Improved Algorithms for Hypergraph Bipartitioning

Andrew E. Caldwell, Andrew B. Kahng and Igor L. Markov*
UCLA Computer Science Dept., Los Angeles, CA 90095-1596
{caldwell,abk,imarkov}@cs.ucla.edu

Abstract

Multilevel Fiduccia-Mattheyses (MLFM) hypergraph partitioning [3, 22, 24] is a fundamental optimization in VLSI CAD physical design. The leading implementation, `hMetis` [23], has since 1997 proved itself substantially superior in both runtime and solution quality to even very recent works (e.g., [13, 17, 25]). In this work, we present two sets of results: (i) new techniques for flat FM-based hypergraph partitioning (which is the core of multilevel implementations), and (ii) a new multilevel implementation that offers leading-edge performance.

Our new techniques for flat partitioning confirm the conjecture from [10], suggesting that *specialized partitioning heuristics* may be able to *actively exploit* fixed nodes in partitioning instances arising in the driving top-down placement context. Our FM variant is competitive with traditional FM on instances without terminals [1] and considerably superior on instances with fixed nodes (i.e., arising during top-down placement [8]).

Our multilevel FM variant avoids several complex heuristics and non-trivial tunings that often lead to complex implementations; it achieves trade-offs between solution quality and run time that are comparable or better than those achieved by `hMetis-1.5.3` (the latest available version). Following [6], we attempt to provide algorithm descriptions that are as detailed and unambiguous as possible, to allow replicability and speed improvements in future research.

1 Introduction

Hypergraph partitioning is important to many application domains including data mining, job scheduling, hardware-software partitioning, VLSI circuit layout and numerical linear algebra. Balanced partitioning typically represents the "divide" step of "divide-and-conquer" algorithms and seeks to assign the nodes of a [hyper]graph into groups of approximately equal total weight (i.e., satisfying balance constraints) while minimizing the number of [hyper]edges that are cut (i.e., adjacent to nodes in different groups).

In some applications (notably in top-down VLSI circuit placement), problem instances have fixed nodes. These instances are easier than the "free hypergraph" instances that have dominated VLSI partitioning research in the past [10]. Therefore, in addition to the ISPD-98 "free hypergraph" suite released by IBM [1], similar instances with fixed nodes were released at ISPD-99 [8]. [10] suggested that *specialized partitioning heuristics* may be able to *actively exploit* fixed nodes, in contrast to common FM variants which simply *tolerate* fixed nodes. In our first set of contributions we demonstrate such a technique with the additional property of not incurring significant overhead when there are no terminals (cf. [10]).

Our second set of contributions is in multilevel partitioning. This algorithm framework entails a clustering of the original hypergraph so that clusters can be partitioned, after which the clustered partitioning solution is refined in many steps [22, 24]. The technique is used, e.g., by the leading hypergraph partitioning tool for large hypergraphs, `hMetis` [23]. `hMetis` has been successfully applied to VLSI circuits, in data mining and in numerical analysis. However, some of the algorithms critical to the performance of `hMetis` require non-trivial tunings and have not been replicated in other VLSI partitioning works; these include *v-cycling*, *V-cycling* and *hyperedge removal*. Even recent partitioner implementations have notable solution quality and/or runtime deficiencies relative to `hMetis` (e.g., [13, 25, 17]). Typically, multilevel partitioner implementations used in VLSI placement [7, 20] have employed much simpler techniques. A lack of documented key implementation details in the literature (cf. [9]), and the implementation complexity of `hMetis` techniques, may be factors contributing to the lack of integration of `hMetis`-quality partitioning methods in the VLSI community.

Our proposed algorithms match results of `hMetis-1.5.3` (the latest available as of July 1999) and can be implemented without lengthy fine-tuning; our main contributions are a new simple clustering algorithm and a new simple technique for managing tight balance constraints.

In what follows, Section 2 reviews previous work on "flat" partitioning, introduces our new methods for exploiting fixed terminals and presents relevant experimental data. Section 3 reviews the multilevel partitioning framework, describes a baseline implementation, introduces our new techniques and validates them empirically. Conclusions are given in Section 4.

2 Techniques for partitioning with terminals

2.1 Previous work

Most modern hypergraph partitioning heuristics are based on the iterative Fiduccia-Mattheyses(FM) algorithm [18] whose neighborhood structure is induced by single-node,

*Supported by a grant from Cadence Design Systems, Inc.

partition-to-partition moves.[1] FM starts with a possibly random solution and changes the solution by a sequence of moves which are 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 can change 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 update, are repeated until every node is locked. 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.

To satisfy particular balance constraints, it is common to generate an initial solution that satisfies the constraints (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. When the partitioning tolerance is small, many moves are rejected and solution quality decreases. The work of [16] studied the performance of the FM algorithm in such circumstances and showed how to improve average-case solution quality at the cost of significantly increased run time.

The FM algorithm naturally accommodates nodes that are fixed in certain partitions and cannot move. A back-of-the envelope computation in [10] using Rent's rule shows that partitioning instances arising in top-down placement typically have many fixed nodes. [10] demonstrated nonlinear effects of fixed vertices on the performance of the FM algorithm and called for *specialized heuristics* that would actively exploit fixed nodes rather than just tolerate them. One particular challenge is that such methods must not be inferior to FM when there are few or no terminals, since there is presently no easy way to quantify the effects of fixed nodes before partitioning.[2] Partitioning benchmarks representative of the top-down placement context have been released at ISPD-99 [8].

2.2 New techniques

Our proposed techniques combine four easy modifications to the FM algorithm and common implementation practices.

- Using an (illegal) initial solution that puts all movable nodes into one partition — the VILE ("very illegal") initial solution generator.
- Relaxing the acceptance criterion for legal moves — a move is accepted if and only if it does not increase the violation of balance constraints (rather than necessarily result in a legal solution, cf. [16]).
- Randomization during gain computation at the beginning of the pass. This is achieved by computing gains of legal moves in a random order.

[1] A comprehensive survey of partitioning formulations and algorithms, centered on VLSI applications is given in [4]. Implementation trade-offs in the classic FM algorithm are discussed, e.g., in [6, 21].

[2] All fixed nodes in a given partition can be collapsed into one node, and the complete spectrum of effects reported in [10] can be replicated with just two fixed nodes.

Algo	1 start	2 starts	4 starts	8 starts
IBM01D_H 2%, 6139 movable, 2155 fixed, 7330 nets				
FM	706.6(1.7)	648.8(3.4)	600.5(6.7)	566.3(13.4)
CLIP	654.7(5.4)	593.6(10.8)	**537.2(21.7)**	**492.0(43.4)**
VRW	**624.2(1.7)**	**588.6(3.4)**	**570.6(6.7)**	**555.5(13.4)**
IBM01D_H 10%				
FM	676.9(1.6)	612.3(3.2)	562.8(6.5)	504.2(12.9)
CLIP	482.4(6.0)	412.0(12.1)	**368.3(24.2)**	**348.3(48.4)**
VRW	**419.9(1.4)**	**403.8(2.9)**	**395.9(5.8)**	**390.4(11.6)**
IBM01D_V 2%				
FM	664.3(2.1)	600.0(4.2)	557.8(8.3)	521.9(16.6)
CLIP	587.7(5.5)	536.8(11.1)	499.0(22.2)	**471.2(44.4)**
VRW	**535.6(1.8)**	**508.3(3.7)**	**489.4(7.4)**	**480.1(14.8)**
IBM01D_V 10%				
FM	623.9(1.8)	582.2(3.51)	551.3(7.0)	528.9(14.1)
CLIP	559.4(5.6)	521.3(11.2)	488.3(22.4)	465.2(44.8)
VRW	**519.8(1.4)**	**494.6(2.8)**	**474.6(5.5)**	**461.9(11.1)**
IBM06D_H 2%, 10314 movable, 7553 fixed, 12438 nets				
FM	**1386(1.8)**	1120(3.5)	900.8(7.0)	822.0(14.0)
CLIP	1354(14.9)	1074(29.8)	853.9(59.7)	**777.2(119)**
VRW	**817.8(2.5)**	**799.7(5.1)**	**791.2(10.1)**	**785.0(20.3)**
IBM06D_H 10%				
FM	**1392(2.1)**	1122(4.2)	911(8.3)	823(16.5)
CLIP	1344(17.2)	1057(34.4)	842(68.7)	770(137.5)
VRW	**781(2.3)**	**772.8(4.7)**	**766(9.3)**	**764(18.7)**
IBM06D_V 2%				
FM	**1715(1.9)**	**1577(3.7)**	**1450(7.4)**	1388(14.9)
CLIP	1679(15.7)	1519(31.3)	1382(62.8)	1301(125)
VRW	**1639(2.3)**	**1478(4.8)**	**1333(9.4)**	**1274(18.7)**
IBM06D_V 10%				
FM	**1707(1.8)**	1562(3.7)	1417(7.4)	1332(14.7)
CLIP	1697(17.0)	1531(34.0)	1376(68.1)	1274(136.1)
VRW	**1559(2.0)**	**1384(4.1)**	**1194(8.2)**	**1150(16.4)**

Table 1: FM and CLIP compared to our VRW algorithm on `IBM01D` and `IBM06D` series from the ISPD-99 benchmark suite. Non-dominated configurations and winning algorithms are boldfaced.

- Preferential placement at the heads of gain buckets of nodes adjacent to fixed nodes. This is accomplished by *moving* the fixed nodes back and forth (*"wiggling"*) after the initial gain computation at the beginning of a pass in conjunction with the use of now common LIFO FM implementation [21].

We know from [16] that partitioning quality may degrade when all intermediate solutions must be legal. That is because large nodes become immobile, while often having high associated gains (large cells in VLSI circuits tend to have many inputs). Using the VILE initial solution generator together with a relaxed move acceptance criterion makes all nodes mobile at the first pass, particularly, this quickly results in favorable assignments of high-gain nodes. Our FM variant typically reaches a legal solution in the first pass.[3] VILE removes all randomization from the traditional FM algorithm and, while the results are good on average (over many circuits), the produced solutions are sometimes very poor. Randomized initial gain computation at the first pass improves upon that and, if applied at every pass, may also help escape local minima (even when applied to the traditional FM).[4]

[3] The best solution in a pass is determined as either the best legal solution or (if no legal solutions exist) an illegal solution with the smallest cost among solutions with the smallest violation.

[4] A similar random re-indexing procedure can be found in the source code of the multilevel graph partitioner `metis-3.0` where it is performed between levels. Also, somewhat related is the methodology of [5] that ensures statistical significance of experiments through

Finally, we expect that a good partitioning solution assigns an average movable node adjacent to a fixed node to the same partition as the fixed node. By "wiggling" fixed nodes, we encourage preferential movement of their adjacent nodes. The sooner all such assignments are implemented, the faster FM will converge to a good solution. We found "wiggling" helpful only at the first pass, but it provides strong improvement when used with the other three modifications.

We denote the combination of the above techniques by the abbreviation **VRW** (VILE + randomization + "wiggling" fixed nodes).

2.3 Empirical validation

We use several ISPD-99 benchmarks [8] (movable nodes have non-unit weights), which reflect the specifics of fixed nodes in top-down placement. [5] recommends the use of several similar variants of each benchmark along with a reasonable pool of different benchmarks to ensure statistical significance and replicability of results. We also follow a reporting methodology proposed in [9] that allows tracking of *non-dominated* configurations (the dominance relation is defined as simultaneous superiority in average CPU time and solution quality).

Table 1 presents "average best of" 1, 2, 4 and 8 starts. Averaging is performed over a sample of 100 independent starts and average CPU time on a Sun Ultra-1/140MHz is given in parenthesis. Results are representative of results on other benchmarks from the ISPD-99 suite. Non-dominated configurations are boldfaced and typically belong to VRW. In other words, given the same amount of CPU time, VRW finds better solutions on average than CLIP or FM; it also finds solutions with similar cost faster. Several FM configurations are non-dominated due to small CPU time used, while their solution quality is rather poor (e.g., for `IBM06D_H` and `IBM06D_V`). Several CLIP configurations are non-dominated due to marginally better solution quality, but use disproportionately large amounts of CPU time (e.g., for `IBM01D_V 2%` and `IBM06D_H 2%`). These configurations are not practical and are known side-effects of the reporting methodology.[5]

Results were somewhat unusual on the `IBM09D` series of benchmarks. One average start of VRW (approx. 6.5 sec) produced better solutions than eight starts of CLIP (approx. 170 sec) or FM (approx. 4.5 sec); the improvement in quality provided by eight starts of any partitioner compared to one was on the order of 0.1%. CLIP and FM produced almost identical costs, while VRW was at least 2/3% better and only 1/2% away from the "best seen" solution reported in [8]. One start of `hMetis-1.5.3` took 27-47 sec depending on the configuration.

We explain such results by a very large number of fixed nodes in the `IBM09D` series of benchmarks and nets incident to them.

Results for the `IBM01A` and `IBM06A` series from the ISPD-99 suite are not as strong because they have very few fixed nodes. While VRW confidently dominates on `IBM01A` instances, it typically loses to both CLIP and FM on `IBM06A` instances.

1,2,4 and 8 starts of VRW performed generally well on "free hypergraph" benchmarks from the ISPD-98 suite (see [1] and Table 2). However, the results are somewhat erratic, e.g., all configurations of VRW on IBM03 10% are non-dominated, VRW is twice as fast and twice as good as CLIP and FM on IBM05 10%. It owns approximately half of the non-dominated configurations on IBM06 10% and loses to LIFO on IBM06 2% by only a small margin (both on speed and quality). In general, we can recommend VRW on free hypergraphs when the number of starts is limited to two or four (e.g., in top-down circuit placement algorithms). This is because the relative lack of randomization in VRW (to due having the same initial solution at each start) limits the impact of multiple starts.

Classic FM and CLIP methods can be combined with our proposed VRW to achieve more robust performance, e.g., by running starts of several methods and taking the best. One can also establish *time-out* for slower starts *relative* to faster starts in order to address instances where VRW is much faster than FM and/or CLIP.

3 Techniques for multilevel partitioning

The multilevel hypergraph partitioning framework was successfully verified in 1997 by [3, 22] and has been conducive to the best known known partitioning results ever since. It consists of three main components: *clustering*, *top-level partitioning* and *refinement* or uncoarsening. During clustering, hypergraph nodes are combined into clusters based on the connectivity, leading to a smaller, *clustered hypergraph*. This step is repeated until there are only several hundred clusters, culminating in a hierarchy of clustered hypergraphs. The smallest (top-level) hypergraph is partitioned, e.g., using the FM algorithm, and the resulting solutions can be interpreted as solutions for the next hypergraph in the hierarchy. During the refinement stage, solutions are projected from one level to the next and iteratively improved, e.g., by the FM algorithm.

Additionally, the `hMetis` partitioning program [23] introduced several new heuristics that are incorporated into their multilevel partitioning implementation and are reportedly performance-critical. One is *hyperedge removal* during refinement, which is analogous to FM, except that a single move "uncuts" a hyperedge by reassigning as many nodes as needed. Another heuristic is *V-cycling*, a repetition of the clustering-partitioning-refinement process that uses a solution produced by a previous execution of this process — nodes in different partitions cannot be clustered. A similar technique employed is *v-cycling*, in which the refinement stage may stop before the lowest-level hypergraph is reached and clustering resumed (starting from a solution for a clustered hypergraph). Similarly, clustering may be stopped earlier than it would normally be, and refinement resumed.

According to [22] and more detailed technical reports, the above heuristics are performance-critical, but require non-trivial fine-tuning. From our experience, implementing a number of other aspects of the `hMetis` algorithm requires careful experimentation, as they are highly interdependent. Finally, we are not aware of a detailed experimental account of improvements since `hMetis 1.0` and up to `hMetis 1.5.3`. This motivates our search for an MLFM variant that is easy to describe and implement, yet competitive with `hMetis`.

3.1 A baseline implementation

In our baseline implementation, we employ the linear-time clustering strategy EC (edge coarsening) proposed in

multiple repetitions with hypergraph nodes randomly permuted.

[5] In particular, more starts of VRW are likely to dominate these CLIP configurations.

[3, 22]. Our baseline EC implementation has the following attributes:

- the netlist is updated continuously as the clustering occurs, i.e., the next pair of merged clusters is selected with the knowledge of the last merged pair.
- no cluster can be merged with another if its weight is more than 4.5 times the average cluster weight at the current level.
- edge weights are additionally divided by the square root of the sum of cluster weights in order to discourage merging large clusters
- clustering ratio used is 1.3, unclustering ratio is 2.5.[6]
- clustering stops the clustered hypergraph has 200 clusters or fewer.

Top-level partitioning requires initial solution generation. Unlike [22], we use one rather simple generator, "randomized engineering method". It assigns nodes to partitions in decreasing order of size using biased random selection ("spinning a roulette wheel" such that each outcome has a prescribed probability). Until all partitions reach minimal required amount of cell area, assignment probabilities are proportional to hypothetical area slacks *after* assigning a given cell to partitions. This keeps slacks approximately equal, yet provides a good degree of randomness. Once all partitions reach their minimal required cell area, slacks are computed relative to the maximal allowed areas.

Top-level partitioning is performed by CLIP-FM [15] with the tolerance that is requested for the original partitioning problem, and the best of three independent starts is further refined (on the less-clustered hypergraph) by LIFO-FM [21]. The slower CLIP-FM produces better solutions, while much faster LIFO-FM ensures good trade-off between solution quality and runtime.[7]

The configuration that we describe provides good speed and reasonable, though not leading-edge, solution quality. We next present a sequence of new techniques that can be applied to this "vanilla" multilevel partitioner to meet or exceed the best previously reported results.

3.2 New techniques

Even when the nodes of original ("flat") hypergraph have similar weights, cluster weights may considerably differ. Even in flat hypergraphs, heaviest nodes may reach over 10% of the total weight (e.g., in the IBM02 benchmark in [1]). Therefore, top-level partitioning with small tolerance is difficult if all intermediate solutions are required to be legal [16]. While it is possible to enforce stricter limits on the size of newly produced clusters, this will prevent tightly connected clusters from merging and adversely affect clustering quality.

To address this problem, we perform two partitioning calls at the top level — the first partitioning artificially increases the original tolerance to twice the largest node weight if the latter is greater, ensuring mobility of all nodes. The resulting partitioning is used as the initial solution to a second partitioning operation performed with the original tolerance.

Our second improvement is a modification to the EC clustering algorithm, which computes the weight of two potentially matched clusters as a summation over shared hyperedges $\sum_e \frac{1}{deg(e)}$. Our new clustering algorithm, *PinEC*, changes the contribution of hyperedges of degree two to two and the contribution of larger hyperedges to one. This corresponds to the number of pins[8] that would be removed from the hyperedge if the two nodes are clustered. We further divide edge weights by the sum of cluster weights to discourage merging large clusters. For a comparison, the latest paper [24] by the authors of `hMetis` mentions three very different clustering algorithms — *EC*[3, 22, 25, 12] (edge coarsening, similar to EC), *HEC* ("hyperedge coarsening") and *FirstChoice* [19]. Combining all of them in `hMetis` further lifts the barrier to the replication of `hMetis` performance.

Test	# nodes	# hyperedges
ibm01	12506	14111
ibm02	19342	19584
ibm03	22853	27401
ibm04	27220	31970
ibm05	28146	28446
ibm06	32332	34826
ibm07	45639	48117
ibm08	51023	50513
ibm09	53110	60902
ibm10	68685	75196
ibm11	70152	81454
ibm12	70439	77240
ibm13	83709	99666
ibm14	147088	152777
ibm15	161187	186608
ibm16	182980	190048
ibm17	184752	189581
ibm18	210341	201920

Table 2: ISPD-98 benchmarks used in our experiments.

We apply a *single V-cycle* to the best of several complete solutions produced by the multilevel partitioner. For this V-cycle, we use the clustering algorithm from the baseline implementation. We have observed better results when applying different clustering algorithms during V-cycling and the main clustering/top-level partitioning/refinement procedure and V-cycling.[9] Notably, `hMetis` can perform multiple V-cycles, which requires additional fine-tuning of "convergence criteria" to the rest of the implementation.

Further improving a solution that is "best of several starts" results in starts which are not independent, i.e., four such starts will take less time than four starts performed by different instances of the executable. Since V-cycling is a relatively cheap operation (it requires less time than a partitioning which begins with a random solution),

[6] Average numbers of children of a cluster during clustering and refinement stages. See [3, 22] for additional discussions of these parameters.

[7] In particular, the top-level partitioning is critical to the overall solution quality, while lower, less clustered levels, take longer to partition. Using the slower and more effective CLIP-FM technique on only the top-level takes maximum advantage of these effects.

[8] A "pin" is a connection between a node and a hyperedge.

[9] Note that during V-cycling only a single top-level partitioning is performed. V-cycling refines an existing solution, and FM-based partitioners are deterministic given an initial solution, making multiple starts useless. In all other aspects, the V-cycling refinement stage is exactly the same as the main clustering/top-level partitioning/refinement procedure.

Test ibm#	Algo	10% configurations			
		1	2	3	4
01	hM	255(5.3)	252(7.3)	247(11)	237(18)
	OUR	**238(4.6)**	**233(6.9)**	**223(12)**	**220(21)**
02	hM	**279(11)**	279(14)	279(19)	278(31)
	OUR	**291(8.2)**	291(13)	**272(21)**	**268(37)**
03	hM	**779(15)**	**776(18)**	**768(27)**	759(43)
	OUR	**802(11)**	**783(16)**	**732(30)**	**705(52)**
04	hM	**506(19)**	510(20)	**492(29)**	**478(51)**
	OUR	**535(12)**	512(19)	**482(32)**	**468(56)**
05	hM	**1759(25)**	1740(28)	1731(42)	**1725(67)**
	OUR	**1773(17)**	**1726(26)**	1728(43)	**1716(77)**
06	**hM**	**402(24)**	**383(26)**	**374(35)**	**370(55)**
	OUR	**465(14)**	433(22)	394(34)	375(61)
07	hM	809(40)	806(46)	796(63)	764(102)
	OUR	**790(20)**	**786(32)**	**760(52)**	**747(93)**
08	**hM**	**1166(43)**	**1162(51)**	**1161(75)**	**1159(123)**
	OUR	**1348(25)**	1330(40)	1195(66)	1168(113)
09	hM	664(33)	**540(42)**	**528(59)**	**525(96)**
	OUR	**572(21)**	**560(34)**	556(55)	**527(90)**
10	**hM**	**842(57)**	**830(72)**	**798(102)**	**779(169)**
	OUR	**1196(35)**	1132(55)	1023(91)	938(161)
11	**hM**	828(59)	**744(71)**	**717(103)**	**710(155)**
	OUR	**788(30)**	**779(48)**	745(77)	728(124)
12	**hM**	2326(83)	**2165(108)**	**2135(153)**	**2047(238)**
	OUR	**2349(38)**	**2304(57)**	**2261(102)**	2206(176)
13	hM	**1045(97)**	**999(104)**	**940(129)**	**902(197)**
	OUR	**1095(38)**	**1058(61)**	**1027(99)**	963(176)
14	hM	1894(212)	1776(250)	1682(351)	1599(565)
	OUR	**1759(67)**	**1727(104)**	**1638(167)**	**1590(300)**
15	hM	2073(264)	2032(301)	**1910(402)**	**1866(564)**
	OUR	**2244(83)**	**2029(133)**	2035(221)	**1975(385)**
16	hM	1942(232)	1754(311)	1829(327)	1721(709)
	OUR	**1976(102)**	**1915(155)**	**1752(257)**	**1714(460)**
17	hM	2417(408)	2372(467)	2337(635)	2311(987)
	OUR	**2316(105)**	**2294(170)**	**2264(290)**	**2239(385)**
18	hM	**1632(306)**	**1617(357)**	**1561(601)**	**1536(836)**
	OUR	**1976(104)**	**1955(160)**	1666(365)	1595(612)

Table 3: Comparison of our partitioner and hMetis1.5.3 on instances with *actual* cell areas. Solutions are constrained to be within 10% of bisection (partitions must contain between 45% and 55% of total cell area). Average CPU time in seconds is given in parenthesis. Non-dominated configurations and winning implementations are boldfaced.

Test ibm#	Algo	2% configurations			
		1	2	3	4
01	hM	267(5.2)	265(7.2)	253(11)	245(19)
	OUR	**250(4.4)**	**238(6.7)**	**231(11)**	**227(20)**
02	**hM**	**320(10)**	**314(14)**	**302(20)**	**299(33)**
	OUR	**348(8.0)**	335(12)	313(22)	**294(40)**
03	hM	885(16)	**869(20)**	859(28)	855(45)
	OUR	**903(10)**	**883(15)**	**847(27)**	**818(48)**
04	**hM**	**550(13)**	**543(18)**	**535(28)**	**534(45)**
	OUR	**592(12)**	575(18)	546(34)	**531(60)**
05	**hM**	**1777(22)**	**1749(27)**	**1744(41)**	**1741(69)**
	OUR	**1841(17)**	1810(26)	1759(44)	1750(79)
06	hM	728(24)	679(30)	637(41)	605(65)
	OUR	**696(14)**	**664(22)**	**633(37)**	**564(65)**
07	hM	855(42.2)	859(54.5)	824(63.5)	794(101)
	OUR	**846(20)**	**840(31)**	**812(53)**	**793(94)**
08	hM	**1246(52)**	**1216(57)**	**1211(74)**	1208(130)
	OUR	**1354(25)**	**1342(39)**	1238(65)	**1206(112)**
09	hM	591 (33)	**530 (42)**	**527 (62)**	**524 (98)**
	OUR	**555(22)**	**534(35)**	**528(56)**	527(91)
10	hM	**1310(78)**	**1273(97)**	**1215(126)**	**1193(192)**
	OUR	**1419(33)**	**1397(53)**	1322(90)	**1211(157)**
11	hM	914(67)	**883(77)**	**845(100)**	**813(150)**
	OUR	**926(32)**	**908(50)**	**862(79)**	**842(136)**
12	**hM**	**2304(99)**	**2180(126)**	**2150(154)**	**2131(241)**
	OUR	**2676(32)**	**2578(55)**	**2498(88)**	2353(153)
13	**hM**	**1110(103)**	**1009(107)**	**956(134)**	**931(208)**
	OUR	**1247(41)**	**1200(63)**	1140(103)	1036(180)
14	hM	2092(211)	1992(258)	1910(369)	1865(607)
	OUR	**2043(80)**	**2035(121)**	**1917(205)**	**1860(316)**
15	hM	2435(270)	2418(323)	2366(399)	**2221(597)**
	OUR	**2486(85)**	**2464(138)**	**2378(225)**	**2243(397)**
16	**hM**	2165(292)	**1829(327)**	**1732(451)**	**1713(695)**
	OUR	**2040(104)**	**1983(163)**	**1869(263)**	1853(455)
17	hM	2610(426)	2521(471)	2491(645)	2460(996)
	OUR	**2437(113)**	**2413(173)**	**2382(303)**	**2353(544)**
18	hM	**1833(334)**	1836(439)	1754(640)	**1706(1064)**
	OUR	**2002(128)**	**1976(190)**	**1823(399)**	**1737(612)**

Table 4: Comparison of our partitioner and hMetis1.5.3 on instances with *actual* cell areas. Solutions are constrained to be within 2% of bisection (partitions must contain between 49% and 51% of total cell area). Average CPU time in seconds is given in parenthesis. Non-dominated configurations and winning implementations are boldfaced.

the runtime statistics of our starts are not very different from independent starts. `hMetis` uses an additional "pruning" heuristic, that may stop the refining stage half-way through because the current solution was deemed unpromising. Judging from our experiments (below), this heuristic is fine-tuned in a rather aggressive way — on some benchmarks `hMetis`' "four starts" take only twice longer than "one start". The V-cycling and pruning techniques both result in runtime/solution-quality profiles that can not be sampled. That is, the expected result for, say, 4-starts of either partitioner can not be determined by sampling a pool of 2-start results. For this reason each entry in tables 3 and 4 is the average of at least 50 independent runs with the given configuration. This accounts for the occasional non-monotonicity of the results.

3.3 Empirical validation

To demonstrate the effectiveness of our multilevel partitioner Tables 3 and 4 present a comparison with `hMetis` on a set of standard benchmarks [1] derived from VLSI circuits at IBM. These instances, presented first at ISPD-98, range from 12506 nodes in IBM01 to more than two hundred thousand nodes in IBM18 (see Table 2). Our reporting style is that proposed in [9] and emphasizes the trade-off between solution quality and speed rather then just average solution quality. For `hMetis`, configurations 1 through 4 represent the average best produced in 1, 2, 4 and 8 starts, respectively. Default `hMetis` configurations were used. For our partitioner, configuration 1 contains the average best of 1 start, and configurations 2 to 4 the average best of 1, 2 or 4 starts followed by a single V-cycle on the best solution.

As can be seen in Tables 3 and 4, for an equivalent amount of runtime our partitioner produces comparable or better results than `hMetis` overall. In particular, with a 2% balance tolerance our partitioner is clearly superior to `hMetis` on 8 of the 18 testcases presented (including four of the largest 5) and produces equivalent results (sharing most of the non-dominated front) on 4 of the remaining testcases. When a 10% tolerance is used, our partitioner dominates `hMetis` on 9 testcases (including 5 of the largest 6), and again ties on 4 testcases. Further, for larger examples it is able to produce a single-start result much quicker, allowing our implementation to be used in situations with tight runtime constraints.

Our implementation of multilevel FM partitioning is available on the Web at [11], together with the latest performance results.

4 Conclusions

We have presented new techniques for flat partitioning which specifically address the presence of fixed terminals. The new initial solution generator, VILE, when combined with randomization and terminal "wiggling", produces significantly better solutions when sufficient fixed terminals are present. Our experiments on ISPD-99 benchmarks validate these techniques for instances with fixed nodes arising in the top-down placement process and show their superiority when sufficiently many nodes are fixed.

The multilevel FM variant that we describe produces leading-edge results while avoiding complex heuristics and non-trivial tunings. The proposed new clustering algorithm *PinEC* is simple to implement and yet effective, while the tolerance relaxation technique used on the must clustered hypergraphs requires only calling the partitioner a second time. Finally, our implementation scales very well with increasing problem size, allowing us to produce useful partitionings of even the largest available testcase (IBM18 contains 210,341 vertices) in under two minutes.

Open issues include exploiting fixed terminals in clustering algorithms and multilevel partitioners, and a careful elucidation of the small, medium and large partitioning regimes to find the most effective techniques for each.

New information relevant to leading-edge multilevel partitioning, including benchmarks, format descriptions, implementations and performance results will be maintained on the Web at [11].

References

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

[2] C. J. Alpert, "The ISPD-98 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", *ACM/IEEE Design Automation Conf.*, pp. 530-533.

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

[5] F. Brglez, "Design of Experiments to Evaluate CAD Algorithms: Which Improvements Are Due to Improved Heuristic and Which are Merely Due to Chance?", *Technical report* CBL-04-Brglez, NCSU Collaborative Benchmarking Laboratory, April 1998.

[6] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning", *Proc. Workshop on Algorithm Engineering and Experimentation (ACM/SIAM ALENEX)*, January 1999.

[7] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Optimal Partitioners and End-case Placers for Top-down Placement" in *Proc. ACM/IEEE International Symposium on Physical Design*, June 1999, pp. 90-96.

[8] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Partitioning With Fixed Vertices: 'New' Problem and new benchmarks", in *Proc. ACM/IEEE Internation Symposium on Physical Design*, June 1999, pp. 151-157.

[9] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Hypergraph Partitioning for VLSI CAD: Methodology for Heuristic Development, Experimentation and Reporting", in *Proc. ACM/IEEE Design Automation Conf.*, June 1999, pp. 349-354.

[10] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Hypergraph Partitioning With Fixed Vertices", in *Proc. ACM/IEEE Design Automation Conf.*, June 1999, pp. 355-360.

[11] A. E. Caldwell, A. B. Kahng and I. L. Markov, "MARCO/GSRC bookshelf for VLSI CAD algorithms", `http://vlsicad.cs.ucla.edu/GSRC/bookshelf`, 1999.

[12] J. S. Cherng, S. J. Chen, C. C. Tsai, J. M. Ho, "An Efficient Two-Level Partitioning Algorithm for CLSI Circuits", *Proc. Asia and South Pacific Design Automation Conf.*, January 1999, pp. 69-72.

[13] 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 Conf. on Computer-Aided Design*, April 1997, pp. 441-446.

[14] A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits", *IEEE Transactions on Computer-Aided Design* 4(1) (1985), pp. 92-98.

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

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

[17] C.-K. Eem and J. Chong, "An Efficient Iterative Improvement Technique for VLSI Circuit Partitioning Using Hybrid Bucket Structures", *Proc. Asia and South Pacific Design Automation Conf.*, January 1999, pp. 73-76. See also *J. Institute of Electronics Engineers of Korea C* 35-C(3) (1998), pp. 16-23.

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

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

[20] D. J. Huang and A. B. Kahng, "Partitioning-Based Standard Cell Global Placement with an Exact Objective", *Proc. ACM/IEEE International Symposium on Physical Design*, 1997, pp. 18-25.

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

[22] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design", *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 526-529. Additional publications and benchmark results for hMetis-1.5 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] G. Karypis and V. Kumar, "Multilevel k-way Hypergraph Partitioning", *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 343-348.

[25] S. Wichlund and E. J. Aas, "On Multilevel Circuit Partitioning", *Proc. IEEE International Conf. on Computer Aided Design*, 1998, pp. 505-511.