIP-based iterative partitioners in high-cost local minima (this *corking effect* is described and addressed in Section 3). Such phenomena are magnified by tight balance tolerances (e.g., $\leq 2\%$) and the presence of heavy nodes, e.g., in the instances of the ISPD-98 benchmark suite (see [1, Table 2, p.81]).

2.2 The FM algorithm

As is well known, the FM heuristic [16, 8] iteratively improves an initial partitioning solution by moving nodes one by one between partitions. FM starts with a possibly random solution and applies a sequence of moves organized as *passes*. At the beginning of a pass, all nodes are free to move (*unlocked*), and each possible move is labeled with the immediate change in total cost it would cause; this is called the *gain* of the move (positive gains reduce solution cost, while negative gains increase it). Iteratively, a move with highest gain is selected and executed, and the moving node is *locked*, i.e., is not allowed to move again during that pass. Since moving a node changes the gains of adjacent nodes, after a move is executed all affected gains are updated. Selection and execution of a best-gain move, followed by gain

[2]In other words, one first solves a respective *number partitioning* problem. Since iterative partitioners typically get trapped in a number of relatively high-cost local minima, an important additional requirement is to generate randomized initial solutions with "sufficiently good" distribution in the hope to non-deterministically avoid high-cost local minima after several independent starts.

update, are repeated until every node is locked, or until no legal move is available. Then, the best solution seen during the pass is adopted as the starting solution of the next pass. The algorithm terminates when a pass fails to improve solution quality.

The FM algorithm can be easily seen [8] to have three main operations: (1) the computation of initial gain values at the beginning of a pass; (2) the retrieval of the best-gain (feasible) move; and (3) the update of all affected gain values after a move is made. The contribution of Fiduccia and Mattheyses lies in observing that circuit hypergraphs are sparse, so that the gain of any move is bounded by plus or minus the maximal node degree in the hypergraph (times the maximal edge weight, if edge weights are used). This allows hashing of moves by their gains: any update to a gain value requires constant time, yielding overall linear complexity per pass. In [16], all moves with the same gain are stored in a linked list representing a "gain bucket".

To guarantee that the output solution is *balanced*, moves that cause violations of balance constraints are typically ignored. Furthermore, in a typical implementation if the first move in a bucket is ignored, then, for CPU time considerations, the entire bucket is ignored for choosing moves (it is extremely time consuming to traverse a bucket's entire list, hoping that one of the nodes in it can be legally moved. Note that moves are examined in priority order, so the first legal move found is the best. We believe that current practice is not only motivated by speed, but is also partly a historical legacy from partitioners being tuned for unit-area, exact-bisection benchmarking. Recent work of Dutt and Theny [14] is notable for addressing the issue of partitioning with tight balance constraints, and a comparison of results is given further below. However, our techniques are orthogonal in the sense that [14] changes the structure of a pass in a sophisticated way, while we simply show how to fix a classic FM implementation in the context of tight balance constraints and uneven node weights.

2.3 The CLIP Algorithm

The *actual gain* of a node in the classic FM algorithm can be viewed as a sum of *initial gain* (i.e., the gain at the beginning of the pass) and the *updated gain* due to nodes moved. The CLIP algorithm of [13] uses *updated gain* instead of *actual gain* to prioritize moves. At the beginning of the pass, all moves have zero updated gain, and ties are broken by total (initial) gain. The authors of CLIP report very impressive experimental results [13], and CLIP has been cited as enabling within a recent multilevel partitioner implementation [3]. The method has also been the basis of such extensions as [15].

3 The Corking Effect

As noted above, CLIP begins any pass by placing all node moves into buckets corresponding to zero updated gain. The nodes with highest initial gain are placed at the heads of these zero-gain buckets. Hence, if the move at the head of each bucket at the beginning of a CLIP pass is not legal, the whole pass terminates without making any moves. Particularly when starting from a random initial solution, *the nodes with highest gain will tend to be the nodes of highest degree, which correspond to the heaviest nodes.* Furthermore, even if the first move is legal, CLIP is still vulnerable to termination soon afterward: without enough time for the moves to "spread out", nearly all moves will still be in the zero-gain bucket when it is revisited, and then ignored due to an illegal move (ending the pass). We call this the *corking effect*: the heavy node at the head of the bucket acts as a *cork*.[3]

Test Case	Generic CLIP	L-Uncorked CLIP	F-Uncorked CLIP	LF-Uncorked CLIP	Other CLIP [2]
ibm01	309/559.0(3.7)	279/547.2(3.8)	266/483.7(5.1)	299/496.0(5.4)	471/2456
ibm02	305/591.0(4.0)	266/596.0(4.3)	294/498.1(8.8)	266/486.4(8.3)	1228/12158
ibm03	1288/2683.6(6.8)	1076/2716.7(6.1)	1048/1744.8(21.5)	1019/1835.1(21.2)	2569/16695
ibm04	818/2081.6(7.2)	936/2157.3(8.0)	623/1242.8(30.7)	674/1399.0(30.0)	17782/20178
ibm05	1920/3134.4(23.7)	1814/3045.8(26.4)	1799/2988.1(25.2)	1877/3064.5(27.0)	1990/3156
ibm06	917/1677.1(11.1)	944/1728.3(10.1)	787/1431.3(23.1)	848/1324.8(24.5)	1499/18154
ibm07	1244/2993.8(15.5)	1182/3280.6(16.6)	1008/1835.3(55.6)	1136/2214.6(67.4)	14166/31326
ibm08	1494/3492.0(23.4)	1444/3242.1(28.7)	1544/3385.3(29.5)	1640/2736.4(88.1)	4283/30694
ibm09	1244/3494.3(15.5)	2326/4117.3(16.4)	1105/2087.0(81.4)	1193/2327.1(90.8)	2144/37124
ibm10	1826/3417.0(44.0)	1455/3811.2(55.2)	1594/2720.1(142.0)	1526/3062.0(146.0)	5958/46700

Table 1: Comparison of generic (Corked), L-Uncorked, F-Uncorked, and LF-Uncorked CLIP results for ISPD98 benchmark test cases. Results shown are minimum/average netcut (average CPU seconds on Sun Ultra-10) obtained over 100 independent single-start trials, with actual node weights and a 2% balance constraint. We also show the CLIP FM results reported by Alpert in [2] ("Other CLIP").

Our traces of CLIP executions show that corking occurs quite often with the more modern ISPD98 benchmarks. This is because these benchmarks contain very heavy nodes whose weight approaches or exceeds typical balance tolerances (see Table 2 in [2]). We have developed three *uncorking* techniques to counteract the corking effect.

[3] This effect is not unique to CLIP; it can apply, albeit less dramatically, to the original FM heuristic and other variants.

Explicit uncorking. Continue to look beyond the first move in a bucket, if the first move is illegal.

LIFO pass before starting CLIP. Execute a single LIFO FM pass [19] before starting CLIP passes. This greatly reduces the likelihood of large-degree nodes having the highest total gain, and corking the CLIP gain buckets. This technique should not noticeably increase CPU time as CLIP typically makes dozens of passes and an additional LIFO pass will not significantly affect runtime.

Fixing heavy nodes. At the beginning of the pass, do not place any node whose weight is greater than the balance tolerance into the gain structure. This technique has essentially zero overhead.[4]

Test Case	Generic CLIP	L-Uncorked CLIP	F-Uncorked CLIP	LF-Uncorked CLIP	Other CLIP [2]
ibm01	220/441.0(4.7)	221/401.1(4.6)	250/437.6(4.9)	223/412.8(4.8)	246/462
ibm02	257/436.0(5.7)	269/407.7(6.3)	275/419.2(7.7)	256/412.0(7.1)	439/4163
ibm03	749/1555.3(11.0)	743/1664.0(10.5)	809/1371.9(19.3)	654/1585.2(19.1)	1915/9720
ibm04	526/920.4(16.6)	510/1024.3(19.9)	479/950.3(16.9)	449/924.2(19.3)	488/1232
ibm05	1786/2849.1(25.9)	1732/2961.6(27.9)	1794/2976.3(25.3)	1774/2918.3(27.1)	2146/3016
ibm06	859/1492.4(11.6)	766/1638.8(11.6)	666/1246.9(21.2)	791/1256.9(23.3)	1303/15658
ibm07	727/1520.9(30.8)	737/1882.7(38.6)	746/1576.6(32.5)	737/1861.6(37.2)	748/1711
ibm08	1306/2283.5(52.6)	1466/2840.6(73.4)	1279/1944.7(63.7)	1492/2538.9(63.3)	2176/15907
ibm09	523/1877.5(38.7)	638/2312.0(46.1)	549/1784.8(37.2)	559/2281.2(45.8)	527/2828
ibm10	804/1907.8(64.5)	877/2040.4(71.1)	885/1945.1(63.5)	900/2214.4(77.1)	971/2242

Table 2: Comparison of generic (corked), L-Uncorked, F-Uncorked, and LF-Uncorked CLIP results for ISPD98 benchmark test cases. Results shown are minimum/average cutsize (average CPU seconds on Sun Ultra-10) obtained over 100 independent single-start trials, with actual node weights and a 10% balance constraint. We also show the CLIP FM results reported by Alpert in [2] ("Other CLIP").

We find the first technique to be too time-consuming, and it moreover appears to have a harmful effect on solution quality. Independently applying or not applying the two remaining techniques – L-Uncorking by adding an initial LIFO pass, and F-Uncorking by fixing heavy nodes – yields four different CLIP implementations: generic (corked) CLIP, L-Uncorked CLIP, F-Uncorked CLIP, and LF-Uncorked CLIP. Tables 1 and 2 show the cutsize results for these variants on ISPD98 benchmarks.[5] We report the best and average cutsize obtained

[4]With respect to the breakdown of CPU resources in Section 2, only initial gain computation may be affected — we are adding one extra `if` per node. Initial gain computation already has a number of `if`s, e.g. in net traversals and cost computations; it also entails a number of memory accesses which are much more expensive than branching.

[5]Only the first 10 test cases in the ISPD98 suite are used since runtimes for flat CLIP FM quickly become too long to be of interest in the driving context of top-down placement. I.e., faster multilevel engines would likely be necessary for the larger test cases.

over 100 independent single-start trials for each benchmark, and we also report the average CPU time (seconds on a 300MHz Sun Ultra-10 workstation with 128MB RAM) required by a single-start trial.

The experimental data clearly reveals the correlation between corking effect, early CLIP termination (small runtimes), and inferior solution quality. There are substantial performance differences between the corked and uncorked CLIP variants, and we believe that the F-Uncorked CLIP variant is the most useful in practice. We also reproduce the best and average cutsizes for CLIP, published by Alpert in [2]. Our uncorked CLIP implementation obtains stunning improvements over Alpert's CLIP implementation (up to factors of 25 reduction in average cutsize).

4 Temporary tolerance relaxation

A brief examination of the recent ISPD98 Circuit Benchmark Suite [1] reveals cells(nodes) whose area(weight) takes more than 10% of the total area(weight). Such cells are guaranteed to always be immobile during move-based partitioning with tolerance less than 10% and likely to be immobile even with larger tolerance. As explained in Section 2, this prevents move-based algorithms from achieving low-cost solutions from most initial solutions.

Temporarily relaxing partitioning tolerance in order to move otherwise immobile nodes is a natural idea; it has been explored in [14] where high-gain nodes could be moved in a pass even when this caused illegal solutions. Such *temporary illegalities* were resolved in the same pass upon reaching a certain threshold. The proposed algorithms appear difficult to tune, are far from conventional FM or CLIP and can take up to four times longer to run. A different type of temporary tolerance relaxation appears more successful and easier to implement.

4.1 Proposed metaheuristic

We perform two or more "chained" calls to a black-box iterative partitioner; every next call uses a smaller partitioning tolerance; the tolerance for the first call is large enough for every node to be movable, while the last call uses the originally requested tolerance. Solutions produced by a proceeding partitioner call are used by the next call. A solution that is illegal with respect to the smaller tolerance is "greedily legalized" before the next partitioner call. To do this, nodes are moved from overfilled and to underfilled partitions, always choosing a highest-gain move first. In practice, a separate "greedy legalization" step is unnecessary because reasonable FM and CLIP implementations, if given an illegal initial solution, automatically perform "greedy legalization" whenever necessary. A similar technique is used in the `Metis` package of Karypis et al. (and, likely, in `hMetis` [22, 23] as well), which implements *multi-level* partitioning heuristics. However, we are not aware of any works exploring it for flat partitioning.

Two implementation details are useful (but, strictly speaking, unnecessary): (a) tie-breaking on balances, and (b) the ability to limit the number of passes. If during a pass, the current solution has the best-seen cost, it will be preferred over the previous best solution if and only if it is closer to begin exactly balanced. Secondly, the last several passes in a given partitioning call often produce very little improvement. Given that the resulting solution will be processed by another partitioner call with a different tolerance, it may not be useful to wait for a non-improving pass. Therefore the number of passes may be limited; alternatively, one can require minimal improvement in a pass.

4.2 Empirical evaluation

To juxtapose the performance of the two proposed approaches to partitioning with varying node weights, we compare the best uncorking variants of LIFO and CLIP (LIFO_U and CLIP_U) described in Section 3 to their further improvements LIFO_2 and CLIP_2 with a simple-minded two-stage temporary tolerance relaxation. For the first pass, the tolerance is set to the larger of (a) three times the maximal node weight [5], and (b) 20% of the total. We also limited the number of passes in the first stage to 10 and, in CLIP_2, used CLIP only at the first partitioning stage. Appropriate experiments have suggested this particular combination from among a number of similar settings.

We analyze algorithm performance in the context of "average best of n" for $n = 1, 2, 4, 8$. This technique, advocated in [10], allows detailed analyses of run-time-versus-quality trade-offs and is also representative of important application contexts, e.g., VLSI placement. The results are presented in Table 3 and suggest that two-stage tolerance relaxation indeed improves solution costs without considerably increasing runtime.

As can be seen in 3, the LIFO_2 and CLIP_2 algorithms provide substantial improvements over even the "uncorked" LIFO and CLIP partitioners. The two-stage algorithms actually *improve* per-start runtime for some examples, and improve results given equal runtimes for nearly all the testcases. It is unclear whether $LIFO_2$ or $CLIP_2$ is the superior algorithm. This is surprising, given the clear dominance of CLIP over LIFO. The IBM04 testcase is a particularly striking example, as it shows an improvement of 25% from $CLIP_U$ (the best uncorked result) to $CLIP_2$ (the best 2-stage result) and a reduction in single-start runtime of 65%! Of the eight examples presented, IBM04 contains the largest number of nodes larger than the tolerance of 2%. Thus, it is encouraging to see that the two-stage approach addresses this difficult problem so well.

Next, we compare our two-stage temporary tolerance relaxation to PROP-REX_{est}, a leading algorithm from [14] which employs temporary illegalities within passes to address similar issues. PROP-REX_{est} is the best of the several algorithms reported in [14].[6] These experiments were performed on a 140MHz Sun Ultra-1 workstation. To calibrate our runtimes to those reported in [14], we ran our plain FM implementation on the ACM/SIGDA benchmarks [6] used in that work. The overall performance ratio of approximately 1.7 was fairly consistent, and our FM implementation produced very similar average solution costs.

The results of our comparisons to PROP-REX are given in Table 4. They suggest that four starts of our two-stage CLIP variant $CLIP_2$ achieves superior solution costs in comparable amounts of time, improving upon the performance of PROP-REX by up to 31%. A single start of $CLIP_2$ produces results similar to that of PROP-REX, while requiring much less runtime (up to 86% less for the avq_small testcase). At the same time, $CLIP_2$ is only a "fix" to CLIP and is rather simple to implement.

5 Conclusions

From analysis of partitioning results from [1], we notice that the performance of FM and CLIP partitioners deteriorates when node areas are allowed to vary. We describe two general effects that cause such performance deterioration, and that are likely to affect a wide variety of iterative partitioners. In addition, we describe the previously unknown *corking effect*, which is particularly harmful to the popular CLIP algorithm [13], notably within CLIP's motivating context of top-down standard-cell VLSI placement. We propose easy-to-implement, low-overhead techniques to counteract the latter problem, and demonstrate notable improvements in solution quality. We speculate that the CLIP corking effect was not diagnosed earlier because of the tendency to compare partitioners according to unit-area

[6]The results of PROP-REX_{est} may be found in Table 3 of [14].

Test Case	Algo	1 start	2 starts	4 starts	8 starts
IBM01	LIFO	569(4.5)	494(9.0)	451(18.0)	418(36.1)
	CLIP	505(6.4)	435(12.7)	388(25.5)	350(51.0)
	$LIFO_U$	569(4.2)	498(8.3)	455(16.6)	415(33.2)
	$CLIP_U$	440(10.1)	385(20.3)	336(40.5)	324(81.1)
	$LIFO_2$	399(4.8)	331(9.6)	266(19.3)	251(38.5)
	$CLIP_2$	401(4.9)	336(9.8)	301(19.5)	274(39.1)
IBM02	LIFO	498(6.8)	425(13.6)	386(27.1)	357(54.2)
	CLIP	600(6.6)	518(13.2)	458(26.4)	418(52.8)
	$LIFO_U$	498(5.2)	435(10.3)	388(20.6)	341(41.3)
	$CLIP_U$	480(16.3)	419(32.5)	368(65.1)	327(130.2)
	$LIFO_2$	391(10.3)	351(20.6)	321(41.1)	299(82.2)
	$CLIP_2$	426(12.5)	376(24.9)	335(49.9)	313(99.8)
IBM03	LIFO	2137(10.6)	1663(21.2)	1334(42.5)	1152(85.0)
	CLIP	2200(8.9)	1709(17.8)	1398(35.5)	1155(71.1)
	$LIFO_U$	1905(9.4)	1540(18.8)	1241(37.7)	1048(75.4)
	$CLIP_U$	1430(32.6)	1175(65.3)	1015(130.6)	933(261.1)
	$LIFO_2$	1914(14.7)	1698(29.4)	1571(58.8)	1474(117.5)
	$CLIP_2$	1646(16.3)	1452(32.6)	1330(65.3)	1257(130.5)
IBM04	LIFO	1989(11.0)	1745(22.1)	1534(44.2)	1348(88.4)
	CLIP	2241(8.8)	1920(17.6)	1685(35.2)	1510(70.4)
	$LIFO_U$	2061(9.4)	1799(18.8)	1507(37.6)	1367(75.3)
	$CLIP_U$	1432(52.1)	1227(104.3)	1048(208.6)	975(417.2)
	$LIFO_2$	1659(19.8)	1413(39.6)	1229(79.2)	1062(158.4)
	$CLIP_2$	1080(18.3)	856(36.5)	736(73.0)	651(146.0)
IBM05	LIFO	3379(33.4)	3160(66.8)	2924(133.6)	2727(267.3)
	CLIP	2953(49.2)	2746(98.4)	2568(196.9)	2355(393.7)
	$LIFO_U$	3525(32.3)	3302(64.6)	3153(129.1)	3033(258.3)
	$CLIP_U$	3085(51.3)	2819(102.6)	2615(205.2)	2358(410.3)
	$LIFO_2$	2893(39.7)	2567(79.3)	2371(158.6)	2226(317.2)
	$CLIP_2$	2635(26.1)	2387(52.3)	2212(104.6)	2045(209.2)
IBM06	LIFO	1453(16.9)	1135(33.9)	976(67.7)	891(135.4)
	CLIP	1459(16.8)	1211(33.7)	1077(67.4)	965(134.7)
	$LIFO_U$	1199(14.1)	969(28.1)	849(56.3)	784(112.6)
	$CLIP_U$	1383(45.2)	1215(90.3)	1075(180.7)	959(361.4)
	$LIFO_2$	1605(23.3)	1393(46.5)	1247(93.1)	1088(186.1)
	$CLIP_2$	1508(24.5)	1272(49.1)	1141(98.2)	1080(196.4)
IBM07	LIFO	2455(36.5)	1994(72.9)	1732(145.8)	1541(291.7)
	CLIP	2786(27.4)	2305(54.9)	1902(109.8)	1693(219.5)
	$LIFO_U$	2253(26.7)	1884(53.4)	1672(106.7)	1570(213.5)
	$CLIP_U$	1779(97.6)	1528(195.3)	1326(390.5)	1211(781.0)
	$LIFO_2$	2181(32.7)	1893(65.3)	1576(130.6)	1475(261.3)
	$CLIP_2$	1865(36.0)	1459(71.9)	1300(143.8)	1147(287.7)
IBM08	LIFO	2450(42.9)	2134(85.8)	1949(171.6)	1816(343.3)
	CLIP	3014(35.5)	2433(71.0)	2111(142.0)	1922(284.0)
	$LIFO_U$	2760(32.7)	2321(65.5)	1968(131.0)	1780(261.9)
	$CLIP_U$	2362(124.2)	2012(248.4)	1767(496.8)	1652(993.6)
	$LIFO_2$	2335(44.6)	2105(89.3)	1918(178.6)	1777(357.2)
	$CLIP_2$	2160(48.9)	1918(97.7)	1671(195.4)	1562(390.9)

Table 3: Comparison of LIFO, CLIP, uncorked LIFO ($LIFO_U$), uncorked CLIP ($CLIP_U$), two-stage LIFO ($LIFO_2$) and two-stage CLIP ($CLIP_2$) partitioning algorithms on IBM test cases. Nodes were assigned varying (actual) weights. Solutions are constrained to be within 2% of bisection (partitions must contain between 49% and 51% of total). Data expressed as (average cut / average CPU time), with CPU seconds on measured on a 140MHz Sun Ultra-1.

Test Case	PROP-REX$_{ext}$	LIFO$_2$			CLIP$_2$		
		1 Start	2 Starts	4 Starts	1 Start	2 Starts	4 Starts
avq_large	494.8(58.44)	725(10.1)	639(20.2)	570(40.4)	511(7.9)	452(15.9)	393(31.8)
avq_small	452.0(50.49)	599(17.8)	533(35.6)	488(71.3)	386(16.2)	341(32.5)	308(64.9)
biomed	134.4(13.48)	123(2.7)	111(5.5)	103(11.0)	120(2.6)	104(5.3)	95(10.4)
industry2	351.9(45.67)	504(8.5)	462(17.0)	427(34.2)	392(9.0)	339(17.9)	294(35.9)
primary1	58.5(0.89)	64(0.3)	58(0.7)	53(1.4)	63(0.3)	58(0.7)	55(1.4)
primary2	194.8(5.80)	256(1.7)	238(3.5)	221(7.1)	224(1.9)	206(3.7)	190(7.5)

Table 4: Comparison of reported CLIP-REX results with those produced by 2-stage LIFO and CLIP methods. Nodes were assigned actual cell areas. Solutions are constrained to be within 0.5% of bisection (partitions must contain between 49.75% and 50.25% of total cell area). Data expressed as average cut(average CPU time). CPU times were normalized to those reported in [14].

bisection results, and because of a reliance on older benchmarks that have only uniformly-sized cells. We also propose a simple technique of temporary tolerance relaxation, different and more successful than the best of all techniques presented in [14].

Our results suggest that prospective advances in algorithm technology should be evaluated with respect to a full range of applicable instances and contexts (i.e., use models). Furthermore, the substantial performance differences between our CLIP implementation and, e.g., that reported by Alpert [2] suggest that the partitioning research community can still benefit from improved understanding of the iterative heuristics upon which new methods are based.

Acknowledgments

Our thanks go to Professor Shantanu Dutt of the University of Illinois at Chicago for his valuable cooperation. We also thank Max Moroz of UCLA for design, implementation and support of an efficient hypergraph package used in our research.

References

[1] C. J. Alpert, "Partitioning Benchmarks for the VLSI CAD Community", Web page, `http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html`

[2] C. J. Alpert, "The ISPD98 Circuit Benchmark Suite", *Proc. ACM/IEEE International Symposium on Physical Design*, April 98, pp. 80-85. See errata at `http://vlsicad.cs.ucla.edu/~cheese/errata.html`

[3] C. J. Alpert, J.-H. Huang and A. B. Kahng, "Multilevel Circuit Partitioning", *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 530-533.

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

[5] A.E. Abbot, "The number three : its occult significance in human life", 57 p., London : Emerson Press ; Wheaton, Ill, 1992.

[6] F. Brglez, "ACM/SIGDA Design Automation Benchmarks: Catalyst or Anathema?", *IEEE Design and Test*, 10(3) (1993), pp. 87-91.

[7] T. Bui, S. Chaudhuri, T. Leighton and M. Sipser, "Graph Bisection Algorithms with Good Average Behavior", *Combinatorica* 7(2), 1987, pp. 171-191.

[8] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning", *Proc. ALENEX-99* in *Lecture Notes in Computer Science* vol. 1619, Springer, 1999.

[9] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Optimal partitioners and end-case placers", *Proc. International Symposium on Physical Design*, 1999, pp. 90-96.

[10] A. E. Caldwell, A. B. Kahng, A. A. Kennings and I. L. Markov, "Hypergraph Partitioning for VLSI CAD: Methodology for Heuristic Development, Experimentation and Reporting", *ACM/IEEE Design Automation Conference*, June, 1999.

[11] J. Cong, H. P. Li, S. K. Lim, T. Shibuya and D. Xu, "Large Scale Circuit Partitioning with Loose/Stable Net Removal and Signal Flow Based Clustering", *Proc. IEEE International Conference on Computer-Aided Design*, 1997, pp. 441-446.

[12] W. Deng, *personal communication*, July 1998.

[13] S. Dutt and W. Deng, "VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques", *Proc. IEEE International Conference on Computer-Aided Design*, 1996, pp. 194-200

[14] S. Dutt and H. Theny, "Partitioning Around Roadblocks: Tackling Constraints with Intermediate Relaxations", *Proc. IEEE International Conference on Computer-Aided Design*, 1997, pp. 350-355.

[15] C.-K. Eem and J. Chong, "An Efficient Iterative Improvement Technique for VLSI Circuit Partitioning Using Hybrid Bucket Structures", to appear in *Proc. ASP-DAC*, January 1999. See also *J. Institute of Electronics Engineers of Korea C* 35-C(3) (1998), pp. 16-23.

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

[17] M. R. Garey and D. S. Johnson, "Computers and Intractability, a Guide to the Theory of NP-completeness", W. H. Freeman and Company: New York, 1979, pp. 223

[18] S. Hauck and G. Borriello, "An Evaluation of Bipartitioning Techniques", *IEEE Transactions on Computer-Aided Design* 16(8) (1997), pp. 849-866.

[19] L. W. Hagen, D. J. Huang and A. B. Kahng, "On Implementation Choices for Iterative Improvement Partitioning Methods", *Proc. European Design Automation Conference*, 1995, pp. 144-149.

[20] F. R. Johannes, "Tutorial: Partitioning of VLSI Circuits and Systems", *Proc. ACM/IEEE Design Automation Conference*, 1996, pp. 83-87.

[21] A. B. Kahng, "Futures for Partitioning in Physical design", *Proc. IEEE/ACM International Symposium on Physical Design*, April 1998, pp. 190-193.

[22] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Domain", *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 526-529.
Additional publications and benchmark results for hMetis-1.5 or later are available at `http://www-users.cs.umn.edu/~karypis/metis/hmetis/main.html`

[23] G. Karypis and V. Kumar, "hMetis: A Hypergraph Partitioning Package Version 1.5", *user manual*, June 23, 1998.

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

[25] B. Krishnamurthy, "An Improved Min-cut Algorithm for Partitioning VLSI Networks", *IEEE Transactions on Computers* 33 (1984), pp. 438-446.

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

[27] L. Sanchis, "Multiple-way network partitioning with different cost functions", *IEEE Transactions on Computers* 42(12) (1993), pp. 1500-1504.