Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning*

Andrew E. Caldwell, Andrew B. Kahng and Igor L. Markov

UCLA Computer Science Dept., Los Angeles, CA 90095-1596

Abstract. We discuss the implementation and evaluation of move-based hypergraph partitioning heuristics in the context of VLSI design applications. Our first contribution is a detailed software architecture, consisting of seven reusable components, that allows flexible, efficient and accurate assessment of the practical implications of new move-based algorithms and partitioning formulations. Our second contribution is an assessment of the modern context for hypergraph partitioning research for VLSI design applications. In particular, we discuss the current level of sophistication in implementation know-how and experimental evaluation, and we note how requirements for real-world partitioners – if used as motivation for research – should affect the evaluation of prospective contributions. We then use two "implicit decisions" in the implementation of the Fiduccia-Mattheyses [20] heuristic to illustrate the difficulty of achieving meaningful experimental evaluation of new algorithmic ideas. Finally, we provide anecdotal evidence that our proposed software architecture is conducive to algorithm innovation and leading-edge quality of results.

1 Introduction: Hypergraph Partitioning in VLSI Design

Given a hyperedge- and vertex-weighted hypergraph $H = (V, E)$, a *k-way partitioning* of V assigns the vertices to k disjoint nonempty partitions. The *k-way partitioning problem* seeks to minimize a given cost function $c(P^k)$ whose arguments are partitionings. A standard cost function is *net cut*,[1] which is the sum of weights of hyperedges that are cut by the partitioning (a hyperedge is *cut* exactly when not all of its vertices are in one partition). Constraints are typically imposed on the partitioning solution, and make the problem difficult. For example, certain vertices can be fixed in particular partitions (*fixed constraints*). Or, the total vertex weight in each partition may be limited (*balance constraints*), which results in an NP-hard formulation [21]. Thus, the cost function $c(P^k)$ is minimized over the set of *feasible solutions* S_f, which is a subset of the set of all possible k-way

* This research was supported by a grant from Cadence Design Systems, Inc. Author contact e-mail: {caldwell,abk,imarkov}@cs.ucla.edu.

[1] Or simply *cut*, as in *minimum cut partitioning*. Note that in the VLSI context, a circuit hypergraph is called a *netlist*; a hyperedge corresponds to a *signal net*, or *net*; and a vertex corresponds to a *module*.

partitionings. Effective move-based heuristics for k-way hypergraph partitioning have been pioneered in such works as [36], [20], [9], with refinements given by [38], [43], [26], [40], [18], [4], [12], [25], [34], [19] and many others. A comprehensive survey of partitioning formulations and algorithms, centered on VLSI applications and covering move-based, spectral, flow-based, mathematical programming-based, etc. approaches, is given in [5]. A recent update on balanced partitioning in VLSI physical design is provided by [31].

1.1 The VLSI Design Context

VLSI design has long provided driving applications and ideas for hypergraph partitioning heuristics. For example, the methods of Kernighan-Lin [36] and Fiduccia-Mattheyses [20] form the basis of today's move-based approaches. The method of Goldberg-Burstein [24] presaged the multilevel approaches recently popularized in the parallel simulation [22], [27], [32] and VLSI [3],[34],[4] communities. As noted in [5], applications in VLSI design include test; simulation and emulation; design of systems with multiple field-programmable devices; technology migration and repackaging; and top-down floorplanning and placement.

Depending on the specific VLSI design application, a partitioning instance may have directed or undirected hyperedges, weighted or unweighted vertices, etc. However, in all contexts the instance represents – at the transistor-level, gate-level, cell-level, block-level, chip-level, or behavioral description module level – a human-designed system. Such instances are highly non-random. Many efforts (e.g., [30], [14], [23], [8]) have used statistical attributes of real-world circuit hypergraphs (based on Rent's parameter [39], shape, depth, fanout distribution, etc.) to generate random hypergraphs believed relevant to evaluation of heuristics. These efforts have not yet met with wide acceptance in the VLSI community, mostly because generated instances do not guarantee "realism". Hence, the current practice remains to evaluate new algorithmic ideas against suites of benchmark instances.

In the VLSI partitioning community, performance of algorithms is typically evaluated on the ACM/SIGDA benchmarks now maintained by the Collaborative Benchmarking Laboratory at North Carolina State University `http:www.cbl.ncsu.edu/benchmarks`.[2] Alpert [2] noted that many of these circuits no longer reflect the complexity of modern partitioning instances, particularly in VLSI physical design; this motivated the release of eighteen larger benchmarks produced from internal designs at IBM [1].[3]

Salient features of benchmark (real-world) circuit hypergraphs include (see also Table 9):

- size: number of vertices can be up to one million or more (instances of all sizes are equally important).
- sparsity: average vertex degrees are typically between 3 and 5 for device-, gate- and cell-level instances; higher average vertex degrees occur in block-level design.

[2] These benchmarks are typically released by industry or academic designers at various workshops and conferences (e.g., LayoutSynth90, LayoutSynth92, Partitioning93, PDWorkshop93, ...).

[3] While those benchmarks are now used in most partitioning papers, we would like to stress that they present considerably harder partitioning problems than earlier available benchmarks available from `http://www.cbl.ncsu.edu/benchmarks`, primarily due to more esoteric distributions of node degrees and weights. See, e.g., Tables 9, 7 and 8.

- number of hyperedges (nets) typically between 0.8x and 1.5x of the number of vertices (each module typically has only one or two outputs, each of which represents the source of a new signal net).
- average net sizes are typically between 3 to 5.
- a small number of very large nets (e.g., clock, reset, test) connect hundreds or thousands of vertices.

Partitioning heuristics must also be highly efficient in order to be useful in VLSI design.[4] As a result – and also because of their flexibility in addressing variant objective functions – fast and high-quality iterative move-based partitioners based on the approach of Fiduccia-Mattheyses [20] have dominated recent practice.

1.2 The Fiduccia-Mattheyses Approach

The Fiduccia-Mattheyses (FM) heuristic for bipartitioning circuit hypergraphs [20] is an iterative improvement algorithm. Its neighborhood structure is induced by single-vertex, partition-to-partition moves.[5] 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 vertices 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 vertex is *locked*, i.e., is not allowed to move again during that pass. Since moving a vertex can change gains of adjacent vertices, 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 vertex 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.

The FM algorithm can be easily seen 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 any move gain is bounded between two and negative two times the maximal vertex degree in the hypergraph (times the maximal edge weight, if edge weights are used). This allows hashing of moves by their gains: all affected gains can be updated in linear time, yielding overall linear complexity per pass. In [20], all moves with the same gain are stored in a linked list representing a "gain bucket".

[4] For example, a modern top-down standard-cell placement tool might perform timing- and routing congestion-driven recursive min-cut bisection of a cell-level netlist to obtain a "coarse placement", which is then refined into a "detailed placement" by stochastic hill-climbing search. The *entire* placement process in currently released tools (from companies like Avant!, Cadence, CLK CAD, Gambit, etc.) takes approximately 1 CPU minute per 6000 cells on a 300MHz Sun Ultra-2 uniprocessor workstation with adequate RAM. The implied partitioning runtimes are on the order of 1 CPU second for netlists of size 25,000 cells, and 30 CPU seconds for netlists of size 750,000 cells [16]. Of course, we do not advocate performance tuning to match industrial-strength runtimes. However, absent other justifications, "experimental validation" of heuristics in the wrong runtime regimes (say, hundreds of CPU seconds for a 5000-cell benchmark) has no practical relevance.

[5] By contrast, the stronger Kernighan-Lin (KL) heuristic [36] uses a pair-swap neighborhood structure.

1.3 Contributions of This Paper

In this paper, we discuss the implementation and evaluation of move-based hypergraph partitioning heuristics, notably the FM heuristic, in the context of VLSI design applications. Our first contribution is a detailed software architecture, consisting of seven reusable components, that allows flexible, efficient and accurate assessment of the practical implications of new move-based algorithms and partitioning formulations. Our second contribution is an assessment of the modern context for hypergraph partitioning research for VLSI design applications. In particular, we discuss the current level of sophistication in implementation know-how and experimental evaluation, and we note how requirements for real-world partitioners – if used as motivation for research – should affect the evaluation of prospective contributions. We then use two "implicit decisions" in the implementation of the FM heuristic to illustrate the difficulty of achieving meaningful experimental evaluation of new algorithmic ideas. Finally, we provide brief anecdotal evidence that our proposed software architecture is conducive to algorithm innovation and leading-edge quality of results.

2 Architecture of a Move-Based Partitioning Testbench

In this section, we describe a seven-component software architecture for implementation of move-based partitioning heuristics, particularly those based on the FM approach. By way of example, we reword the Fiduccia-Mattheyses algorithm in terms of these seven software components. By carefully dividing responsibilities among components we attempt to provide the implementation flexibility and runtime efficiency that is needed to evaluate the practical impact of new algorithmic ideas and partitioning formulations.

2.1 Main Components

Common Partitioner Interface. Formally describes the input and output to partitioners without mentioning internal structure and implementation details. All partitioner implementations then conform to this input/output specification.

Initial Solution Generator. Generates partitionings that satisfy given constraints, typically using randomization in the construction.

Incremental Cost Evaluator. Evaluates the cost function for a given partitioning and dynamically maintains cost values when the partitioning is changed by applying moves. Updates typically should be performed in constant time.

Legality Checker. Verifies whether a partitioning satisfies a given constraint. The Legality Checker is used to determine the legality of a move. Multiple constraints may be handled with multiple legality checkers.

Gain Container. A general container for moves, optimized for efficient allocation, retrieval and queueing of available moves by their gains. Moves can be retrieved by, e.g., the index of the vertex being moved, and/or the source or destination partition. The Gain Container supports quick updates of the gain for a move, and fast retrieval of a move with the highest gain. The Gain Container is also independent of the incremental cost evaluator and legality checker; it is populated and otherwise managed by the Move Manager.

Move Manager. Responsible for choosing and applying one move at a time. It may rely on a Gain Container to choose the best move, or randomly generate moves. It can undo moves on request. If used in pass-based partitioners, it incrementally computes the change in gains due to a move, and updates the Gain Container.
The Move Manager maintains "status information", such as the current cost and how each partition is filled. It may be controlled by the caller via parameter updates before every move selection (e.g. a temperature parameter in simulated annealing).

Pass-Based Partitioner (proper). Solves "partitioning problems" by applying incrementally improving passes to initial solutions. A pass consists of legal moves, chosen and applied by the move manager. Within a pass, a partitioner can request that the Move Manager *undo* some of the moves, i.e. perform inverse moves. The Pass-Based Partitioner is an implementation of the *Common Partitioning Interface*.

This modularity allows for separate benchmarking and optimization of most components. It also provides flexibility to use multiple alternative implementations relevant to special cases.[6] A fundamental facility enabling such modularity is a common efficient hypergraph implementation.[7]

2.2 Component Descriptions

We now give somewhat more detailed component descriptions, omitting three components for which implementation choices are less critical.

Incremental Cost Evaluator Initialized with a hypergraph, the Incremental Cost Evaluator is responsible for evaluating the cost function for a given partitioning, and incrementally maintaining this value when the partitioning changes (i.e., a vertex is moved). When the cost function is computed as sum of hyperedge costs, those costs should also be maintained and available.

[6] For example, many optimizations for 2-way partitioning from the general k-way case can be encapsulated in the evaluator. On the other hand, in our experience optimizing the Gain Container for 2-way is barely worth maintaining separate pieces of code.

[7] A generic hypergraph implementation must support I/O, statistics, various traversals and optimization algorithms. However, no such implementation will be optimal for all conceivable uses.
In particular, the excellent LEDA library is bound to have certain inefficiencies related to hypergraph construction and memory management. We decided to implement our own reusable components based on the Standard Template Library and optimize them for our use models.
Features directly supported by the hypergraph component include memory management options, conversions, I/O, various construction options such as ignoring hyperedges of size less than 2 or bigger than a certain threshold, lazily executed calls for sorting nodes or edges in various orders etc. Many trade-off had to be made, e.g. the hypergraph objects used in critical pieces of code have to be unchangeable after their initial construction so as to allow for very efficient internal data structures.
None of the many existing generic implementations we reviewed was sufficiently malleable to meet our requirements without overwhelming their source code by numerous compiler `#defines` for adapting the code to a given use model. Having complete control over the source code and internal interfaces also allows for maximal code reuse in implementing related functionalities.

Efficient implementations typically maintain an internal state, e.g. relevant statistics, for each hyperedge. This facilitates efficient constant-time cost updates when single moves are performed. An Evaluator whose values are guaranteed to be from a small (esp. finite) range should be able to exploit this range to enable faster implementations of the Gain Container (e.g. buckets *versus* priority queues).

Interface:

- Initialize (internal structures) with a hypergraph and a partitioning solution.
- Report current cost (total or of one net) without changing internal state.
- Complete change of internal state (re-initialization) for all vertices and nets.
- Incremental change of internal state (for all nets whose cost is affected or for a given net) due to one elementary move without updating the costs.[8]

Gain Container The Gain Container stores all moves currently available to the partitioner (these may not all be legal at a given time) and prioritizes them by their gains (i.e., the immediate effect each would have on the total cost). A container of move/gain pairs is defined by the interface below, which allows quick updates to each and any move/gain pair after a single move.

A Gain Container should be able to find the move with highest gain quickly, possibly subject to various constraints such as a given source or destination partition, and may provide support for various *tie-breaking schemes* in order to choose the best move among the moves with highest gain. A Gain Container does not initiate gain updates by itself and is not aware of Cost Evaluators, the Move Manager, or how the gains are interpreted. Gain Containers do not need to determine the legality of moves. This makes them reusable for a range of constrained partitioning problems. Faster implementations (e.g. with buckets) may require that the maximal possible gain be known.

Interface:

- Add a move to the Container, given the gain.
- Get the gain for a move.
- Set the gain for a move (e.g. to update).
- Remove a move from the Container.
- Find a move of highest gain.
- Invalidate current highest gain move, in order to request the next highest gain move.[9] Typically applied if the current highest gain move appears illegal.
- Invalidate current highest gain bucket to access the next highest gain bucket.

The primary constituents of a Gain Container are a *repository* and *prioritizers*.

[8] Changing the state of one net will, in general, make the overall state of the evaluator inconsistent. This can be useful, however, for "what-if" cost lookups when a chain of incremental changes can return to the original state.

[9] Note that this does not remove the move from the Gain Container.

Repository for gain/move pairs handles allocation and deallocation of move/gain pairs, and supports fast gain lookups given a move.

Prioritizer finds a move with highest gain. In addition, may be able to choose choose best-gain moves among moves with certain properties, such as a particular destination or source partition. Updates gains and maintains them queued, in particular, is responsible for tie-breaking schemes.

We say that some moves stored in the repository are *prioritized* when they participate in the prioritizer's data structures. Not prioritizing the moves affecting a given vertex corresponds to "locking" the vertex, as it will never be chosen as the highest-gain move. The standard FM heuristic locks a given cell as soon as it is moved in a pass; however, variant approaches to locking have been proposed [15].

Move Manager A Move Manager handles the problem's move structure by choosing and applying the best move (typically, the best *legal* move), and incrementally updates the Gain Container that is used to choose the best move. The Move Manager reports relevant "status information" after each move, e.g. current cost and partition balance, which allows the caller to determine the best solution seen during the pass. In order to return to such best solution, the move manager must perform undo operations on request.

Interface:

- Choose one move (e.g., the best feasible) and apply it. Ensure all necessary updates (gain container, incremental evaluator).
- Return new "status info", e.g., total cost, partition balances, etc.
- Undo a given number of moves (each move applied must be logged to support this).

Pass-Based Partitioner (Proper) Recall that a Pass-Based Partitioner applies incrementally improving passes to initial solutions and returns best solutions seen during such passes.[10] A pass consists of moves, chosen and applied by move manager. After a pass, a Partitioner can request that the Move Manager perform undo operations to return to the best solution seen in that pass. A Partitioner decides when to stop a pass, and what intermediate solution within the pass to return to, on the basis of its control parameters and status information returned by the Move Manager after each move. The Partitioner can have access to multiple combinations of Incremental Cost Evaluators, Move Managers and Gain Containers, and can use them flexibly at different passes to solve a given partitioning problem. Note that a Partitioner is not necessarily aware of the move structure used: this is a responsibility of Move Managers.

Interface:

- Takes a "partitioning problem" and operational parameters on input.
- Returns all solutions produced, with the best solution marked.

[10] While every pass as a whole must not worsen current solution, individual moves within a pass may do so.

2.3 A Generic Component-based FM Algorithm

A "partitioning problem" consists of

- hypergraph
- solution placeholders ("buffers") with or without initial solutions
- information representing relevant constraints, e.g., fixed assignments of vertices to partitions and maximum total vertex area in each partition
- additional information required to evaluate the cost function, e.g., geometry of partitions for wirelength-driven partitioning in the top-down placement context.

The Partitioner goes over relevant places in solution buffers and eventually writes good partitioning solutions into them. An existing solution may thus be improved, but if a place is empty, an initial solution generator will be called. A relevant Move Manager must be instantiated and initialized; this includes instantiation of the constituent Evaluator and Gain Container. The Partitioner then performs successive passes as long as the solution can be improved.

At each pass, the Partitioner repetitively requests the Move Manager to pick one [best] move and apply it, and processes information about the new solutions thus obtained. Since no vertex can be moved twice in a pass, no moves will be available beyond a certain point (*end of a pass*). Some best-gain moves may increase the solution cost, and typically the solution at the end of the pass is not as good as the best solutions seen during the pass. The Partitioner then requests that the Move Manager undo a given number of moves to yield a solution with best cost.

While the reinitialization of the Move Manager at the beginning of each pass seems almost straightforward, picking and applying one move is subtle. For example, note that the Move Manager requests the best move from the gain container and can keep on requesting more moves until a move passes legality check(s). As the Move Manager applies the chosen move and locks the vertex, gains of adjacent vertices may need to be updated.

In performing "generic" gain update, the Move Manager walks all nets incident to the moving vertex and for each net computes gain updates (*delta gains*) for each of its vertices due to this net (these are combinations of the given net's cost under four distinct partition assignments for the moving and affected vertices; see Section 3.4). These partial gain updates are immediately applied through Gain Container calls, and moves of affected vertices may have their priority within the Gain Container changed. Even if the delta gain for a given move is zero, removing and inserting it into the gain container will typically change tie-breaking among moves with the same gain.

In most implementations the gain update is the main bottleneck, followed by the Gain Container construction. Numerous optimizations of generic algorithms exist for specific cost functions, netcut being particularly amenable to such optimizations.

3 Evaluating Prospective Advances in Partitioning

3.1 Formulations and Metrics for VLSI Partitioning

VLSI design presents many different flavors of hypergraph partitioning. Objective functions such as ratio-cut [46], scaled cost [11], absorption cut [45] sum of degrees, number of vertices on the cut line [28], etc. have been applied for purposes ranging from routability-driven clustering to multi-level annealing placement. In top-down coarse placement, partitioning involves fixed or "propagated" terminals [17, 44], tight partition balance constraints (and non-uniform vertex weights), and an estimated-wirelength objective (e.g., sum of half-perimeters of net bounding boxes). By contrast, for logic emulation the partitioning might have all terminals unfixed, loose balance constraints (with uniform vertex weights), and a pure min-cut objective. The partitioning can also be multi-way instead of 2-way [44, 43, 29], "multi-dimensional" (e.g., simultaneous balancing of power dissipation and module area among the partitions), timing-driven, etc. With this in mind, partitioners are best viewed as "engines" that plug into many different phases of VLSI design. Any prospective advance in partitioning technology should be evaluated in a range of contexts.

In recent VLSI CAD partitioning literature, comparisons to previous work are made using as wide a selection of benchmark instances as practically possible; using uniform vs. non-uniform vertex weights; and using tight vs. loose partition balance constraints (typically 49-51% and 45-55% constraints for bipartitioning).[11] Until recently, heuristics have typically been evaluated according to solution quality and runtime. Even though the quality-runtime tradeoff is unpredictable given widely varying problem sizes, constraints and hypergraph topologies, most papers report average and best solution quality obtained over some fixed number of independent runs (e.g., 20 or 100 runs). This reporting style can obscure the quality-runtime tradeoff, notably for small runtimes, and is a failing of the VLSI CAD community relative to the more mature metaheuristics/INFORMS communities.[12] Statistical analyses (e.g., significance tests) are not particularly popular yet, but are recognized as necessary to evaluate the significance of solution quality variation in diverse circumstances [8].

3.2 Need For "Canonical" Testbench

The components described in Section 2 yield a testbench, or "framework", that can be recombined and reused in many ways to enable experiments with

- multiple objective functions, e.g., ratio cut [46], absorption [45], the number of boundary vertices [28], the "$k-1$ objective" [13] etc.

[11] VLSI CAD researchers also routinely document whether large nets were thresholded, the details of hypergraph-to-graph conversions (e.g., when applying spectral methods), and other details necessary for others to reproduce the experiments. The reader is referred to [5, 2] for discussions of reporting methodology.

[12] See, e.g., Barr et al. [6]. Separate work of ours has addressed this gap in reporting methodology within the VLSI CAD community [10].

- multiple constraint types ([35])
- variant formulations, e.g., multi-way [43, 33, 15], replication-based [37] etc.
- new partitioning algorithms and variations

The component-based framework allows seamless replacement of old algorithms by improved ones in containing applications. Even more important, a solid testbench is *absolutely essential* to identify algorithmic improvements "at the leading edge" of heuristic technology. I.e., it is critical to evaluate proposed algorithm improvements not only against the best available implementations, but also *using a competent implementation*. This is the main point we wish to make.

In our experience, new "improvements" often look good if applied to weak algorithms, but may actually worsen strong algorithms. Only after an improvement has been thoroughly analyzed, implemented and confirmed empirically, can it be turned on by default and be applied to all evaluations of all subsequent proposed improvements. On the other hand, one often encounters pairs of conflicting improvements of which one, if applied by itself, dominates the other while the combination of the two is the worst. Therefore, interacting improvements must be implemented as options, and tested in all possible combinations.

In the following, we focus on the very pernicious danger of reporting "experimental results" that are irreproducible and possibly meaningless due to a poorly implemented partitioning testbench. We demonstrate that a fundamental cause of a poor testbench is failure to understand the "implicit implementation decisions" that dominate quality/runtime tradeoffs. A corollary is that researchers must clearly report such "implicit decisions" in order for results to be reproducible.

3.3 Taxonomy of Algorithm and Implementation Improvements

In this section we propose a general taxonomy of implementation decisions for optimization metaheuristics, and illustrate it for the Fiduccia-Mattheyses heuristic [20] and its improvements.

Modifications of the algorithm — important changes to steps or the flow of the original algorithm as well as new steps and features.

- "lookahead" tie-breaking [38] – among moves with the same gain, one chooses those which increase gains of other moves
- CLIP [18] – instead of actual gains of moves, maintains their "updated" gain, i.e., the actual gain minus the gain in the beginning of the pass. With updated gain used for move selection, CLIP tends to move (clusters of) adjacent vertices together and achieves considerably better quality.
- *cut line refinement* [27] – only vertices incident to nets with nonzero cost are considered in move selection. If net changes its cost from 0 to 1 during a vertex move, all its vertices previously disregarded are brought into consideration.
- *multiple unlocking* [15] – allows vertices to move more than once during a pass.

Implicit decisions – underspecified features and ambiguities in the original algorithm description that need to be resolved in any particular implementation. Examples for the Fiduccia-Mattheyses heuristic include:
- *tie-breaking* in choosing highest gain bucket (see Section 3.4)
- *tie-breaking* on where to attach new element in gain bucket, i.e., LIFO versus FIFO versus random [26][13]
- whether to update, or skip the updating, when the delta gain of a move is zero (see Section 3.4)
- breaking ties when selecting the best solution during the pass — choose the first or last one encountered, or the one that is furthest from violating constraints.

Tuning that can change the result — minor algorithm or implementation changes, typically to avoid particularly bad special cases or pursue only "promissing computations". If bad cases are rare and criteria for promising computations are good, the resulting quality may be affected minimally.
- thresholding large nets from the input to reduce run time
- skipping gain update for large nets to reduce run time
- skipping zero delta gain updates changes the resolution of hash collisions in the Gain Container.
- loose net removal [12] performs gain updates only for [loose] nets that are likely to be uncut, and skips gain updates for all other nets
- stable net removal [12] performs special passes ("kick-moves") between regular FM passes deliberately uncutting nets that have been cut during the previous passes
- allowing illegal solutions during a pass (to improve hill-climbing ability of the algorithm) [19]

Tuning that can not change the result — minor algorithm or implementation changes to simplify computations in critical or statistically significant special cases.
- skipping nets which cannot have non-zero delta gains (updates)
- netcut-specific optimizations
- optimizations for nets of small degree
- 2-way specific optimizations

3.4 An Empirical Illustration

We now illustrate how "implicit implementation decisions" can severely distort the experimental assessment of new algorithmic ideas. Uncertainties in the description of the Fiduccia-Mattheyses algorithm have been previously analyzed, notably in [26], where the authors show that inserting moves into gain buckets in LIFO order is much preferable to doing so in FIFO order (also a constant-time insertion) or at random. Since the work of [26], all FM implementations that we are aware of use LIFO insertion.[14] In our experiments, we consider the following two implicit implementation decisions:

[13] In other words, gain buckets can be implented as stacks, queues or random priority queues where the chances of all elements to be selected are equal at all times. [26] demonstrated that stack-based gain containers (i.e. LIFO) are superior.

[14] A series of works in the mid-1990s retrospectively show that the LIFO order allows vertices in "natural clusters" to move together across the cutline. The CLIP variant of [18] is a more direct way of moving clusters.

– **Zero delta gain update.** Recall that when a vertex x is moved, the gains for all vertices y on nets incident to x must potentially be updated. In all FM implementations, this is done by going through the incident nets one at a time, and computing the changes in gain for vertices y on these nets. A straightforward implementation computes the change in gain ("delta gain") for y by adding and subtracting four cut values for the net under consideration,[15] and immediately updating y's position in the gain container.

ALGORITHM		TESTCASES with **unit** areas and **10%** balance					
Updates	Bias	**primary1**	**primary2**	**biomed**	**ibm01**	**ibm02**	**ibm03**
		Flat LIFO FM					
All Δ_{gain}	Away	**102**(0.253)	**472**(1.6)	**447**(31)	**1723**(12.8)	**1404**(30.8)	**4191**(33.2)
All Δ_{gain}	Part0	**96.2**(0.26)	**438**(1.76)	**386**(36.4)	**1226**(16.3)	**1468**(43.2)	**3854**(37.8)
All Δ_{gain}	Toward	**80.9**(0.257)	**294**(1.58)	**370**(35.9)	**577**(12.6)	**585**(23.7)	**3209**(37.7)
Nonzero	Away	**72.4**(0.248)	**283**(1.48)	**149**(29.7)	**529**(8.44)	**471**(18.9)	**2327**(28)
Nonzero	Part0	**76.7**(0.237)	**280**(1.41)	**127**(26.8)	**436**(8.81)	**444**(18.4)	**2087**(29.1)
Nonzero	Toward	**75.4**(0.228)	**263**(1.47)	**133**(26.7)	**454**(9.29)	**453**(17)	**2093**(26.9)
		Flat CLIP FM					
All Δ_{gain}	Away	**63.4**(0.356)	**227**(2.2)	**140**(54.8)	**463**(16.9)	**662**(50.7)	**1705**(49)
All Δ_{gain}	Part0	**63.3**(0.337)	**216**(2.34)	**117**(46.9)	**395**(15.9)	**513**(41)	**1612**(44.7)
All Δ_{gain}	Toward	**63.5**(0.319)	**209**(2.05)	**116**(43)	**436**(13.3)	**446**(40.3)	**1515**(40)
Nonzero	Away	**62.6**(0.287)	**219**(1.76)	**109**(36.2)	**415**(13.6)	**466**(35.5)	**1631**(40.1)
Nonzero	Part0	**61.3**(0.299)	**207**(1.86)	**105**(24.9)	**371**(13.7)	**442**(35)	**1543**(39.3)
Nonzero	Toward	**60.2**(0.295)	**216**(1.81)	**105**(26.9)	**397**(13.2)	**445**(32.1)	**1528**(35.6)
		ML LIFO FM					
All Δ_{gain}	Away	**59.3**(0.838)	**171**(4.79)	**142**(36.3)	**236**(27)	**285**(62.2)	**1023**(156)
All Δ_{gain}	Part0	**58.7**(0.795)	**170**(4.98)	**137**(38.6)	**238**(25.8)	**288**(65.9)	**992**(133)
All Δ_{gain}	Toward	**59.5**(0.832)	**166**(4.52)	**138**(34.2)	**236**(27.6)	**291**(61.4)	**995**(121)
Nonzero	Away	**58.5**(0.778)	**168**(4.69)	**139**(27.8)	**234**(26.3)	**282**(60.5)	**1010**(126)
Nonzero	Part0	**57.3**(0.848)	**166**(4.71)	**140**(27)	**232**(25.9)	**280**(58.3)	**1035**(141)
Nonzero	Toward	**57.6**(0.745)	**164**(4.62)	**140**(28.5)	**245**(26.9)	**281**(56.7)	**988**(126)
		ML CLIP FM					
All Δ_{gain}	Away	**59.8**(0.782)	**171**(4.77)	**144**(33.8)	**239**(25.7)	**281**(65.7)	**994**(133)
All Δ_{gain}	Part0	**58.8**(0.853)	**169**(4.78)	**136**(37.6)	**235**(27.4)	**278**(62)	**1213**(170)
All Δ_{gain}	Toward	**58.5**(0.836)	**166**(4.79)	**136**(36.3)	**239**(24.5)	**281**(60.1)	**1023**(136)
Nonzero	Away	**59.1**(0.816)	**170**(4.37)	**142**(26.2)	**239**(25.8)	**286**(51)	**1009**(109)
Nonzero	Part0	**58.1**(0.834)	**166**(4.78)	**137**(26.1)	**240**(26)	**286**(57.7)	**1009**(118)
Nonzero	Toward	**59.1**(0.885)	**164**(4.49)	**138**(28.5)	**237**(24.6)	**284**(55.8)	**1000**(121)

Table 1. Average cuts in partitioning with **unit** areas and **10%** balance tolerance, over 100 independent runs. Average CPU time per run in Sun Ultra-1 (140MHz) seconds is given in parentheses.

Notice that sometimes the delta gain can be zero. An implicit implementation decision is whether to reinsert a vertex y when it experiences a zero delta gain move (**"AllΔ_{gain}"**), or whether to skip the gain update (**"Nonzero"**). The former will shift the position of y within the same gain bucket; the latter will leave y's position unchanged. The effect of zero delta gain updating is not immediately obvious.[16]

[15] These four cut values correspond to: (a) x, y in their original partitions; (b) x in original partition, y moved; (c) x moved, y in original partition; and (d) x and y both moved. (a) - (b) is the original gain for y due to the net under consideration; (c) - (d) is the new gain for y due to the same net. The difference ((a)-(b)) - ((c)-(d)) is the delta gain. See [29] for a discussion.

[16] The gain update method presented in [20] has the *side effect* of skipping all zero delta gain updates. However, this method is both netcut- and two-way specific; it is not certain that a first-time experimenter with FM will find analogous solutions for k-way partitioning with a general objective.

– **Tie-breaking between two highest-gain buckets in move selection.** When the gain container is implemented such that available moves are segregated, typically by source or destination partition, there can be more than one nonempty highest-gain bucket. Notice that when the balance constraint is anything other than "exact bisection", it is possible for all the moves at the heads of the highest-gain buckets to be legal. The FM implementer must choose a method for dealing with this situation. In our experiments, we contrast three approaches:[17] (i) choose the move that is not from the same partition as the last vertex moved (**"away"**); (ii) choose the move in partition 0 (**"part0"**); and (iii) choose the move from the same partition as the last vertex moved (**"toward"**).

ALGORITHM		TESTCASES with **actual** areas and **10%** balance					
Updates	Bias	**primary1**	**primary2**	**biomed**	**ibm01**	**ibm02**	**ibm03**
				Flat LIFO FM			
All Δ_{gain}	Away	**99.7**(0.284)	**470**(2.09)	**442**(34.2)	**1680**(16.3)	**2291**(29.4)	**4076**(24.3)
All Δ_{gain}	Part0	**93.7**(0.294)	**425**(2.38)	**384**(43.8)	**863**(18)	**1880**(33.3)	**3969**(25.9)
All Δ_{gain}	Toward	**80**(0.301)	**292**(2.05)	**365**(38.8)	**490**(12.8)	**734**(21.6)	**3262**(22)
Nonzero	Away	**72.8**(0.279)	**286**(1.67)	**154**(30.4)	**574**(10.1)	**507**(15.1)	**2266**(23.5)
Nonzero	Part0	**71**(0.274)	**273**(1.73)	**125**(28.6)	**503**(9.85)	**470**(16.7)	**2161**(21.7)
Nonzero	Toward	**71.7**(0.252)	**259**(1.88)	**132**(29.8)	**481**(9.1)	**437**(13.8)	**2167**(21.4)
				Flat CLIP FM			
All Δ_{gain}	Away	**62.6**(0.384)	**233**(2.62)	**139**(50.6)	**493**(21)	**740**(37.8)	**2260**(32.8)
All Δ_{gain}	Part0	**58.9**(0.392)	**222**(2.79)	**128**(43.1)	**428**(16.8)	**573**(36.2)	**1826**(44.6)
All Δ_{gain}	Toward	**59.9**(0.386)	**223**(2.37)	**107**(37.2)	**417**(16.1)	**478**(28.7)	**1562**(47.7)
Nonzero	Away	**57.7**(0.361)	**220**(2.52)	**112**(33.9)	**479**(15.7)	**432**(25.6)	**1689**(33.9)
Nonzero	Part0	**56.6**(0.339)	**218**(2.49)	**105**(23.5)	**415**(15.6)	**421**(29.3)	**1585**(32.6)
Nonzero	Toward	**56.8**(0.344)	**206**(2.52)	**104**(22.4)	**442**(13.5)	**404**(25)	**1526**(28.7)
				ML LIFO FM			
All Δ_{gain}	Away	**55**(0.94)	**140**(4.09)	**142**(44.6)	**264**(25.3)	**281**(40.5)	**829**(55)
All Δ_{gain}	Part0	**55.4**(0.954)	**140**(4.4)	**138**(47.6)	**263**(25.6)	**291**(38.7)	**823**(55.1)
All Δ_{gain}	Toward	**54.6**(0.848)	**141**(4.42)	**133**(46.8)	**263**(25)	**291**(41.3)	**822**(54.4)
Nonzero	Away	**54.6**(0.87)	**139**(4.19)	**141**(34.8)	**267**(23.6)	**277**(40.8)	**819**(46.8)
Nonzero	Part0	**54.1**(0.837)	**138**(4.54)	**136**(32.5)	**260**(25.5)	**281**(41.1)	**808**(46)
Nonzero	Toward	**55.2**(0.831)	**136**(4.44)	**136**(34.4)	**262**(22.6)	**286**(39.5)	**809**(42.6)
				ML CLIP FM			
All Δ_{gain}	Away	**55.9**(0.89)	**141**(4.86)	**142**(43.8)	**263**(27)	**281**(41)	**822**(51)
All Δ_{gain}	Part0	**54.7**(0.889)	**142**(4.84)	**131**(43)	**265**(24.3)	**282**(44.3)	**829**(54.4)
All Δ_{gain}	Toward	**55.8**(0.891)	**140**(4.99)	**138**(38.6)	**259**(25.6)	**279**(40.6)	**807**(54.3)
Nonzero	Away	**56.1**(0.83)	**141**(4.1)	**139**(33.4)	**266**(26.2)	**284**(40.9)	**826**(47.9)
Nonzero	Part0	**53**(0.895)	**136**(4.23)	**139**(31.3)	**257**(24.8)	**274**(41.4)	**809**(48)
Nonzero	Toward	**54.5**(0.839)	**143**(4.31)	**137**(36.1)	**261**(24.6)	**287**(42.8)	**833**(44.9)

Table 2. Average cuts in partitioning with **actual** areas and **10%** balance tolerance, over 100 independent runs. Average CPU time per run in Sun Ultra-1 (140MHz) seconds is given in parentheses.

Our experimental testbench allows us to test an FM variant in the context of flat LIFO (as described in [26]), flat CLIP (as described in [18]), and multilevel LIFO and multilevel CLIP (as described in [4]). Our implementations are in C++ with heavy use of STL3.0; we currently run in the Sun Solaris 2.6 and Sun CC4.2 environment. We use standard VLSI benchmark instances available on the Web at [1] and several older benchmarks from

[17] These approaches are described for the case of bipartitioning. Other approaches can be devised for k-way partitioning.

http://www.cbl.ncsu.edu/benchmarks. Node and hyperedge statistics for the benchmarks are presented in Tables 7, 8 and 9. Our tests are for bi-partitioning only. We evaluate all partitioning variants using actual vertex areas and unit vertex areas, incorporating the standard protocols for treating pad areas described in [2]. We also evaluate all partitioning variants using a 10% balance constraint (i.e., each partition must have between 45% and 55% of the total vertex area) as well as a 2% balance constraint. All experiments were run on Sun Ultra workstations, with runtimes normalized to Sun Ultra-1 (140MHz) CPU seconds. Each result represents a set of 100 independent runs with random initial starting solutions; Tables 1-4 report triples of form "**average cut** (average CPU sec)". From the data, we make the following observations.

ALGORITHM		TESTCASES with **unit** areas and **2%** balance					
Updates	Bias	**primary1**	**primary2**	**biomed**	**ibm01**	**ibm02**	**ibm03**
				Flat LIFO FM			
All Δ_{gain}	Away	**102**(0.247)	**486**(1.6)	**459**(29.3)	**1778**(16.4)	**1810**(36.5)	**4175**(34.1)
All Δ_{gain}	Part0	**102**(0.27)	**465**(2.06)	**422**(37.8)	**1673**(19.8)	**1570**(43.3)	**4064**(35)
All Δ_{gain}	Toward	**102**(0.264)	**374**(1.94)	**316**(36.5)	**1030**(15.4)	**931**(30)	**3323**(39.6)
Nonzero	Away	**80.5**(0.222)	**285**(1.31)	**166**(22.8)	**543**(10.2)	**549**(18)	**2304**(29.2)
Nonzero	Part0	**80**(0.236)	**289**(1.47)	**150**(26.3)	**551**(11.8)	**549**(18.6)	**2383**(31.1)
Nonzero	Toward	**78.8**(0.219)	**291**(1.41)	**143**(24.8)	**508**(10.9)	**551**(19.3)	**2285**(33.1)
				Flat CLIP FM			
All Δ_{gain}	Away	**69.8**(0.351)	**246**(2.22)	**149**(53.9)	**555**(21.1)	**707**(46.2)	**1747**(45.5)
All Δ_{gain}	Part0	**68**(0.319)	**242**(2.3)	**138**(52)	**562**(21.3)	**636**(45.5)	**1686**(51.7)
All Δ_{gain}	Toward	**68.4**(0.35)	**236**(2.14)	**121**(44)	**561**(19.3)	**619**(44)	**1664**(43.2)
Nonzero	Away	**66.3**(0.284)	**231**(2.02)	**130**(29.1)	**488**(17.5)	**534**(46.8)	**1650**(41.9)
Nonzero	Part0	**66.4**(0.305)	**226**(2.2)	**124**(33.7)	**467**(17.1)	**507**(37.4)	**1618**(41.2)
Nonzero	Toward	**64.6**(0.304)	**223**(2.13)	**125**(31.4)	**474**(15.8)	**559**(36.9)	**1551**(41.9)
				ML LIFO FM			
All Δ_{gain}	Away	**64.2**(0.818)	**183**(4.95)	**147**(36.8)	**280**(29.7)	**399**(70.6)	**1043**(138)
All Δ_{gain}	Part0	**63.8**(0.871)	**182**(5.21)	**145**(33.5)	**282**(31.6)	**406**(66.8)	**999**(113)
All Δ_{gain}	Toward	**63.2**(0.836)	**180**(4.74)	**145**(33.6)	**272**(30.8)	**421**(66.1)	**1035**(118)
Nonzero	Away	**62.9**(0.807)	**182**(4.69)	**142**(28.1)	**274**(29.7)	**396**(60.4)	**1037**(123)
Nonzero	Part0	**61.8**(0.849)	**176**(5.15)	**144**(26.8)	**270**(28.8)	**403**(60.2)	**1048**(121)
Nonzero	Toward	**61.4**(0.879)	**181**(4.91)	**143**(24.8)	**271**(29.1)	**415**(56)	**1015**(119)
				ML CLIP FM			
All Δ_{gain}	Away	**63.4**(0.912)	**180**(5.27)	**146**(32.6)	**276**(30.2)	**400**(68.7)	**1042**(138)
All Δ_{gain}	Part0	**62.8**(0.849)	**180**(5.18)	**145**(39.1)	**278**(29.1)	**408**(67.8)	**1022**(129)
All Δ_{gain}	Toward	**63.3**(0.871)	**178**(5.36)	**141**(36.7)	**270**(29.4)	**425**(64.4)	**1032**(101)
Nonzero	Away	**61.2**(0.887)	**178**(5.12)	**142**(26.8)	**268**(28.4)	**392**(57)	**1037**(116)
Nonzero	Part0	**62.8**(0.878)	**176**(5.14)	**142**(29.3)	**275**(28.6)	**409**(55.6)	**1026**(111)
Nonzero	Toward	**63**(0.924)	**179**(4.86)	**142**(28.5)	**271**(28.2)	**409**(52.9)	**1034**(115)

Table 3. Average cuts in partitioning with **unit** areas and **2%** balance tolerance, over 100 independent runs. Average CPU time per run in Sun Ultra-1 (140MHz) seconds is given in parentheses.

- The average cutsize for a flat partitioner can increase by rather stunning percentages if the worst combination of choices is used instead of the best combination. Such effects far outweigh the typical solution quality improvements reported for new algorithm ideas in the partitioning literature.
- Moreover, we see that one wrong implementation decision can lead to misleading conclusions with respect to other implementation decisions.

For example, when zero delta gain updates are made (a wrong decision), the "part0" biasing choice appears significantly worse than the "toward" choice. However, when zero delta gain updates are skipped, "part0" is as good as or even slightly better than "toward".[18]

– Stronger optimization engines (order of strength: ML CLIP > ML LIFO > flat CLIP > flat LIFO) can tend to decrease the "dynamic range" for the effects of implementation choices. This is actually a danger: e.g., developing a multilevel FM package may hide the fact that the underlying flat engines are badly implemented. At the same time, the effects of a bad implementation choice are still apparent even when that choice is wrapped within a strong optimization technique (e.g., ML CLIP).

ALGORITHM		TESTCASES with **actual** areas and **2%** balance					
Updates	Bias	**primary1**	**primary2**	**biomed**	**ibm01**	**ibm02**	**ibm03**
Flat LIFO FM							
All Δ_{gain}	Away	**105**(0.283)	**481**(2.05)	**446**(27.8)	**1885**(11.1)	**3256**(34.8)	**4389**(25.4)
All Δ_{gain}	Part0	**103**(0.311)	**467**(2.4)	**428**(34.9)	**1909**(12.3)	**2440**(33.5)	**4166**(27.1)
All Δ_{gain}	Toward	**96.8**(0.285)	**380**(2.15)	**408**(30.7)	**1023**(11.6)	**1274**(22.7)	**3939**(22.3)
Nonzero	Away	**80.5**(0.258)	**291**(1.78)	**164**(27.3)	**639**(8.86)	**551**(14.9)	**2838**(25.2)
Nonzero	Part0	**79**(0.271)	**294**(1.74)	**157**(29.2)	**660**(7.87)	**573**(17)	**2938**(24)
Nonzero	Toward	**79.7**(0.25)	**280**(1.79)	**153**(24.7)	**607**(7.62)	**543**(15.8)	**2843**(25.4)
Flat CLIP FM							
All Δ_{gain}	Away	**66.2**(0.361)	**244**(2.71)	**148**(57)	**842**(15.3)	**1841**(27.5)	**3623**(23.4)
All Δ_{gain}	Part0	**66.3**(0.395)	**242**(2.81)	**141**(54.6)	**772**(14.9)	**1499**(32.4)	**3543**(29.3)
All Δ_{gain}	Toward	**65.9**(0.393)	**237**(2.7)	**138**(58)	**615**(13)	**945**(21.5)	**3066**(25.6)
Nonzero	Away	**64.5**(0.351)	**231**(2.66)	**130**(33.3)	**542**(12.1)	**574**(18.5)	**2689**(22.8)
Nonzero	Part0	**62**(0.371)	**242**(2.49)	**123**(35)	**556**(12.4)	**582**(17.8)	**2732**(23.1)
Nonzero	Toward	**63.9**(0.377)	**233**(2.31)	**124**(34.9)	**528**(11.8)	**562**(15.3)	**2504**(23.2)
ML LIFO FM							
All Δ_{gain}	Away	**62.8**(0.905)	**163**(5.27)	**145**(34.9)	**289**(27.9)	**433**(42.9)	**958**(59.4)
All Δ_{gain}	Part0	**62.5**(0.954)	**166**(5.03)	**144**(38.7)	**289**(27.2)	**429**(44.6)	**957**(56.3)
All Δ_{gain}	Toward	**61.1**(0.974)	**161**(5.28)	**143**(36.3)	**289**(27.7)	**423**(47)	**971**(58.7)
Nonzero	Away	**60.4**(0.914)	**158**(4.52)	**142**(29.9)	**287**(22.7)	**432**(39.6)	**969**(52.2)
Nonzero	Part0	**59.9**(0.882)	**158**(4.36)	**142**(29.3)	**282**(25.3)	**421**(44)	**952**(50.6)
Nonzero	Toward	**60.5**(0.9)	**159**(4.5)	**142**(27.5)	**276**(25.4)	**419**(43.2)	**959**(52.5)
ML CLIP FM							
All Δ_{gain}	Away	**63.5**(0.88)	**163**(5)	**144**(35.9)	**283**(24.5)	**428**(41.5)	**960**(59.3)
All Δ_{gain}	Part0	**62.5**(0.891)	**161**(4.24)	**143**(37.1)	**289**(25.3)	**441**(46.5)	**969**(63.6)
All Δ_{gain}	Toward	**61.9**(0.927)	**162**(4.83)	**141**(37.8)	**284**(24.9)	**425**(44.2)	**953**(62.1)
Nonzero	Away	**60.2**(0.939)	**160**(4.66)	**144**(31.8)	**283**(23)	**414**(48.7)	**957**(50.4)
Nonzero	Part0	**61**(0.895)	**161**(4.89)	**144**(28.1)	**285**(24.7)	**447**(41.8)	**934**(53)
Nonzero	Toward	**60.9**(0.864)	**155**(4.7)	**144**(29.6)	**282**(22.4)	**433**(42.6)	**959**(50.6)

Table 4. Average cuts in partitioning with **actual** areas and **2%** balance tolerance, over 100 independent runs. Average CPU time per run in Sun Ultra-1 (140MHz) seconds is given in parentheses.

Another Illustration Further illustration of the pitfalls of a suboptimal testbench is, we believe, to be found in the partitioning studies of Marks et al. [42] reported at ALEX-98. The authors of this work note, very much as we have, that "Unfortunately, empirical analysis of algorithm performance is often done poorly, which sometimes leads to erroneous conclusions." The authors of [42] propose two new heuristics, of which the better is called

[18] We have observed other similar reversals, e.g., in our experience multiple unlocking is less valuable than reported in [15].

PHC/SG+KL. The discussion observes, "In a series of time-equated comparisons with large-sample runs of pure Kernighan-Lin, the new algorithm demonstrates significant superiority in terms of the best bisections found." Marks et al. run their PHC/SG+KL heuristic, then run the Kernighan-Lin (KL) heuristic as many times as are needed to equate the CPU usage. This is performed 25 separate times, with mean and standard deviation of the best result over the multi-starts being reported. While the authors use a number of benchmarks from the VLSI CAD domain, in their testbench no comparisons with previous literature were possible. We have replicated the experiment of [42] for three VLSI test cases for which the superiority of PHC/SG+KL over KL is especially large. Our "heuristic" is simply taking the best of 10 starts of flat CLIP FM. The data shown in Table 5 indicate that the KL implementation of [42] may have led to potentially misleading conclusions.

Graph			KL				PHC/SG+KL		CLIP-FM			
Name	$\|V\|$	avg deg	Time	Runs	Mean	σ	Mean	σ	Runs	Mean	σ	Time'
primary1	833	15.0	15.1	430.6	281.4	17.3	218.0	1.4	10	226.1	9.49	2.04
primary2	3014	24.7	79.6	490.8	1322.9	81.7	585.4	27.0	10	606.2	13.02	15.72
industry3	15406	23.3	408.7	295.6	6827.2	294.8	990.0	132.2	10	797.0	47.8	115.9

Table 5. Comparison against results of [42]. All vertices are assigned equal (unit) area. All hyperedges are converted to cliques with unit-weight edges. An exact bisection constraint is enforced. We run 10 starts of our flat CLIP-FM partitioning engine. This trial is repeated 25 times; we report average best result over the 10 starts, the standard deviation of this average best result, and the average total CPU time for the 10 starts. Data for KL and PHC/SG+KL are quoted from [42]. CPU times (Time) from [42] are for a DEC AlphaStation 500/500. CPU times for our experiments (Time') are for a 300 MHz Sun Ultra-10. Apparently, our workstation had a considerably slower CPU clock and smaller processor cache.

4 Conclusions

The results reported in the previous section are for a "detuned" or "generalized" version of our testbench, where we deliberately re-enabled the zero delta gain update and gain bucket choice as options. In our current testbench, these are not options, i.e., our FM-based engines always skip zero delta gain updates and always choose the "toward" gain bucket in case of ties. Our current testbench is also able to invoke several speedups that exploit the nature of the netcut objective and the two-way partitioning context. Comparison of the results in 6 against the best netcut values ever recorded in the literature [2] shows that our testbench is indeed at the leading edge of solution quality.) We emphasize that our testbench is general: we flexibly address new objectives, neighborhood structures, and constraint types. We therefore incur some runtime overhead due to templating, object-oriented code structure, conditional tests, etc. At the same time, our testbench is sufficiently fast that we can accurately assess quality-runtime tradeoff implications of new algorithm ideas.
In conclusion, we have noted that replicating reported results is a well-known problem in the VLSI partitioning community [5]. Disregarding the issues of experimental protocols, statistical significance tests, data reporting methodology, need to compare with previous results, and so on [6, 8], we

Configuration	primary1	primary2	biomed	ibm01	ibm02	ibm03
2% unit area	57.1(0.418)	164.7(1.3)	132.8(2.24)	275.0(5.7)	372.1(10.6)	1031.4(11.6)
10% unit area	52.1(0.412)	143.6(1.3)	117.0(2.16)	247.9(5.82)	268.8(10.3)	820.9(11.8)
2% actual area	61.3(0.389)	193.1(1.41)	137.2(2.07)	284.5(6.63)	421.3(14.6)	1079.8(15.8)
10% actual area	54.1(0.421)	177.8(1.07)	125.0(2.23)	244.4(5.36)	275.7(14.7)	1062.1(16.4)

Table 6. Results of applying our optimized multi-level partitioner on 6 test-cases. Solutions are constrained to be within 2% or 10% of bisection. Data expressed as (average cut / average CPU time) over 100 runs, with the latter normalized to CPU seconds on a 140MHz Sun Ultra-1.

still find that implementations reported in the literature are almost never described in sufficient detail for others to reproduce results. In this paper, we have illustrated the level of detail necessary in reporting; our illustration also shows how even expert programmers may fail to write a useful testbench for research "at the leading edge". Our main contributions have been the description of a software architecture for partitioning research, a review of the modern context for such research in the VLSI CAD domain, and a detailed list of hidden implementation decisions that are crucial to obtaining a useful testbench.

We thank Max Moroz for design, implementation and support of an efficient STL-based hypergraph package used in the paper in lieu of the LEDA library.

Node degree statistics for testcases					
primary1(833)	primary2(3014)	biomed(6514)	ibm01(12752)	ibm02(19601)	ibm03(23136)
deg: #	deg: #	deg: #	deg: #	deg: #	deg: #
Avg: 3.49	Avg: 3.72	Avg: 3.230	Avg: 3.965	Avg: 4.143	Avg: 4.044
1: 48	1: 43	1: 97	1: 781	1: 1591	1: 363
2: 145	2: 453	2: 792	2: 3722	2: 4448	2: 7093
3: 205	3: 1266	3: 4492	3: 2016	3: 2318	3: 4984
4: 273	4: 519	4: 440	4: 1430	4: 1714	4: 4357
5: 234	5: 402	5: 35	5: 1664	5: 3406	5: 2778
6: 5	6: 23	6: 658	6: 1761	6: 4435	6: 1252
7: 22	7: 260		7: 542	7: 1055	7: 300
9: 1	8: 4		8: 194	8: 303	8: 689
	9: 44		9: 368	9: 319	9: 708
			10: 3	29: 2	10: 50
			13: 220	32: 1	11: 25
			39: 1	51: 4	12: 192
				53: 3	13: 47
				60: 1	14: 7
				69: 1	16: 10
					17: 7
					18-19: 4
					20: 16
					21-23: 8
					24: 136
					25: 101
					84-100: 9

Table 7. Hypergraph node degree statistics. The numbers of nodes in degree ranges are given for each testcase together with the total nodes and average node degree.

Node weights statistics for testcases						
weight range	# nodes					
	primary1	primary2	biomed	ibm01	ibm02	ibm03
1	81	107	97	12749	19589	23126
2				2	3	
3					3	
4	89	367				
5			723			7
6	142	715			1	
7	151	494	2818			1
8	73	398				
9			1323		1	
10						
11	27	278			11	
12			35			
13						
14			3			
15	7	550				
16						
17	1	52				
18			860			
19						
20	262	53	655	1	1	2

Table 8. Hypergraph node weight statistics The interval between the smallest and largest node weight has been divided into 20 equal ranges for each testcase. For each such range we report the number of nodes with weight in this range.

Hyperedge statistics for testcases					
primary1(902)	primary2(3029)	biomed(5742)	ibm01(14111)	ibm02(19568)	ibm03(27401)
deg: #	deg: #	deg: #	deg: #	deg: #	deg: #
Avg: 3.22	Avg: 3.70	Avg: 3.664	Avg: 3.583	Avg: 4.146	Avg: 3.415
2: 494	2: 1835	2: 3998	2: 8341	2: 10692	2: 17619
3: 236	3: 365	3: 870	3: 2082	3: 1934	3: 3084
4: 62	4: 203	4: 427	4: 1044	4: 1951	4: 2155
5: 26	5: 192	5: 184	5: 737	5: 1946	5: 1050
6: 25	6: 120	6: 13	6: 407	6: 376	6: 790
7: 13	7: 52	7: 11	7: 235	7: 332	7: 436
8: 2	8: 14	8: 28	8: 188	8: 256	8: 342
9: 9	9: 83	9: 7	9: 192	9: 424	9: 501
10: 1	10: 14	10: 4	10: 194	10: 431	10: 235
11: 6	11: 35	11: 5	11: 147	11: 498	11: 198
12: 9	12: 5	12: 5	12: 91	12: 46	12: 162
13: 1	13: 3	13: 1	13: 133	13: 52	13: 195
14: 3	14: 10	14: 2	14: 54	14: 52	14: 112
16: 1	15: 3	15: 41	15: 34	15: 85	15: 79
17: 11	16: 1	17: 21	16: 54	16: 94	16: 100
18: 3	17: 72	18: 1	17: 31	17: 143	17: 119
	18: 1	20: 2	18: 17	18: 100	18: 81
	23: 1	21: 65	19: 12	19: 44	19: 41
	26: 1	22: 34	20: 21	20: 15	20: 24
	29: 1	23: 6	21: 18	21: 11	21: 12
	30: 1	24: 6	22: 31	22: 5	22: 16
	31: 1	43: 6	23: 18	24-29: 10	23: 9
	33: 14	656: 4	25: 2	30: 4	24: 3
	34: 1	861: 1	28: 1	31: 11	25: 6
	37: 1		30: 2	32: 4	26: 3
			31: 2	33: 2	27: 2
			32: 5	34: 1	28: 2
			33: 6	35: 5	29: 6
			34: 1	36: 3	30: 1
			35: 7	37: 2	31: 2
			38: 1	38: 2	31: 3
			39: 2	39: 1	32: 3
			42: 1	40-97: 51	33: 5
				107: 1	34: 3
				134: 1	37-55: 5

Table 9. Hyperedge statistics. The numbers of hyperedges in ranges are given for each testcase together with total hyperedges and average hyperedge degree.

References

1. C. J. Alpert, "Partitioning Benchmarks for VLSI CAD Community", Web page, `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 and L. W. Hagen and A. B. Kahng, "A Hybrid Multilevel/Genetic Approach for Circuit Partitioning", *Proc. IEEE Asia Pacific Conference on Circuits and Systems*, 1996, pp. 298-301.
4. C. J. Alpert, J.-H. Huang and A. B. Kahng,"Multilevel Circuit Partitioning", *ACM/IEEE Design Automation Conference*, pp. 530-533. `http://vlsicad.cs.ucla.edu/papers/conference/c68.ps`
5. C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration*, 19(1995) 1-81.
6. R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende and W. R. Stewart, "Designing and Reporting on Computational Experiments with Heuristic Methods", *technical report* (extended version of *J. Heuristics* paper), June 27, 1995.
7. F. Brglez, "ACM/SIGDA Design Automation Benchmarks: Catalyst or Anathema?", *IEEE Design and Test*, 10(3) (1993), pp. 87-91.
8. 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.
9. T. Bui, S. Chaudhuri, T. Leighton and M. Sipser, "Graph Bisection Algorithms with Good Average Behavior", *Combinatorica* 7(2), 1987, pp. 171-191.
10. A. E. Caldwell, A. B. Kahng and I. L. Markov, *manuscript*, 1998.
11. P. K. Chan and M. D. F. Schlag and J. Y. Zien, "Spectral K-Way Ratio-Cut Partitioning and Clustering", *IEEE Transactions on Computer-Aided Design*, vol. 13 (8), pp. 1088-1096.
12. 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.
13. J. Cong and S. K. Lim, "Multiway Partitioning with Pairwise Movement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1998, to appear.
14. J. Darnauer and W. Dai, "A Method for Generating Random Circuits and Its Applications to Routability Measurement", *Proc. ACM/SIGDA International Symposium on FPGAs*, 1996, pp. 66-72.
15. A. Dasdan and C. Aykanat, "Two Novel Multiway Circuit Partitioning Algorithms Using Relaxed Locking", *IEEE Transactions on Computer-Aided Design* 16(2) (1997), pp. 169-178.
16. W. Deng, *personal communication*, July 1998.
17. 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
18. 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
19. S. Dutt and H. Theny, "Partitioning Using Second-Order Information and Stochastic Gain Function", *Proc. IEEE/ACM International Symposium on Physical Design*, 1998, pp. 112-117
20. 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.
21. 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
22. M. Ghose, M. Zubair and C. E. Grosch, "Parallel Partitioning of Sparse Matrices", *Computer Systems Science & Engineering* (1995) 1, pp. 33-40.
23. D. Ghosh, "Synthesis of Equivalence Class Circuit Mutants and Applications to Benchmarking", *summary of presentation at DAC-98 Ph.D. Forum*, June 1998.
24. M. K. Goldberg and M. Burstein, "Heuristic Improvement Technique for Bisection of VLSI Networks", *IEEE Transactions on Computer-Aided Design*, 1983, pp. 122-125.
25. S. Hauck and G. Borriello, "An Evaluation of Bipartitioning Techniques", *IEEE Transactions on Computer-Aided Design* 16(8) (1997), pp. 849-866.

26. 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.
27. B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs", draft, 1995.
28. B. Hendrickson and T. G. Kolda, "Partitioning Rectangular and Structurally Nonsymmetric Sparse Matrices for Parallel Processing", *manuscript*, 1998 (extended version of PARA98 workshop proceedings paper).
29. 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.
`http://vlsicad.cs.ucla.edu/papers/conference/c66.ps`
30. M. Hutton, J. P. Grossman, J. Rose and D. Corneil, "Characterization and Parameterized Random Generation of Digital Circuits", *In Proc. IEEE/ACM Design Automation Conference*, 1996, pp. 94-99.
31. A. B. Kahng, "Futures for Partitioning in Physical design", *Proc. IEEE/ACM International Symposium on Physical Design*, April 1998, pp. 190-193.
`http://vlsicad.cs.ucla.edu/papers/conference/c77.ps`
32. G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning", draft, 1995
33. G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme For Irregular Graphs", Technical Report 95-064, University of Minnesota, Computer Science Department.
34. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design", *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 526-529.
Additional publications and benchmark results for hMetis-1.5 are avilable at `http://www-users.cs.umn.edu/ ~karypis/metis/hmetis/main.html`
35. G. Karypis and V. Kumar, "Multilevel Algorithms for Multi-Constraint Graph Partitioning", Technical Report 98-019, University of Minnesota, Department of Computer Science.
36. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Tech. Journal* 49 (1970), pp. 291-307.
37. C. Kring and A. R. Newton, "A Cell-Replicating Approach to Mincut-Based Circuit Partitioning", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1991, pp. 2-5.
38. B. Krishnamurthy, "An Improved Min-cut Algorithm for Partitioning VLSI Networks", *IEEE Transactions on Computers*, vol. C-33, May 1984, pp. 438-446.
39. B. Landman and R. Russo, "On a Pin Versus Block Relationship for Partitioning of Logic Graphs", *IEEE Transactions on Computers* C-20(12) (1971), pp. 1469-1479.
40. 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.
41. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, 1990.
42. J. Marks, Wheeler Ruml, Stuart M. Shieber and Thomas Ngo, "A seed-Growth Heuristic for Graph Bisection", *Proc. ALEX-98*, also Technical Report, Harvard University, Computer Science.
43. L. Sanchis, "Multiple-way network partitioning with different cost functions", *IEEE Transactions on Computers*, Dec. 1993, vol.42, (no.12):1500-4.
44. P. R. Suaris and G. Kedem, "Quadrisection: A New Approach to Standard Cell Layout", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1987, pp. 474-477.
45. W. Sun and C. Sechen, "Efficient and Effective Placements for Very Large Circuits", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1993, pp. 170-177.
46. Y. C. Wei and C. K. Cheng, "Towards Efficient Design by Ratio-cut Partitioning", *Proc. IEEE International Conference on Computer-Aided Design*, 1989, pp. 298-301.