# Hypergraph Partitioning and Clustering

David A. Papa and Igor L. Markov

University of Michigan, EECS Department, Ann Arbor, MI 48109-2121

## 1 Introduction

A hypergraph is a generalization of a graph wherein edges can connect more than two vertices and are called hyperedges. Just as graphs naturally represent many kinds of information in mathematical and computer science problems, hypergraphs also arise naturally in important practical problems, including circuit layout, Boolean SATisfiability, numerical linear algebra, etc. Given a hypergraph $H$, $k$-*way partitioning* of $H$ assigns vertices of $H$ to $k$ disjoint nonempty partitions. The $k$-way partitioning problem seeks to minimize a given cost function of such an assignment. A standard cost function is *net cut*, which is the number of hyperedges that span more than one partition, or, more generally, the sum of weights of such edges. Constraints are typically imposed on the solution, and make the problem difficult. For example, certain vertices can be fixed in their partitions (fixed constraints) or the total vertex weight in each partition may be limited (balance constraints). With balance constraints, the problem of optimally partitioning a hypergraph is known to be NP-hard [28]. However, since partitioning is critical in several practical applications, heuristic algorithms were developed with near-linear runtime. Such move-based heuristics for $k$-way hypergraph partitioning appear in [46, 27, 14], with refinements given by [47, 58, 32, 49, 24, 10, 20, 35, 41, 25]. The following is an introduction to partitioning formulations and algorithms, centered on the Fiduccia-Mattheyses heuristic [27] and its derivatives.

There are a wide variety of contexts for hypergraph partitioning. Several of them are outlined in Section 2. Each context uses a hypergraph to represent another kind of data structure. The mappings of several mathematical structures to hypergraphs are described below:

**Matrices.** The pattern of non-zero entries of a matrix $\mathbf{A}$ can be represented by a hypergraph whose hyperedges correspond to rows of $\mathbf{A}$ and vertices correspond to the columns of $\mathbf{A}$. Each hyperedge, $e$, will be connected to a vertex, $v$, if $\mathbf{A}_{e,v} \neq 0$. Figure 1 gives an example of a matrix and its corresponding hypergraph.

**Logic Circuits.** Logic circuits are composed of gates (or standard cells) that perform logical operations and connected by metal wires. The same electrical signal may propagate from one gate to several other gates — such a connection is called a net, and can be conveniently represented by a hyperedge. The hypergraph corresponding to a logic circuit directly maps gates to vertices and nets to hyperedges. The dual of this hypergraph is sometimes used as well. In the dual hypergraph, vertices correspond to nets, and hyperedges correspond to gates. An example of a logic circuit and corresponding hypergraph are given in Figure 2.

**Boolean Formulae.** The *conjunctive normal form* (CNF) for Boolean formulae consists of Boolean variables or their complements grouped into *clauses* and combined with the OR operation. All of the clauses are then combined with the AND operation (see Figure 3 for an example). To convert a CNF formula to a hypergraph, the following mapping is used. Each *literal* in the formula (a Boolean variable is a literal and its complement is another literal) maps to one vertex in the hypergraph. Each clause maps to a hyperedge, which connects to the vertices that correspond to the literals in the clause. A Boolean formula in conjunctive normal form is shown in Figure 3 along with the corresponding hypergraph.

The remainder of this survey discusses hypergraph partitioning as illustrated by these three contexts. Section 2 describes how these contexts give rise to practical applications of hypergraph partitioning. Terms used in this survey are defined in Section 3. Section 4 summarizes the various techniques presented in this survey and outlines how scale dictates which technique is most applicable. The Fiduccia-Mattheyses heuristic is described in detail in Section 5 and the Multilevel Fiduccia-Mattheyses extension is discussed in Section 6. Available software and benchmarks are briefly described in Section 7. Sample empirical results are given in the appendix.

## 2 Motivation and Applications

Hypergraph partitioning arises in several practical applications, three of which are outlined below

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

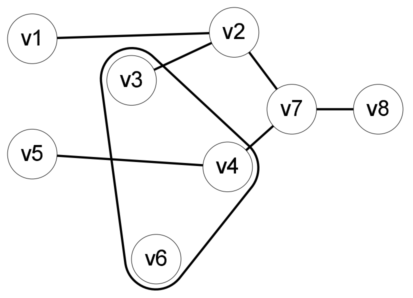


Figure 1: An example of a nearly block-diagonal matrix and corresponding hypergraph. Each row of the matrix corresponds to a hyperedge, and each column corresponds to vertices $v1$ through $v8$. Recursively bisecting the graph aligns the blocks of the matrix on the diagonal.

### 2.1 Numerical Linear Algebra

The runtime of linear algebra computations can vary dramatically depending on the sparsity of input matrices and their patterns of non-zero values [40, 56], which can affect the sparsity of intermediate matrices appearing during computations. In particular, many linear algebra operations (such as matrix-vector multiply, matrix-matrix multiply, solving systems of linear equations, eigenvalue problems, etc.) are faster for block-diagonal matrices. Such operations are easily parallelized because each block is an independent computation. Computing these operations on matrices with permuted rows or columns will give an answer that differs by the same permutation. Therefore the result can be interpreted in the same way, and it is desirable to swap rows in order to bring non-zero elements in input matrices as close to the diagonal as possible before applying such operations. One way to achieve this is through recursive calls to hypergraph partitioning on a hypergraph representation of the matrix. Figure 1 shows a small example of a sparse block-diagonal matrix with its corresponding hypergraph. This permutation on vertices was obtained by recursively partitioning the hypergraph.

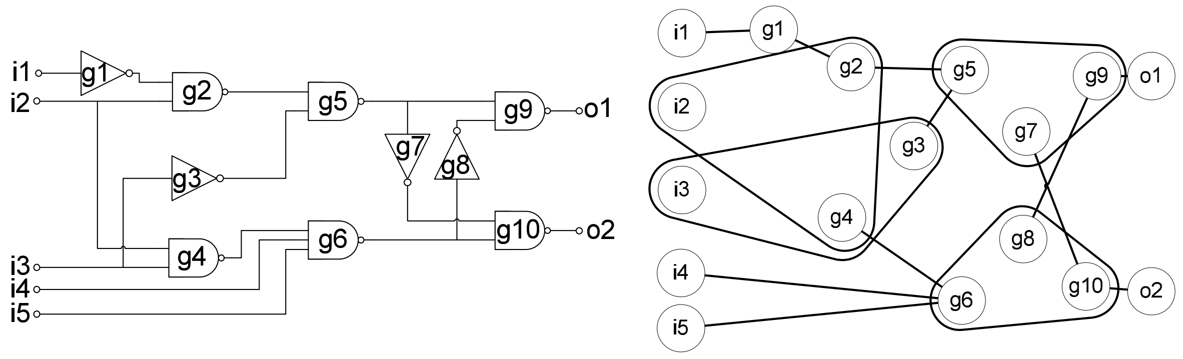


Figure 2: An example of a logic circuit and the corresponding hypergraph.

## 2.2 Integrated Circuit Design

VLSI circuit design has long provided driving applications and ideas for hypergraph partitioning heuristics. For example, the methods of Kernighan-Lin [46] and Fiduccia-Mattheyses [27] form the basis of today's move-based approaches. The method of Goldberg-Burstein [31] presaged the multilevel approaches recently popularized in the parallel simulation [29, 36, 42] and VLSI [9, 41, 10] communities. As noted in [6], 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. Hence, the current practice remains to evaluate new algorithmic ideas against suites of human-designed benchmark instances. In the VLSI partitioning community, performance of algorithms is typically evaluated on the ISPD 1998 benchmarks released by IBM. Alpert [4] noted that previous circuits did not 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 [3].

Salient features of benchmark (real-world) circuit hypergraphs include:

- size: number of vertices can be in the millions (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,
- 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 be highly efficient in order to be useful in VLSI design.[1] Because of this and also because of their flexibility in addressing variant objective functions, fast

[1]For example, a modern top-down standard-cell placement tool will perform recursive min-cut bisection of a gate-level hypergraph to obtain a coarse global placement, which is then refined into a detailed placement by local optimizations. This entire placement process, for example, takes approximately 1 CPU minute per 7000 standard cells on an AMD Opteron 250 work-station with adequate RAM. The implied partitioning runtimes are on the order of 1 CPU second for hypergraphs with 3500 vertices, and 10 minutes for hypergraphs with 2 million vertices.

and high-quality iterative move-based partitioners using the approach of Fiduccia-Mattheyses [27] have dominated recent practice.

The primary use of partitioning heuristics in VLSI design is that of top-down recursive min-cut bisection placement. In this placement framework, a region of a chip is divided geometrically, and the logic inside that region is partitioned topologically. Each of these pieces are then recursively divided until the regions are so small that an optimal end-case placer can solve the problem in a reasonable amount of time.

$$\phi = (v1+v3)(v3+v5)(v2+v5+v4)$$
$$(v5+v6)(v6+v7+v8)(v8+v9)$$

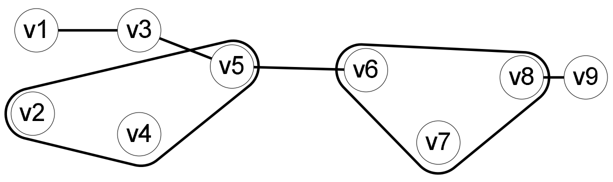


Figure 3: An example of a CNF logic formula $\phi$ and the corresponding hypergraph.

## 2.3 Automated Theorem-Proving and Formal Verification

Algorithms for electronic design automation (EDA) [55, 62], including those for synthesis as well as hardware and software verification, require efficient manipulation of Boolean functions. Boolean satisfiability (SAT) [54, 59] solvers and binary decision diagrams (BDDs) [13] have traditionally been used with such applications, but their worst-case complexity remains exponential and can scarcely be improved.

A key observation is that Boolean functions arising in EDA applications and constraint satisfaction problems possess useful structural properties, e.g., related variables in satisfiability typically participate in the same clauses. Uses of problem structure are known to improve the efficiency of SAT and BDD algorithms. For example, Prasad et al. [57] theoretically show that combinational circuits with small net cuts give easy instances of automatic test pattern generation (ATPG), which are essentially SAT instances. BDDs with smaller cuts tend to have fewer hyperedges and vertices, speeding up BDD manipulations [1, 11].

Based on these observations, the authors of [2] reorder Boolean variables to place "connected" variables close to each other. The ordering process relies on recursive calls to hypergraph partitioning to reduce net cut. This optimization can accelerate SAT solving and BDD manipulation, and reduce BDD memory consumption. The authors of [7, 51] apply partitioning techniques to more general types of reasoning and more sophisticated theorem provers.

## 3 Definitions and Terminology

In what follows, $V$ is the set of vertices in a hypergraph.

**1 Disjoint Partitions:** *A $k$-tuple $\mathbf{P} = (\mathbf{p}_0, ..., \mathbf{p}_{k-1})$ with each $\mathbf{p}_i$ a set of vertices such that $\cup_{i=0}^{k-1}\mathbf{p}_i = \mathbf{V}$ and $\cap_{i=0}^{k-1}\mathbf{p}_i = \emptyset$.*

**2 $k$-way Partitionment:** *A function of the form $\delta : \mathbf{V} \rightarrow \mathbf{P}$ wherein all vertices of $\mathbf{V}$ are mapped to a disjoint partitions from the $k$-tuple $\mathbf{P}$. More practically, this function assigns the vertices to one of $k$ disjoint partitions.*

**3 Balance Constraint:** *A pair of values $(l, u)$. A partition $\mathbf{p}$ with a balance constraint must obey $l \leq \sum_{v \in \mathbf{p}} W(v) \leq u$, where $W(v)$ is the weight of vertex $v$ or 1 if the vertex has no weight.*

When $\forall v \in V, W(v) = 1$, then $\sum_{v \in \mathbf{p}} W(v) = |\mathbf{p}|$, the number of vertices in $\mathbf{p}$. Partitioning using similar balance constraints for all partitions will equalize the number of vertices in each.

**4 Hypergraph:** *A pair of sets $\mathbf{H} = (\mathbf{V}, \mathbf{E})$. $\mathbf{V}$ is the set of* vertices *of the hypergraph and $\mathbf{E}$ is the set of* hyperedges *of the hypergraph. Each hyperedge in a hypergraph is a non-empty subset of $\mathbf{V}$, the size of this subset is called the hyperedge's* degree. *A* weighted hypergraph *has non-negative numeric weights associated with each vertex, each hyperedge, or both.*

Conventional graphs are a special case of hypergraphs, where all hyperedges have degree 2. In most cases, the degree of a hyperedge is no smaller than 2.

Vertices and hyperedges optionally have non-negative numeric weights. A weight of 0 usually means that the edge or the vertex is deleted. Weights usually have additive semantics, e.g., two weighted edges $e_1$ and $e_2$ connecting the same pair of vertices can be replaced with a single edge $e_3$ such that $W(e_3) = W(e_1) + W(e_2)$ (see also Definition 8).

**5 Cut:** *A hyperedge* $\mathbf{e}$ *of hypergraph is cut if, with respect to a particular partitionment* $\delta$*, its vertices are mapped to more than one partition. The net cut of a partitionment is the total number of hyperedges that are cut. The* weighted net cut *of a partitionment is the sum of the weights of hyperedges that are cut.*

**6 Hypergraph Partitioning:** *The process of finding a partitionment of a hypergraph such that some cost function, such as net cut, is minimized. When the solutions must additionally satisfy balance constraints, the process is called* balanced hypergraph partitioning. *Hypergraph partitioning that results in two partitions is called* bisection.

**7 The $k$-way Hypergraph Partitioning Problem:** *Given a hypergraph* $\mathbf{H} = (\mathbf{V}, \mathbf{E})$*, find a* $k$*-way partitionment* $\delta : \mathbf{V} \rightarrow \mathbf{P}$ *that maps the vertices of* $\mathbf{H}$ *to one of* $k$ *disjoint partitions such that some cost function* $c : \delta \rightarrow \mathbb{R}$ *is minimized.*

One typically deals with a particular hypergraph partitioning problem *instance* which consists of a particular hypergraph, a set of two or more balance constraints, and an objective function.

**8 Clustering:** *The process of computing a coarser hypergraph from an input hypergraph by merging vertices into larger groups of vertices called* clusters.

The weight of each cluster will be the sum of the weights of its vertices, or simply the number of vertices if they have no weights.

Several cost objectives exist for the hypergraph partitioning problem. The most common by far is net cut. In VLSI placement, reduced net cut is correlated with shorter wires, and in parallelization smaller net cut means reduced interprocessor communication. For more than 2-way partitioning, the sum-of-degrees metric is sometimes used. For this metric, the cost of each hyperedge is equal to the number of different partitions which contain some of its vertices.

## 4 Summary of Techniques and Their Scale Dependence

In many applications of hypergraph partitioning, the size of input grows every year, and the demand for performance is high. For example, the number of transistors in a typical VLSI design continues to grow exponentially (according to Moore's Law). The algorithms applicable to these circuits must scale to tens of millions of components today and hundreds of millions in the foreseeable future. Because of these large inputs, any partitioning technique used must have near-linear complexity in the worst case in order to be effective.

State-of-the-art partitioning tools use local search heuristics to refine a given partitionment. The basic technique common to all VLSI partitioning applications is the Fiduccia-Mattheyses (FM) algorithm [27], which applies linear-time *passes* to iteratively improve a given partitionment by moving every vertex exactly once. A prevalent extension of this algorithm is the Multi-level FM (MLFM) algorithm, which improves both solution quality and runtime of partitioning large hypergraphs by *clustering* tightly connected components and partitioning the resulting smaller hypergraph. However, for sufficiently small hypergraphs, "flat" FM can produce optimal solutions faster than MLFM, indicating that different techniques should be applied at different scales.

Despite growing input sizes, recursive applications of partitioning will still produce a large number of small partitioning instances, and the performance of partitioning on small instances is equally relevant in practice. In high-performance applications, actual runtime, rather than asymptotic complexity, is a design objective of prime importance. Due to this, occasionally an algorithm with higher complexity will be selected if it is faster for practical input sizes. The following is a summary of the techniques available, and at which scale they are effective.

## 4.1 Exhaustive Enumeration

Exhaustive enumeration produces optimal partitionments and has exponential asymptotic complexity but low constant runtime factor. As such, at very small scales, exhaustive enumeration is the fastest known technique. The evaluation of each solution is the bottleneck of this algorithm, and it can be sped up by incrementally evaluating the cost objective. However, it is straightforward to iteratively update the cut of a partitionment when one vertex is moved [15]. We can represent each partitionment as a vector, with each entry corresponding to a vertex's partition. Then, enumerating the values of this vector in Gray code order moves only one vertex at a time, thus allowing incremental update of net cut for fast evaluation. Empirically, we find this to be the fastest technique for one to nine vertices [15].

## 4.2 Branch and Bound

The runtime of exhaustive enumeration grows exponentially in the number of vertices and thus cannot scale very far. However, its scalability can be improved through intelligent search space pruning; this technique is known as branch and bound (B&B). A B&B implementation does a depth-first search of the tree of partial partitionments (some vertices assigned to a partition, some yet unassigned) by choosing a partition for the next unassigned vertex and recursing. It can search the entire solution space in the worst case and therefore also has exponential time complexity, but maintains optimality by bounding away only suboptimal results. Partition balance constraints can be used for pruning illegal solutions and best-seen cost can be used to bound away suboptimal results.

One notable improvement to the bounding function is known as *inevitable cut*[15], which applies to the portion of a hypergraph that is made up of two-pin nets. Choosing *any* partition

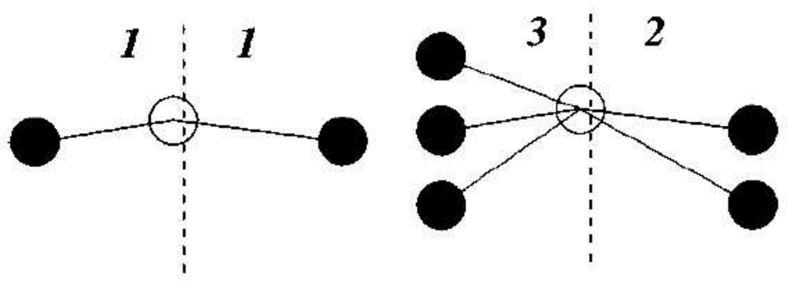


Figure 4: Two small examples of an inevitable cut computation. The black vertices have been assigned to their respective partitions and the white vertex is still unassigned. In each case, any assignment of the white vertex implies that some edges will be cut. The inevitable cut is 1 on the left and 2 on the right. For a given unassigned vertex, inevitable cut is computed as the smallest number of adjacent vertices assigned to any one partition.

for a particular vertex may imply some additional cost, because it may be connected to vertices in both (all) partitions. The minimum possible additional cost (considering only two-pin nets) over all legal partition assignments for a particular vertex is known as its *inevitable cut*, and can be computed directly. It is the minimum number of two-pin connections to vertices locked in any particular partition. Figure 4 shows two small examples of an inevitable cut computation (for more details see [15]). The inevitable cut of a vertex can be safely added to the lower-bound cost computation in the bounding function to strengthen the pruning of suboptimal results. It is unclear how to extend this technique to multi-pin nets. The inevitable cut technique can be extended to handle a larger portion of the hypergraph by replacing three-pin nets with three-cliques (triangles) with each hyperedge's weight multiplied by one-half the weight of the original three-pin hyperedge. This "triangle technique" preserves cost of all partitionments exactly and allows inevitable cut to be applied to all two and three-pin nets. The runtime of B&B can be unreasonably high for certain pathological cases that have many optimal solutions. In practice we detect these cases with a time-out. If the runtime limit is exceeded we will stop B&B and use a more scalable algorithm such as Fiduccia-Mattheyses discussed below. Empirically, we find B&B to be the best technique for ten to thirty-five movable objects [15].

## 4.3 The Fiduccia-Mattheyses Heuristic

Partitioning problems with more than thirty-five movable objects take an impractical amount of runtime to solve optimally using B&B. Fortunately, an amortized near-linear-time heuristic exists for iterative improvement of hypergraph partitions. The Fiduccia-Mattheyses (FM) algorithm works by prioritizing *moves* by *gain*. A *move* changes to which partition a particular vertex belongs, and the *gain* is the corresponding change to the cost function. After each vertex is moved, gains for connected modules are updated.

The FM algorithm runs in *passes* wherein each vertex is moved exactly once. Passes are generally applied until little or no improvement remains. Initial solutions are often produced using a simple randomized algorithm. Empirically we find FM to be the best technique for thirty-six to two hundred movable objects. Details of the FM algorithm are discussed below and, at length in Section 5.

## 4.4 Multilevel Fiduccia-Mattheyses Framework

The multilevel hypergraph partitioning framework provides the best known partitioning results for large-scale circuit hypergraphs. It consists of three main components: clustering, top-level partitioning and refinement. During clustering, hypergraph vertices are combined into clusters based on connectivity, leading to a smaller, clustered hypergraph. This step is repeated until the hypergraph is small enough to be solved effectively by a “flat” partitioning algorithm (e.g., the FM heuristic). The smallest hypergraph is partitioned with a very fast initial solution generator (e.g., random) and iteratively improved using the flat partitioner. The resulting partitionment is then interpreted as a solution for the next (less clustered) hypergraph during the refinement stage, and improved again using the flat partitioning algorithm until the bottom level is reached.

Using this multilevel framework with the FM algorithm is known as the Multilevel Fiduccia-Mattheyses (MLFM) technique, and it is the most effective and commonly used hypergraph partitioning algorithm for large circuit hypergraphs. Empirically we find the MLFM algorithm to be the best technique for more than 200 movable objects. Details of the MLFM algorithm are discussed at length in Section 6.

### 4.5 Other Types of Algorithms

Several other approaches exist for solving the hypergraph partitioning problem [19, 23, 26, 37, 63]. Typically, these techniques have some substantial drawback that makes their use impractical for high-performance applications. For example, meta-heuristics such as simulated annealing and tabu search may be able to achieve better solution quality at the cost of an impractical increase in runtime [26]. Other techniques may have constraints on their input that are violated by some problem instances, as is the case with spectral techniques [23]. Spectral algorithms find eigenvalues of the Laplacian matrix of the connectivity graph and derive a partitionment from coefficients of an eigenvalue, e.g., by comparing them to the median [5]. These algorithms may not handle fixed terminals well and are therefore not general enough to handle many practical applications. Another technique relies on the Min-cut Max-flow theorem and efficient network-flow algorithms to identify optimally small cuts in graphs [63]. These polynomial-time algorithms do not typically perform approximation and can handle hyperedges using accurate conversion to graphs, but do not handle balance constraints, which results in expensive trial-and-error in flow-based partitioners. Work by Hur and Lillis [37, 38] applies similar techniques to placement, in a somewhat different setting, and describes an incremental flow-solver that sweeps through a series of input configurations, however their placer is still fairly slow.

## 5 The Fiduccia-Mattheyses Heuristic

The Fiduccia-Mattheyses (FM) heuristic for partitioning hypergraphs [27] is an iterative improvement algorithm. Its neighborhood structure is induced by single-vertex, partition-to-partition moves.[2] 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 to the 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. Pseudo-code for the FM heuristic is given in Figure 5.

The FM algorithm has three main components (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. One contribution of Fiduccia and Mattheyses lies in observing that circuit hypergraphs are sparse, and any move's gain is bounded between plus and minus the maximal vertex degree in the hypergraph (times the maximal hyperedge weight, if weights are used). This allows prioritization of moves by their gains. All affected gains can be updated in amortized-constant time, giving overall linear complexity per pass[27]. In [27] all moves with the same gain are stored in a linked list representing a "gain bucket". It is important to note that some gains may be negative, and as such, FM performs hill-climbing and is not purely greedy.

## 5.1 FM Passes

The Fiduccia-Mattheyses (FM) algorithm consists of incrementally-improving passes [27]. During each pass, FM will search the neighborhood of a given partitionment, and record the

[2]By contrast, the stronger Kernighan-Lin (KL) heuristic [46] uses a pair-swap neighborhood structure and has cubic runtime per pass.

```
FM(hypergraph, partitionment)
    do
        initialize gain_container from partitionment;
        FMpass(gain_container, partitionment);
    while(solution quality improves);
FMpass (gain_container, partitionment)
    solution_cost = partitionment.get_cost();
    while(not all vertices locked)
        move = choose_move();
        solution_cost += gain_container.get_gain(move);
        gain_container.lock_vertex( move.vertex() );
        gain_update(move);
        partitionment.apply(move);
    roll back partitionment to best seen solution;
    gain_container.unlock_all();
```

Figure 5: Pseudo-code for the FM heuristic. choose_move and gain_update are defined in Figure 8

best seen solution. FM performs successive passes as long as the solution can be improved.

At each pass, the FM repetitively chooses one (best) move and applies it, followed by the processing of 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. FM will then undo a given number of moves to return to a solution with best-seen cost.

## 5.2 Gain Computation and Gain Update

The initialization of the FM data structures at the beginning of each pass is straightforward. First traverse all hyperedges and count the number of vertices in each partition. If a hyperedge is uncut, then decrease the gain of all adjacent vertices by the hyperedge weight (or by -1 when no weights are given). If there is only one vertex in some partition, then increase the gain of that vertex by the hyperedge weight (or by 1 when no weights are given) and do not change the gain of all other adjacent vertices.

Picking and applying one move is subtle. FM requests the best move from the gain container and can continue requesting more moves until a feasible (i.e., not violating the balance constraints) move is found. As FM applies the chosen move and locks the vertex, gains of adjacent vertices likely need to be updated. In performing "generic" gain update, an implementation of FM must walk all hyperedges incident to the moving vertex and for each hyperedge computes gain updates (*delta gains*) for each of its vertices due to this hyperedge (these are combinations of the given hyperedge's cost under four distinct partition assignments for the moving and affected vertices; see Figures 8 and 13). These partial gain updates are immediately applied to the gain container, 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. The net cut objective is particularly amenable to optimizations during gain update [58, 27, 18]. For example, hyperedges that have at least one vertex locked in each partition can be safely ignored during gain update, as the cost of such a hyperedge cannot change until the end of a pass, and they do not contribute to gain update. We term such hyperedges *locked*. Once a hyperedge becomes locked, it is marked as such in a dedicated bitvector. While this requires additional effort, the savings in the overall runtime per pass are considerable.

## 5.3 Custom Data Structures

The efficient custom data structures used by the FM algorithm are critical to its performance. The *gain bucket list* data structure introduced in [27] is necessary to allow linear-time gain update. Figure 6 shows the gain bucket list structure as originally illustrated. On the left is a *bucket-array* structure which stores $2 * pmax + 1$ pointers to gain buckets, where $pmax$ is the maximum possible gain. The bucket-array allows constant-time lookup of moves which have

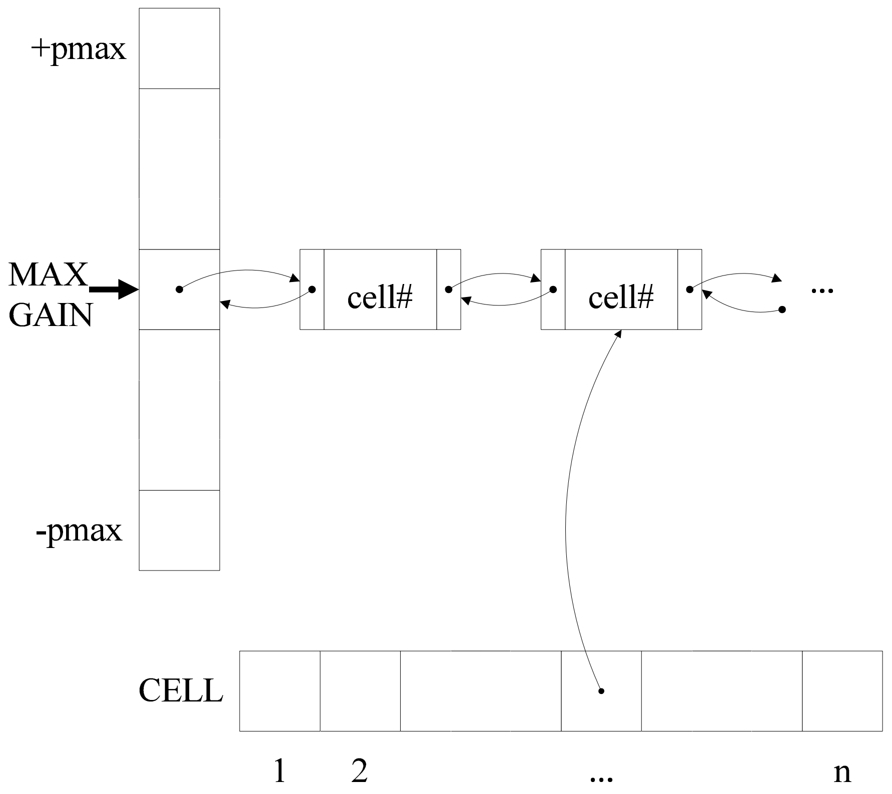
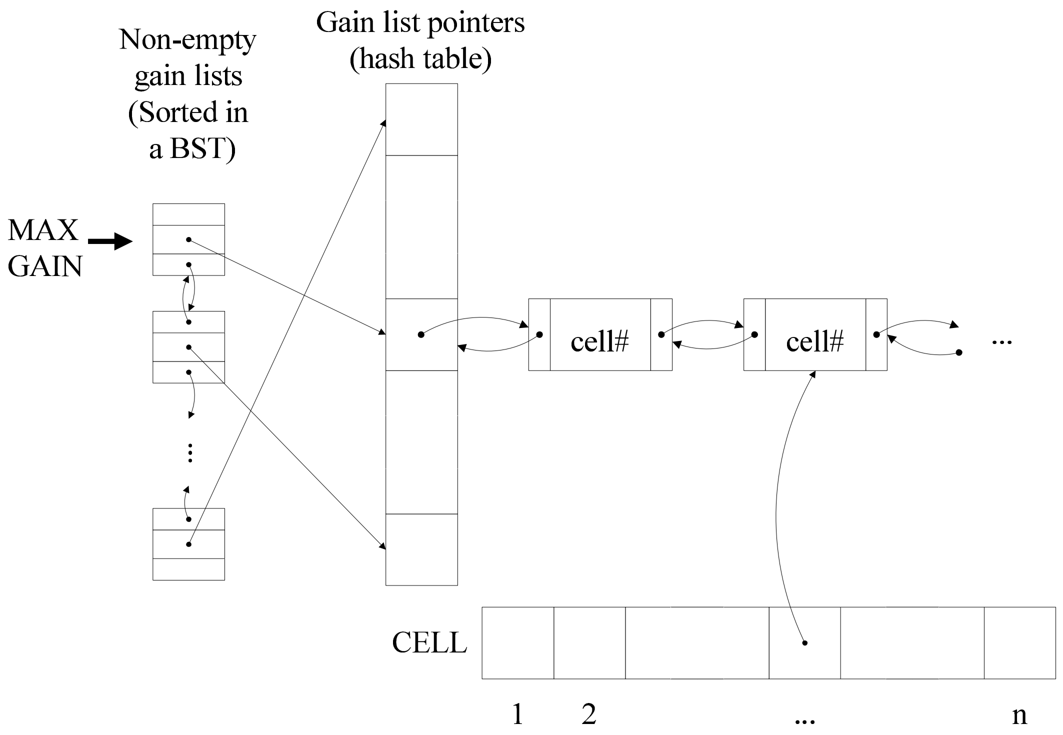


Figure 6: The gain bucket list structure as illustrated in [27]. More efficient implementations than what is depicted exist.

a particular gain. Each bucket is a possibly empty doubly-linked list of *gain elements*. On the bottom is a *repository* which stores pointers to the gain elements associated with each vertex. The repository allows constant time lookup of the gain for a particular vertex. To efficiently find the move with max-gain, the index of the highest-gain, non-empty bucket is maintained.

The bucket-array structure is efficiently implemented with an array when the magnitude of hyperedge weights is limited by some constant. However, an array- based implementation of the bucket-array has space complexity dependent upon the magnitude of hyperedge weights. For arbitrary hyperedge weights, the bucket-array must be modified in order to efficiently support the operations of a bucket-array in the context of an FM gain container. These operations include (1) finding a particular gain bucket; (2) insertion and removal of gain elements; (3) finding the maximum non-empty gain bucket and (4) finding the second-highest non-empty gain


Non-empty gain lists (Sorted in a BST)
Gain list pointers (hash table)
MAX GAIN
cell#
cell#
...
CELL
1
2
...
n
Figure 7: The gain list structure with a hybrid BST+hash-table bucket-array for improved FM complexity with arbitrary hyperedge weights.

bucket. The last operation is necessary, for example, when the max-gain bucket is exhausted and the new max-gain must be determined. If hyperedge weights are large, then an array-based implementation of bucket-array will result in a large, sparse and wasteful structure that must be searched (linear in the magnitude of weights) to implement operation (4).

Figure 7 shows an alternative implementation of bucket-array that allows for time complexities of operations (1), (3) and (4) that do not depend on the magnitude or sparsity of hyperedge weights. Operation (2) is logarithmic in the number of non-empty gain buckets. On the right is a hash table that stores pointers to gain buckets. Since the expected lookup time for a hash table is constant, this efficiently implements operation (1). On the left is a binary search tree (BST)[3] that stores the gains which are possible from some move (i.e., non-empty gain lists). Since a BST is sorted, the largest item in the BST can be found in constant time, and this is

[3]We assume an implementation of a binary search tree that allows constant time traversal of the sorted sequence. The standard C++ set is an example of such a data structure.

an efficient implementation of operation (3). Operation (4) is as easy as operation (3) for this structure, since the second-highest gain is the second from the last entry in the non-empty gain BST. The most difficult operation for this structure is operation (2), since BST insertion and removal are logarithmic time operations. Nevertheless, the complexity of operation (4) for an array-based implementation is linear in the magnitude of hyperedge weights. Therefore, for sufficiently large weights, the runtime of this BST+hash bucket-array will be less than that of an array-based implementation of bucket-array.

## 5.4 Implementation Insights

There are important degrees of freedom in the implementation of the above gain containers which can have a significant effect on solution quality. Among the most notable is whether items are taken from the gain buckets in Last-In-First-Out (LIFO) or First-In-First-Out (FIFO) order. It has been shown that choosing LIFO order gain buckets can provide significant improvement in solution quality [32]. The intuition for this effect is that it exploits some locality in the structure of the hypergraph. When moving one vertex to a different partition causes a change to the gain of another vertex, that other vertex should also be considered for movement.

Extending the theme of exploiting the locality in graphs, Dutt and Deng [24] propose the CLuster-oriented Iterative-improvement Partitioner (CLIP) algorithm. The key observation in this algorithm is that if tightly-connected cluster is cut, then the solution cost can be reduced by moving the cluster entirely into a single partition. CLIP first computes the gains of all vertices. It then sorts them into a single linear order and puts them all into the zero-gain bucket. This allows for choosing the highest gains first. More importantly, as partitioning proceeds, the gain of each move will be determined solely by the change in gain due to moves of adjacent vertices. Thus, when part of a cluster is moved, it is more likely that the rest of the cluster will be moved along with it. CLIP does not track net cut as accurately as conventional FM, and therefore it usually does not improve initial solutions that are already good. CLIP is used in

```
choose_move (gain_container)
    move = gain_container.max_feasible_move();
    while(move is infeasible)
        gain_container.mark_infeasible(move);
        move = gain_container.max_feasible_move();
    gain_container.mark_all_feasible();
    return move;
gain_update (move)
    source_part = partition that move.vertex() is in;
    dest_part = partition where move.vertex() is going;
    for_each(hyperedge e incident to move.vertex())
        if(e has no vertices in dest_part before applying move)
            for_each(vertex v on e)
                gain_container.update(v, dest_part, e.weight());
        if(only 1 vertex on e in source_part before applying move)
            for_each(vertex v on e)
                gain_container.update(v, source_part, -e.weight());
        if(only 1 vertex v remains in source_part after applying move)
            gain_container.update(v, dest_part, e.weight());
        if(only 1 vertex v in dest_part before applying move)
            gain_container.update(v, source_part, -e.weight());
```

Figure 8: Functions called by the FM heuristic. A faster version of gain_update that takes advantage of special cases is given in Figure 13.

non-incremental contexts, starting with a random initial solution and is typically postprocessed by conventional FM passes.

FM also tends to have problems when the highest-gain move is a vertex with high weight that cannot move across the cutline due to balance constraints. This is known as the *corking effect* and some techniques for handling it are presented in [17]. Techniques specific to hypergraphs with fixed vertices are presented in [16].

## 6 Multilevel Fiduccia-Mattheyses Partitioning Framework

The multilevel hypergraph partitioning framework was successfully verified in 1997 by [10, 41, 43] and has been conducive to the best known known partitioning results ever since. The main advantage that MLFM has over flat partitioners is its ability to more effectively search the solution space by spending comparatively more effort on smaller coarsened hypergraphs. Good coarsening algorithms allow for high correlation between good partitionments for coarsened hypergraphs and good partitionments for the initial hypergraph. Therefore, a thorough search at the top of the multilevel hierarchy is worthwhile because doing so is relatively inexpensive when compared to flat partitioning of the original hypergraph, but can still preserve most of the possible improvement. The result is an algorithmic framework with both improved runtime and solution quality over a completely flat approach. Pseudo-code for an implementation of the MLFM framework is given in Figure 9.

Multilevel partitioning consists of three main components: clustering, top-level partitioning and refinement or "uncoarsening." During clustering, hypergraph vertices are combined into clusters based on 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. We describe this hierarchy with the smaller hypergraphs being "higher" and the larger hypergraphs being "lower."[4] The smallest (top-level) hypergraph is partitioned with a very fast initial solution generator and iteratively improved, for example, using the FM algorithm. The resulting partitionment is then interpreted as a solution for the next hypergraph in the hierarchy. During the refinement stage, solutions are projected from one level to the next and iteratively improved, for example, by the FM algorithm.

Additionally, the hMETIS partitioning program [45] 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 vertices 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 – vertices in different partitions cannot be

[4]This is the most common notation used for a multilevel partitioning hierarchy. hMETIS related works invert the notation, including the naming of V-cycles which may be more appropriately called $\Lambda$-cycles (lambda-cycles).

clustered. A similar technique is v-cycling, in which the refinement stage may stop before the bottom-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.

## 6.1 Coarsening

The multilevel partitioning framework begins by "coarsening" the input hypergraph which results in smaller hypergraphs that retain as much of the structure of the original hypergraph as possible. Solutions to these coarsened hypergraphs are then interpreted as solutions to larger hypergraphs during refinement. "Clustering," the name for merging two or more vertices together, is the mechanism by which hypergraphs are coarsened. Here we cover the details of two simple but effective clustering algorithms. A countless number of more complex schemes exist in partitioning literature [34, 20, 21, 44, 33].

### 6.1.1 Edge Coarsening

A simple linear-time clustering strategy called (hyper)Edge Coarsening (EC) was proposed in [10, 41]. The EC technique works by combining connected vertices and is commonly randomized by choosing a vertex and a random neighbor to merge. A straightforward but effective EC implementation has the following attributes [16]:

- the hypergraph 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,
- hyperedge 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 1:2.8,
- clustering stops when the clustered hypergraph has 200 clusters or fewer.

### 6.1.2 Heavy Edge Matching

Heavy (hyper)Edge Matching (HEM) is an alternative clustering strategy which seeks to minimize the net cut of the coarsened hypergraph directly. This is achieved by contracting hyperedges with the highest possible weights. In the hypergraph version of this strategy, a vertex is chosen and it will be probabilistically merged with its "nearest neighbor," defined as that with the largest weight connection. The net cut (and partitioning runtime) of a coarsened hypergraph can also be improved when nets disappear entirely due to all of its connected vertices being merged into one. Since it is more likely that a smaller net will be removed, HEM weights connections via smaller nets more highly. A common weighting scheme $W$ is $W(e) = \frac{1}{degree-1}$ for hyperedge $e$, but many variations of this are possible. HEM also seeks to minimize the number of remaining nets by favoring neighbors which have multiple connections via several nets. So the weight of a connection $(u, v)$ will be $\sum_{e \in E}(W(e)$ if $u, v \in e$ or 0 otherwise).

## 6.2 Top-level Partitioning

The FM algorithm is only capable of iteratively improving an initial solution. At the top-level no previous solution is available to improve, therefore top-level partitioning requires initial solution generation. A rather simple generator will suffice, because the FM algorithm will quickly find a nearby local minimum. The following "randomized engineering method" (REM) creates random, legal initial solutions. REM first sorts vertices according to decreasing size, so that it can assign the largest vertices first to satisfy balance constraints and avoid "painting itself into a corner." It assigns vertices to partitions in sorted order using biased random selection ("spinning a roulette wheel" such that each outcome has a prescribed probability). Assignment probabilities are proportional to the hypothetical area remaining before the solution will satisfy minimal area requirements, the *area slack*, computed after assigning a given vertex to the various partitions. This keeps area slacks approximately equal, yet provides a good degree of randomness. REM will continue assigning vertices to partitions in this way until all partitions reach the lower bound of their respective balance constraints, at which point it will compute slacks relative to the upper bound.

### 6.3 Refinement

After a partitionment is chosen at a particular level, it is projected onto the less clustered (finer) hypergraph at the next level. A vertex in the finer hypergraph is projected into the same partition as the cluster it belongs to in the coarsened hypergraph. The solution for this hypergraph will have cut identical to that for the more clustered hypergraph. However, it is likely that the solution can be improved with respect to the finer hypergraph. This is done through iterative improvement by calling FM on the partitioning solutions at each level.

## 7 Available Software and Benchmarks

There are two common, freely-available academic tools for performing hypergraph partitioning. MLPart [53] is an open source C++ implementation of MLFM hypergraph partitioning geared toward circuit hypergraphs and partitioning based placement. hMETIS [52] is the hypergraph version of METIS, a multilevel graph partitioning algorithm implemented in C. hMETIS is distributed freely for academic use in the form of a precompiled library and executables.

These two partitioning tools use different file formats to represent hypergraphs. hMETIS uses one text file to represent a hypergraph, while MLPart uses several text files in the Bookshelf format to represent partitioning problems, which includes a hypergraph.[5] Both tools additionally define balance constraints in terms of partition capacity targets, and a tolerance parameter

[5]In addition, industrial VLSI design tools commonly use LEF/DEF files to represent hypergraphs. Tools exist to convert LEF/DEF to the Bookshelf format. Please see [50, 61] for more information.

```
MLFM(hypergraph)
     level = 0;
     hierarchy[level] = hypergraph;

//Coarsening phase
     while(hierarchy[level].vertex_count() > 200)
         next_level = cluster(hierarchy[level]);
         level = level + 1;
         hierarchy[level] = next_level;

//Top level partitioning phase
     partitionment[level]
            = a random initial partitionment for top-level hypergraph;
     FM(hierarchy[level], partitionment[level])

//Refinement phase
     while(level > 0)
         level = level - 1;
         partitionment[level]
            = project(partitionment[level+1], hierarchy[level]);
         FM(hierarchy[level], partitionment[level])

     return partitionment[0];
```

Figure 9: Pseudo-code for the MLFM algorithm.

that specifies how far it is allowed to deviate from its target.

## 7.1 hMETIS Benchmark Format

The input hypergraph to hMETIS has a simpler format than Bookshelf. The first line contains the number of hyperedges, the number of vertices, and an optional format bit-flag that indicates the presence of weights for hyperedges, vertices or both. If the vertices have weights, the file will have $|E| + |V| + 1$ lines, and $|E| + 1$ lines otherwise. The file may additionally contain commented lines preceded by $\%$. Each of the following $|E|$ lines represents one hyperedge, and will optionally contain an integer weight, followed by a list of the indices of vertices on that hyperedge. If vertex weights are present, the remaining $|V|$ lines will contain the integer

| Line no. | Graph file | Solution file |
|---|---|---|
| 1 | 7 8 | 1 |
| 2 | 1 2 | 1 |
| 3 | 5 4 | 0 |
| 4 | 3 4 6 | 0 |
| 5 | 2 3 | 0 |
| 6 | 4 7 | 0 |
| 7 | 2 7 | 1 |
| 8 | 7 8 | 1 |

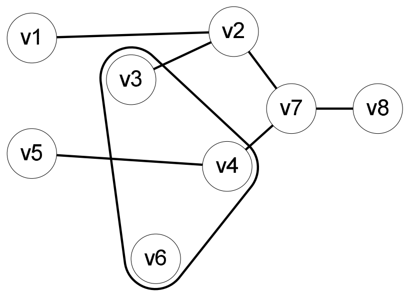


Figure 10: An example of an hMETIS graph file and partitioning solution for the hypergraph on the right. Line 1 of the graph file specifies that there are 7 hyperedges and 8 vertices. The remaining lines specify hyperedges. The $i^{th}$ line of the solution file specifies the partition of the $i^{th}$ vertex.

weight of each vertex. The solution is specified by giving the partition of the $i^{th}$ vertex on the $i^{th}$ line of the solution file. Partitioning tolerance is specified with a parameter called *UBfactor* that can be specified on the command line or when making calls to the hMETIS library. This number specifies how far any partition's total weight can deviate from the average weight of a partition, as a percentage. For more details and examples, please see Figure 10 and [45].

## 7.2 Bookshelf Benchmark Format

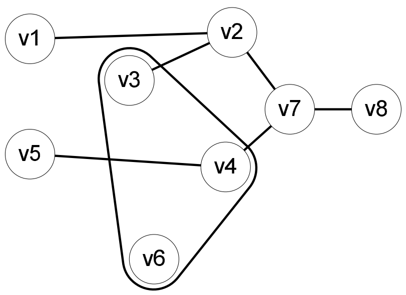
The Bookshelf benchmark format is more expressive because it is used in other VLSI applications than hypergraph partitioning, including placement and floorplanning. Each Bookshelf partitioning benchmark will be represented with several files, at minimum: a .nodes file containing a list of vertices and their sizes, a .nets file containing a list of nets (hyperedges) and the vertices they connect to, a .blk file specifying partitions and their balance constraints, and a .aux file containing one line listing all of the other filenames. Balance constraints are specified by a *target* for each partition, and one *capacity tolerance* as a percent. The balance constraint of a particular partition will be $(target * (1 - capacity\_tolerance/100), target * (1 + capacity\_target/100))$. Other optional files may be included, namely: a .wts listing vertices

| Line no. | example.aux |
|---|---|
| 1 | `PartProb :  example.nodes example.nets example.wts example.blk` |

| Line no. | example.nodes |
|---|---|
| 1 | `UCLA nodes 1.0` |
| 2 | `NumNodes :  8` |
| 3 | `NumTerminals :  0` |
| 4 | `v1` |
| 5 | `v2` |
| 6 | `v3` |
| 7 | `v4` |
| 8 | `v5` |
| 9 | `v6` |
| 10 | `v7` |
| 11 | `v8` |

| Line no. | example.wts |
|---|---|
| 1 | `UCLA wts 1.0` |
| 2 | `v1 1` |
| 3 | `v2 1` |
| 4 | `v3 1` |
| 5 | `v4 1` |
| 6 | `v5 1` |
| 7 | `v6 1` |
| 8 | `v7 1` |
| 9 | `v8 1` |

| Line no. | example.blk |
|---|---|
| 1 | `UCLA blk 1.0` |
| 2 | `Regular partitions :  2` |
| 3 | `Pad partitions :  0` |
| 4 | `Relative capacities :  no` |
| 5 | `Capacity tolerances :  25%` |
| 6 | `b0 rect 0 0 2 3 :  4` |
| 7 | `b1 rect 2 0 4 3 :  4` |



| Line no. | example.nets |
|---|---|
| 1 | `UCLA nets 1.0` |
| 2 | `NumNets :  7` |
| 3 | `NumPins :  15` |
| 4 | `NetDegree :  2` |
| 5 | `v1 B` |
| 6 | `v2 B` |
| 7 | `NetDegree :  2` |
| 8 | `v4 B` |
| 9 | `v5 B` |
| 10 | `NetDegree :  3` |
| 11 | `v3 B` |
| 12 | `v4 B` |
| 13 | `v6 B` |
| 14 | `NetDegree :  2` |
| 15 | `v2 B` |
| 16 | `v3 B` |
| 17 | `NetDegree :  2` |
| 18 | `v4 B` |
| 19 | `v7 B` |
| 20 | `NetDegree :  2` |
| 21 | `v2 B` |
| 22 | `v7 B` |
| 23 | `NetDegree :  2` |
| 24 | `v7 B` |
| 25 | `v8 B` |

| Line no. | example.sol |
|---|---|
| 1 | `UCLA sol 1.0` |
| 2 | `Regular partitions :  2` |
| 3 | `Pad partitions :  0` |
| 4 | `Fixed Pads :  0` |
| 5 | `Fixed NonPads :  8` |
| 6 | `v1 :  b1` |
| 7 | `v2 :  b1` |
| 8 | `v3 :  b0` |
| 9 | `v4 :  b0` |
| 10 | `v5 :  b0` |
| 11 | `v6 :  b0` |
| 12 | `v7 :  b1` |
| 13 | `v8 :  b1` |

Figure 11: An example of a Bookshelf format partitioning problem for the hypergraph above. example.aux is a list of all of the other files. NumPins on line 3 of example.nets specifies the sum of hyperedge degrees. example.blk specifies balance constraints, the sum of vertex weights in each partition must be $4 \pm 25\%$. An example solution is given in example.sol.

and nets with one or more weights, a .fix file which lists vertices and which partitions they are constrained to, and a .sol file which contains an input partitionment. All bookshelf benchmarks may contain commented lines beginning with '#'. Solutions are specified by listing all vertices and their partition assignments.

## 7.3 Integrated Circuit Benchmarks from IBM

A set of standard benchmarks derived from VLSI circuits at IBM were presented in ISPD in 1998 [3]. These instances range from 12506 vertices in IBM01 to 210341 vertices in IBM18.

Additionally, partitioning in the context of VLSI placement results in hypergraphs with fixed vertices. Benchmark hypergraphs with fixed vertices were released at ISPD in 1999 [8]. Techniques for evaluating partitioning heuristics and experiments using these benchmarks were given in [17]. Examples of such comparisons between hMETIS and MLPart are found in [16].

# Appendix: Empirical Results

Table 12 gives runtimes and average solution qualities for 10 runs of MLPart and hMETIS on the ISPD '98 IBM benchmark suite with partitioning tolerances of 2 and 10%. Smaller net cut is better. Runs were performed by a 2.0GHz Pentium IV Xeon workstation with 2GB of RAM running Linux.

The size of benchmarks range from 12506 vertices in IBM01 to 210341 vertices in IBM18. MLPart is generally faster than hMETIS, but hMETIS often produces better partitioning results. Both tools produce better solutions when the tolerance is higher. By curve-fitting this data, we find empirically that the runtime and memory usage of both partitioners grows nearly linearly with the size of the benchmark.

We estimate that one can partition a 3.5 million vertex hypergraph in 20 minutes on a 32-bit machine with the above processor and 4GB of memory.

## References

[1] F. A. Aloul, I. L. Markov and K. A. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *ICCAD*, 2001, pp. 443-448.

[2] F. A. Aloul, I. L. Markov and K. A. Sakallah, "MINCE: A Static Global Variable-Ordering for SAT Search and BDD Manipulation", *Journal of Universal Computer Science*, vol. 10, no. 12, pp. 1559-1562, December 2004.

[3] C. J. Alpert, "Partitioning Benchmarks for the VLSI CAD Community," `http://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html`.

[4] C. J. Alpert, "The ISPD-98 Circuit Benchmark Suite," *ISPD*, 1998, pp. 80-85. See errata at `http://vlsicad.ucsd.edu/UCLAWeb/cheese/errata.html`.

[5] C. J. Alpert and A. B. Kahng, "Multi-Way Partitioning Via Spacefilling Curves and Dynamic Programming," *DAC*, 1994, pp. 652-657.

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

[7] E. Amir and S. McIlraith, "Partition-Based Logical Reasoning for First-Order and Propositional Theories," *Artifi cial Intelligence*, 162 (1-2), 2005, pp. 49-88.

[8] C. J. Alpert, A. E. Caldwell, A. B. Kahng and I. L. Markov, "Hypergraph Partitioning with Fixed Vertices," *IEEE Trans. on CAD*, 19(2), 2000, pp. 267-272.

[9] C. J. Alpert, L. W. Hagen and A. B. Kahng, "A Hybrid Multilevel/Genetic Approach for Circuit Partitioning," *ASPDAC*, 1996, pp. 298-301.

[10] C. J. Alpert, J.-H. Huang and A. B. Kahng, "Multilevel Circuit Partitioning," *DAC*, 1997, pp. 530-533.

[11] C. Berman, "Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams," *IEEE Trans. on CAD*, 10(8), 1991, pp. 1059-1066.

[12] 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*, 1998.

[13] R. Bryant, “Graph-based Algorithms for Boolean Function Manipulation,” *IEEE Trans. on Comp.*, 35(8), 1986, 677-691.

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

[15] A. E. Caldwell, A. B. Kahng and I. L. Markov, “Optimal Partitioners and End-case Placers for Standard-cell Layout,” *IEEE Trans. on CAD*, vol. 19, no. 11, 2000, pp. 1304-1314.

[16] A. E. Caldwell, A. B. Kahng and I. L. Markov, “Improved Algorithms for Hypergraph Bipartitioning,” *ASPDAC*, 2000, pp. 661-666.

[17] A. E. Caldwell, A. B. Kahng, A. A. Kennings and I. L. Markov, “Hypergraph Partitioning for VLSICAD: Methodology for Heuristic Development, Experimentation and Reporting,” *DAC*, 1999, pp. 349-354.

[18] A. E. Caldwell, A. B. Kahng and I. L. Markov, “Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning,” *Lecture Notes in Computer Science*, vol. 1619, Springer, 1999, pp. 177-193.

[19] C. Choi and Y. Ye, “Application of Semidefinite Programming to Circuit Partitioning Problem,” *Approximation and Complexity in Numerical Optimization, Panos M Pardalos Edition*, 2000, Kluwer Academic Publishers, pp. 130-136.

[20] 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,” *ICCAD*, 1997, pp. 441-446.

[21] J. Cong and S. K. Lim, “Edge Separability Based Circuit Clustering with Application to Circuit Partitioning,” *ASPDAC*, 2000, pp. 429-434.

[22] J. Darnauer and W. Dai, "A Method for Generating Random Circuits and Its Ap-plications to Routability Measurement," *ACM/SIGDA International Sym-posium on FPGAs*, 1996, pp. 66-72.

[23] W. E. Donath and A. J. Hoffman, "Algorithms for Partitioning Graphs and Computer Logic Based on Eigenvectors of Connection Matrices," *IBM Technical Disclosure Bulletin*, 15(3), 1972, pp. 938-944.

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

[25] S. Dutt and H. Theny, "Partitioning Using Second-Order Information and Stochastic Gain Function," *ISPD*, 1998, pp. 112-117.

[26] P. Eles, Z. Peng, K. Kuchcinski and A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search," *Design Automation for Embedded Systems*, 1997, pp. 5 - 32.

[27] C. M. Fiduccia and R. M. Mattheyses, "A Linear-time Heuristic for Improving Network Partitions," *DAC*, 1982, p.175-181.

[28] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," San Francisco, CA: Freeman, 1979.

[29] M. Ghose, M. Zubair and C. E. Grosch, "Parallel Partitioning of Sparse Matrices," *Computer Systems Science & Engineering* 1, 1995, pp. 33-40.

[30] D. Ghosh, "Synthesis of Equivalence Class Circuit Mutants and Applications to Benchmarking," *summary of presentation at DAC Ph.D. Forum*, 1998.

[31] M. K. Goldberg and M. Burstein, "Heuristic Improvement Technique for Bisection of VLSI Networks," *IEEE Trans. on CAD*, 1983, pp. 122-125.

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

[33] L. Hagen and A. B. Kahng, “A New Approach to Effective Circuit Clustering,” *ICCAD*, 1992, pp. 422-427.

[34] E.-H. Han, G. Karypis, V. Kumar and B. Mobasher, “Hypergraph Based Clustering in High-Dimensional Data Sets: A Summary of Results,” *IEEE Bulletin of Technical Committee on Data Engineering*, 1998, pp. 1-8.

[35] S. Hauck and G. Borriello, “An Evaluation of Bipartitioning Techniques,” *IEEE Trans. on CAD*, 16(8), 1997, pp. 849-866.

[36] B. Hendrickson and R. Leland, “A Multilevel Algorithm for Partitioning Graphs,” draft, 1995.

[37] S. W. Hur and J. Lillis, “Mongrel: Hybrid Techniques for Standard Cell Placement,” *ICCAD*, 2000, pp. 165-170.

[38] S.-W. Hur and J. Lillis, “Relaxation and Clustering in a Local Search Framework: Application to Linear Placement,” *VLSI Design*, 2002, pp. 143-154.

[39] M. Hutton, J. P. Grossman, J. Rose and D. Corneil, “Characterization and Pa-rameterized Random Generation of Digital Circuits,” *DAC*, 1996, pp. 94-99.

[40] E. J. Im and K. A. Yelick, “Optimizing Sparse Matrix Vector Multiplication on SMP,” *PPSC*, 1999.

[41] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, “Multilevel Hypergraph Partitioning: Applications in VLSI Design,” *DAC*, 1997, pp. 526-529.

[42] G. Karypis and V. Kumar, “Analysis of Multilevel Graph Partitioning,” draft, 1995

[43] G. Karypis and V. Kumar, “Multilevel k-way Hypergraph Partitioning,” *DAC*, 1999, pp. 343-348.

[44] G. Karypis, E. H. Han and V. Kumar, “CHAMELEON: A Hierarchical Clustering Algorithm Using Dynamic Modeling,” *University of Minnesota Technical Report*, #99-007, 1999.

[45] G. Karypis and V. Kumar, “hMETIS: A Hypergraph Partitioning Package Version 1.5,” *user manual*, June 23, 1998, `http://glaros.dtc.umn.edu/gkhome/fetch/sw/hmetis/manual.pdf`.

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

[47] B. Krishnamurthy, "An Improved Min-cut Algorithm for Partitioning VLSI Net-works," *IEEE Trans. on Comp.*, vol. C-33, May 1984, pp. 438-446.

[48] B. Landman and R. Russo, "On a Pin Versus Block Relationship for Partitioning of Logic Graphs," *IEEE Trans. on Comp.*, C-20(12), 1971, pp. 1469-1479.

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

[50] LEF/DEF Language Reference, product version 5.6, Cadence Design Systems, September 2004.

[51] B. MacCartney, S. McIlraith, E. Amir and T. E. Uribe "Practical Partition-Based Theorem Proving for Large Knowledge Bases," *IJCAI*, 2003.

[52] hMETIS: `http://www-users.cs.umn.edu/~karypis/metis/hmetis/`.

[53] MLPart: `http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/Partitioning/MLPart/`

[54] M. Moskewicz, et al., "Chaff: Engineering an Efficient SAT Solver," *DAC*, 2001, pp. 530-535.

[55] G. Nam, F. A. Aloul, K. A. Sakallah and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *ISPD*, 2001, pp. 222-227.

[56] A. T. Ogielski and W. Aiello, "Sparse Matrix Computations on Parallel Processor Arrays," *The Society of Industrial and Applied Mathematics Journal on Scientifi c Computing*, 14(3), 1993, pp. 519-530.

[57] M. Prasad, P. Chong and K. Keutzer, "Why is ATPG easy?," *DAC*, 1999, pp. 22-28.

[58] L. Sanchis, "Multiple-way Network Partitioning with Different Cost Functions," *IEEE Trans. on Comp.*, Dec. 1993, vol.42, (no.12), pp. 1500-4.

[59] J. Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. on Comp.*, 48(5), 1999, pp. 506-521.

[60] Supplemental illustrations: `http://vlsicad.eecs.umich.edu/BK/PART/illustrations/`.

[61] UM Physical Design Tools: `http://vlsicad.eecs.umich.edu/BK/PDtools/`

[62] M. Velev and R. Bryant, "Superscalar Processor Verification Using Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," *Correct Hardware Design and Verifi cation Methods*, LNCS 1703, 1999, pp. 37-53.

[63] H. H. Yang and D. F. Wong, "Efficient network flow based min-cut balanced partitioning," *IEEE Trans. on CAD*, 1996, pp. 1533-1540.

| Benchmark | Tolerance | Solver | Cut | Runtime | Solver | Cut | Runtime |
|---|---|---|---|---|---|---|---|
| IBM01 | 2% | MLPart | 240.6 | 1.725 | hMETIS | 243 | 3.402 |
| IBM01 | 10% | MLPart | 231.9 | 1.535 | hMETIS | 238.3 | 3.304 |
| IBM02 | 2% | MLPart | 319.4 | 3.005 | hMETIS | 300.3 | 5.775 |
| IBM02 | 10% | MLPart | 294.4 | 2.715 | hMETIS | 296.1 | 5.159 |
| IBM03 | 2% | MLPart | 868.6 | 3.891 | hMETIS | 867.3 | 7.983 |
| IBM03 | 10% | MLPart | 804.8 | 3.756 | hMETIS | 766 | 7.952 |
| IBM04 | 2% | MLPart | 562.4 | 5.730 | hMETIS | 529.3 | 8.707 |
| IBM04 | 10% | MLPart | 494.0 | 4.806 | hMETIS | 449.5 | 9.469 |
| IBM05 | 2% | MLPart | 1752.2 | 6.465 | hMETIS | 1743.7 | 12.556 |
| IBM05 | 10% | MLPart | 1757.8 | 6.194 | hMETIS | 1719.8 | 12.270 |
| IBM06 | 2% | MLPart | 815.6 | 5.179 | hMETIS | 580.3 | 12.068 |
| IBM06 | 10% | MLPart | 451.1 | 5.668 | hMETIS | 392.7 | 11.567 |
| IBM07 | 2% | MLPart | 828.7 | 8.645 | hMETIS | 772.5 | 19.809 |
| IBM07 | 10% | MLPart | 792.0 | 8.576 | hMETIS | 767.3 | 19.782 |
| IBM08 | 2% | MLPart | 1210.4 | 9.078 | hMETIS | 1209 | 25.074 |
| IBM08 | 10% | MLPart | 1193.4 | 10.234 | hMETIS | 1158.6 | 25.031 |
| IBM09 | 2% | MLPart | 553.2 | 10.170 | hMETIS | 524.2 | 20.419 |
| IBM09 | 10% | MLPart | 549.4 | 9.831 | hMETIS | 524.4 | 19.593 |
| IBM10 | 2% | MLPart | 1325.1 | 13.062 | hMETIS | 1166.4 | 38.577 |
| IBM10 | 10% | MLPart | 1057.8 | 13.844 | hMETIS | 783.4 | 32.762 |
| IBM11 | 2% | MLPart | 864.1 | 11.711 | hMETIS | 823.1 | 30.754 |
| IBM11 | 10% | MLPart | 731.9 | 15.100 | hMETIS | 711.5 | 30.717 |
| IBM12 | 2% | MLPart | 2563.3 | 16.238 | hMETIS | 2108.4 | 45.575 |
| IBM12 | 10% | MLPart | 2310.5 | 16.885 | hMETIS | 1994.9 | 46.142 |
| IBM13 | 2% | MLPart | 973.5 | 15.510 | hMETIS | 913.3 | 41.623 |
| IBM13 | 10% | MLPart | 965.3 | 18.908 | hMETIS | 904.6 | 39.130 |
| IBM14 | 2% | MLPart | 1979.9 | 29.567 | hMETIS | 1837.1 | 119.690 |
| IBM14 | 10% | MLPart | 1780.5 | 31.452 | hMETIS | 1618.3 | 115.664 |
| IBM15 | 2% | MLPart | 2569.7 | 40.225 | hMETIS | 2222.5 | 117.491 |
| IBM15 | 10% | MLPart | 2190.9 | 42.251 | hMETIS | 1945.6 | 114.842 |
| IBM16 | 2% | MLPart | 1956.8 | 45.616 | hMETIS | 1720.4 | 155.785 |
| IBM16 | 10% | MLPart | 1988.7 | 47.512 | hMETIS | 1713.8 | 148.252 |
| IBM17 | 2% | MLPart | 2380.2 | 63.992 | hMETIS | 2445.5 | 217.874 |
| IBM17 | 10% | MLPart | 2291.4 | 62.204 | hMETIS | 2274.1 | 221.417 |
| IBM18 | 2% | MLPart | 1840.4 | 47.253 | hMETIS | 1687.5 | 225.259 |
| IBM18 | 10% | MLPart | 1583.5 | 51.323 | hMETIS | 1548.7 | 184.307 |

Figure 12: Performance of available software tools on common circuit hypergraph partitioning benchmarks. Results are the average of 10 runs with default configuration. Runs were performed by a 2.0GHz Pentium IV Xeon workstation with 2GB of RAM running Linux.

# Index

```
gain_update_special_cases ( move )
    source_part = partition that move.vertex() is in;
    dest_part = partition where move.vertex() is going;
    for_each(hyperedge e incident to move.vertex() )
        if( e.degree() == 2)
            v = the vertex on e that is not move.vertex();
            if( v is not locked and v is in source_part)
                gain_container.update( v, dest_part, 2*e.weight() );
            else if( v is not locked)
                gain_container.update( v, source_part, -2*e.weight() );
        else if( tallies[dest_part] == 0)
            for_each( vertex v on e )
                if( not locked( v )
                    gain_container.update( v, dest_part, e.weight());
        else if( tallies[source_part] == 1)
            for_each( vertex v on e )
                if( not locked( v )
                    gain_container.update( v, source_part, -e.weight());
        else
            for_each( vertex v on e )
                if( not locked( v ) )
                    if( v is in source_part )
                        if( tallies[source_part] == 2 )
                            gain_container.update( v, dest_part, e.weight() )
                            break;
                    else if( tallies[dest_part] == 1)
                        gain_container.update( v, source_part, -e.weight() )
                        break;
```

Figure 13: Pseudo-code for a faster gain update that takes advantage of special cases.