INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

AN EFFECTIVE ALGORITHM FOR MULTIWAY HYPERGRAPH PARTITIONING

Zhi Zi Zhao

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science at Concordia University
Montreal, Quebec, Canada

March, 2000

Abstract

An Effective Algorithm for Multiway Hypergraph Partitioning

Zhi Zi Zhao

The problem of hypergraph partitioning has been around for more than a quarter of a century. Its early applications were centered on VLSI circuit design. In recent years, the application of hypergraph partitioning has been extended into the areas including data classifications, efficient storage of very large database and data mining. In this thesis, we propose an effective multiway hypergraph partitioning algorithm. We introduce the concept of net gain and embed it in the selection of cell moves. Unlike traditional FM-based iterative improvement algorithms in which the selection of the next cell to move is only based on its cell gain, our selection is based on both cell gains and net gains. To escape from local optima and to search broader solution space, we propose a new perturbation mechanism. These two strategies significantly enhance the solution quality produced by our algorithm. Based on our experimental justification, we also smoothly decrease the number of iterations from pass to pass to reduce the computational effort so that our algorithm can partition large circuits with reasonable run time. Compared with the recent multiway partitioning algorithms proposed by Dasdan and Aykanat, ours significantly outperforms theirs in term of solution quality (cutsize) and run time: the average improvements in terms of average cutsize over their PLM3 (Partitioning by Locked Moves) and PFM3 (Partitioning by Free Moves) are 47.64% and 36.76% with only 37.17% and 9.66% of their run time respectively.

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to Dr. Lixin Tao, my supervisor, for his support and advice through my studies at Concordia University.

Thanks to Dr. Ali Dasdan at Synopsys Inc. for providing detailed information and implementation of his research work. To Dr. Chuck Alpert at IBM and the research members at VLSI CAD Laboratory (ABK GROUP) at University of California Los Angeles for providing information and benchmarks used in this study.

I wish to thank Dr. Rajagopalan Jayakumar and Dr. David Ford, members of my defense committee, for giving many valuable advice and comments.

My parents are always been there and supporting me. Few words are not enough to express my gratitude to them.

TABLE OF CONTENTS

List of Figures

Figure *Page*

List of Tables

Table *Page*

Chapter 1

Introduction

1.1 An Overview of Multiway Hypergraph Partitioning

Partitioning is a technique to divide a circuit or a system into a collection of smaller components. The main reason that partitioning has become a central and critical task today is the enormous increase of system complexity and the expected further advances of microelectronic system design and fabrication. Soaring system complexities result from a combination of the following reasons [20].

First, widely accepted powerful high-level synthesis tools allow the designers to automatically generate huge systems. By just changing a few lines of code in a functional specification, the size of the resulting structural description (hypergraph) of a system can increase dramatically. Synthesis and simulation tools often cannot cope with the complexity of the entire system under development, and designers want to concentrate on critical parts of a system to speed up the design cycle. Thus, the present state of design technology often requires a partitioning of the system.

Second, fabrication technology makes increasingly smaller feature sizes and augmented die dimensions possible, thus allowing a circuit to accommodate several million transistors. However, circuits are restricted in size and in the number of external

connections. Thus, fabrication technology requires the partitioning of a system into components.

Third, the various components of the system should be implemented in appropriate ways to achieve low-cost fabrication, optimal system performance, and easy adaptation to changing requirements. Thus, profit can be received by partitioning a system optimally.

Usually, circuits are represented as hypergraphs [3], the cells and the nets in circuits are represented by vertices and hyperedges respectively. The problem of multiway hypergraph partitioning has been around for at least a quarter of a century. It focuses on dividing a given hypergraph into a predetermined number (greater than two) of smaller blocks subject to the balance constraint while having the number of connections among these blocks minimized.

The hypergraph partitioning problem is an NP-hard problem [13]. (In [13], graph partitioning is proved to be an NP-complete problem, which is a special case of hypergraph partitioning). The only way to solve this problem is to use heuristic approaches for obtaining suboptimal solutions.

There are two primary approaches for generating multiway hypergraph partitioning solutions: recursive and direct. The recursive approach applies bipartitioning recursively until the desired number of blocks is obtained, whereas the direct approach partitions the circuit directly. The recursive approach is computationally simple and fast. However, it suffers from the following three major limitations. First, if the number of blocks k is not of power of 2, we cannot obtain the desired k-way partitioning by using the bipartitioning recursively. Second, it becomes harder and harder to reduce the cutsize since cut nets in

early stages cannot be removed when the bipartitioning performs finer cuts. For instance, a highly optimized cutset at one stage may cause the following stage to work on dense blocks. Those dense blocks cause negative effects on solution while applying further bipartitioning on them. Third, the recursive multiway partitioning can only minimize cost $(k-1)$ metric, not cost 1 metric that is often the objective to be minimized. Due to the weakness of the recursive approach mentioned above, the direct approach for multiway hypergraph partitioning plays a dominant role both in industry and research field.

1.2 Multiway Hypergraph Partitioning Applications

During the course of VLSI circuit design and synthesis, it is quite important to be able to divide the system specification into clusters so that the inter-cluster connections are minimized. This step has many applications including design packaging, estimation for design optimization, rapid prototyping, and circuit simulation and testing [1].

Design packaging. Semiconductor technology places restrictions on the total number of components that can be placed on a single chip. Logic circuits are partitioned into smaller subcircuits that can be fabricated on separate chips. Circuit partitioning is used to obtain such subcircuits, with a goal of minimizing the cutset, which determines the number of pins required on each chip.

Estimation for design optimization. Accurate estimation of layout area and wireability has always been a critical element of high-level synthesis and floorplanning. Now, such estimates are becoming critical to high-level searches over the system design space. Predictive models often combine analysis of the hypergraph partitioning structure with

analysis of the output characteristics of placement and routing algorithms, in order to yield estimates of wiring requirements and system performance.

Rapid prototyping. Many rapid prototyping systems use partitioning to map a complex circuit onto hundreds of interconnected Field-Programmable Gate Arrays (FPGAs). Such partitioning instances are challenging because the timing, area and I/O resource utilization must satisfy hard device-specific constraints. For example, if the number of signal nets leaving any one of the clusters is greater than the number of signal pins available in the FPGA, then this cluster cannot be implemented using a single FPGA. In this case, the circuit needs to be further partitioned, and thus implemented using multiple FPGAs.

Circuit simulation and testing. Partitioning has also been used to split a circuit into smaller subcircuits that can be simulated independently. The results are combined to study the performance of the overall circuit. This speeds up the simulation process by several times and is used in relaxation-based circuit simulators. This process is also used for simulating circuits on multiprocessors.

Early application of the multiway hypergraph partitioning was centered on VLSI circuit design and it is still a major research direction. In recent years, with the rapid development in the field of database and its applications, such as very large database, web database and large decision support system, the application of multiway hypergraph partitioning has been extended into the areas including data mining [25], efficient storage of very large database on disks [30].

Efficient storage of large databases requires information that is accessed together by individual queries to be stored on a small number of disk blocks. By clustering related information we can minimize the number of disk accesses. Hypergraph partitioning is an effective method for database clustering. In particular, the database is modeled by a hypergraph, in which various items (records) stored in the database are represented by vertices, and the records that are accessed together by single queries are connected via hyperedges. This hypergraph is then partitioned into blocks, so that the size of each block is smaller than the size of the disk sector, and the number of the hyperedges that connect records in different disk sectors is minimized. Other database related applications include the partitioning of roadmap database for routing applications [29].

Internet has become a vast source of information and service that continue to grow rapidly. Powerful search engines have been developed to help user in locating documents by categories, contents or subjects. Many intelligent software agents have used clustering techniques in retrieving, filtering and categorizing documents over the Internet. Multiway hypergraph partitioning based clustering algorithms perform better than traditional clustering methods in these new areas.

1.3 Thesis Outline

The remainder of this thesis is divided into five chapters.

Chapter 2 covers the most basic contents shared by all iterative improvement algorithms. We first present some definitions and notations that are necessary for the formal description of the multiway hypergraph partitioning problem. Based on these definitions

and notations, the multiway hypergraph partitioning problem is formulated. We then give the different objectives for the multiway hypergraph partitioning in current references. The algorithm for initial cell cost and cell gain calculation and the algorithm for cell cost and cell gain update that are shared by all iterative improvement algorithms are presented at the end of this chapter.

Chapter 3 briefly surveys the related work for the hypergraph partitioning. It is important for the formulation of our algorithm even though most of the related work focus on bipartitioning. We try to classify the related work into seven groups according to their characteristics. The graph bipartitioning algorithm proposed by Kernighan and Lin found the basis for most of the subsequent partitioning algorithms in VLSI. Schweikert and Kernighan extended algorithm KL to the hypergraph bipartitioning and its algorithmic speedup was developed by Fiduccia and Mattheyses (FM). Krishnamurthy further improved the algorithm with look-ahead ability. Sanchis extended FM with Krishnamurthy's look-ahead scheme to the multiway hypergraph partitioning. We group all these algorithms together and call them FM-based algorithm. All the algorithms in groups 2 through 6 are the improvements over FM-based algorithm by using different strategies. The network flow based approach in group 7 is used to solve the hypergraph bipartitioning.

Chapter 4 details our new algorithm. Based on the close exploration of the existing FM-based iterative improvement algorithms, we find that there still are rooms for improvement. We then present the motivation for our algorithm. The net gain concept that we have introduced and defined plays a key role in our algorithm. We also give the experimental justification for introducing the factor r to reduce the run time and a new

perturbation mechanism to improve the solution quality. Three parameters (the factor r, the number of perturbations and the amount of perturbation) are to be given in the algorithm. In this chapter, we illustrate the parameter settings by examples and give the recommendation for the values of these parameters.

Chapter 5 shows the experimental studies. All experiments are conducted on seven widely used ACM/SIGDA benchmark circuits. The number of cells ranges from 1752 to 103048, and the circuit density ranges from 0.00449 to 0.07983. We compare our algorithm with the algorithm developed by Sanchis's, and the ones developed by Dasdan and Aykanat(DA) on both solution quality (average cutsize, best cutsize, worst cutsize and standard deviation) and run time. The experimental results show that our algorithm significantly outperforms theirs in solution quality, and the run time of our algorithm is far smaller than those of DA.

We conclude our thesis in chapter 6 with a summary of our research.

Chapter 2

Multiway Hypergraph Partitioning

2.1 Problem Description and Notations

We use a hypergraph $H(C, N)$ to represent a circuit, where $C = \{c_i | i = 1, 2, ..., M_c\}$ (M_c is the number of cells) is the vertex (cell) set; $N = \{n_j | j = 1, 2, ..., M_n\}$ (M_n is the number of nets) is the hyperedge (net) set with each n_j being a subset of C with cardinality $|n_j| \geq 2$.

Definition 1 A *k-way partition* ($k \geq 2$) $\pi = \{B_i | i = 1, 2, ..., k\}$ of a circuit divides the cell set C into k disjoint blocks $B_1, B_2, \ldots, B_k$ such that:

$$B_i \cap B_j = \varnothing (i \neq j) \text{ and } \bigcup_{i=1}^{k} B_i = C.$$

Definition 2 A net n_j is said to be *incident* to a cell c_i if $c_i \in n_j$. If a net n_j is incident to a cell c_i, then we say that c_i is *on* n_j. The set of nets incident to c_i is denoted by $N(c_i) = \{n_j \in N | c_i \in n_j\}$.

Definition 3 A net n_j is said to be *incident* to a block B_i, if $n_j \cap B_i \neq \varnothing$.

Definition 4 Cells u and v are neighbors if and only if $N(u) \cap N(v) \neq \varnothing$.

Definition 5 The *degree* $d(c_i)$ of the cell c_i is defined as the number of nets incident to it, $d(c_i) = |N(c_i)|$. The *maximum cell degree* max.$d(c)$ is defined as $\max_{c_i}[d(c_i)]$. The *average cell degree* ave.$d(c)$ is defined as $\sum_{i=1}^{M_c} \frac{d(c_i)}{M_c}$. The *weight of the cell* c_i is a positive integer and is represented by $w(c_i)$.

Definition 6 The *degree* $d(n_j)$ of the net n_j is defined as the number of cells on it, $d(n_j)=|n_j|$. The *maximum net degree* max.$d(n)$ is defined as $\max_{n_j}[d(n_j)]$. The *average net degree* ave.$d(n)$ is defined as $\sum_{j=1}^{M_j} \frac{d(n_j)}{M_j}$. The *weight of the net* n_j is a positive integer and is represented by $w(n_j)$.

Definition 7 The *pin* is a connection point of a cell and a net.

The total number of pins M_p for a given circuit is $M_p = \sum_{i=1}^{M_c} d(c_i) = \sum_{j=1}^{M_n} d(n_j)$.

Definition 8 The *density D* of a circuit is defined as:

$$D = \frac{\sum_{j=1}^{M_n} (d(n_j) \cdot [d(n_j) - 1])}{M_c (M_c - 1)}.$$

Definition 9 If a net n_j is incident to a block B_i and $0 < |n_j \cap B_i| < |n_j|$, then the net n_j is a *cut*, i.e., a net n_j is a cut if it is incident to more than one block. A *cutset* is a set of all nets that are incident to more than one block. If a net

n_j is incident to a block B_i and $|n_j \cap B_i| = |n_j|$, then the net n_j is not in the cutset.

Definition 10 If a net n_j is incident to h ($h \geq 2$) blocks, there are three cost metrics depending on the cost assigned to the cut net n_j:

It is called "*cost 1*" *metric* if we assign a cost of 1 to the cut net n_j;

It is called "*cost h - 1*" *metric* if we assign a cost of $h - 1$ to the cut net n_j;

It is called "*cost h(h – 1)/2*" metric if we assign a cost of $h(h - 1)/2$ to the cut net n_j.

Like most hypergraph multiway partitioning algorithms, we will concern ourselves with the "*cost 1*" metric in this thesis.

Definition 11 A set $E(B_f)$ of *external nets* of a block B_f is defined as

$$E(B_f) = \{ n_j \in N \mid 0 < |n_j \cap B_f| < |n_j| \}.$$

Definition 12 A set $I(B_f)$ of *internal nets* of a block B_f is defined as

$$I(B_f) = \{ n_j \in N \mid |n_j \cap B_f| = |n_j| \}.$$

Definition 13 $n_j(l)$ is defined as the number of cells on net n_j that is in block B_l, i.e., $n_j(l) = |n_j \cap B_l|$.

Definition 14 Given two blocks B_s and B_t, $s \neq t$, for each cell $c_i \in B_s$, its *external cost* $C_i(s, t)$ is defined as $C_i(s, t) = \sum_{n_j \in E_i(s,t)} w(n_j)$,

where $E_i(s, t) = \{ n_j \in E(B_s) \mid c_i \in n_j \wedge n_j(t) = \mid n_j \mid -1 \}$ is the subset of $E(B_s)$ that would be deleted from the cutset if c_i is moved from B_s to B_t. Each cell has (k - 1) external costs, each of which corresponds to a move direction (target block).

Definition 15 The cost $C_i(s, s)$ of a cell c_i in block B_s is called its *internal cost* and is defined as

$$C_i(s, s) = \sum_{n_j \in I_i(s)} w(n_j),$$

where $I_i(s) = \{ n_j \in I(B_s) \mid c_i \in n_j \}$ is the subset of $I(B_s)$ that would be added to the cutset if c_i is moved from B_s to any other block. Each cell only has one internal cost.

Definition 16 The *cell gain* $g_i(s,t)$ of the move of a cell c_i from its source block B_s to its target block B_t is defined as $g_i(s,t) = C_i(s,t) - C_i(s,s)$.

Definition 17 A *cutsize* of a k-way partition $\pi = \{B_i \mid i = 1, 2, ..., k\}$ is equal to the sum of the weights of all cuts.

$$\text{cutsize}(\pi) = \sum_{j=1}^{M_n} w(n_j) - \sum_{q=1}^{k} \sum_{n_j \in I(B_q)} w(n_j).$$

Definition 18 The *weight* $w(B_i)$ *of a block* B_i is defined as $w(B_i) = \Sigma[w(c_i) \mid c_i \in B_i]$.

The *weight* $w(H)$ *of a circuit* H is defined as $w(H) = \sum_{i=1}^{M_c} w(c_i)$.

Definition 19 Given a k-way partition $\pi = \{B_i \mid i = 1, 2, ..., k\}$, the π is said to be *balanced* if for each B_i $(i = 1, 2, ..., k)$, the following balancing constraint is satisfied:

$$\left\lceil \frac{w(H)}{k} \cdot (1 - t) \right\rceil \leq w(B_i) \leq \left\lfloor \frac{w(H)}{k} \cdot (1 + t) \right\rfloor, \quad 0 < t < 1,$$

t is called the *balance tolerance*.

Definition 20 The *k-way partitioning* is to find a balanced partition $\pi = \{B_i \mid i = 1, 2, ..., k\}$ such that the cutsize(π) is minimized. The k-way partitioning is called *bipartitioning* for $k = 2$, and *multiway partitioning* for $k > 2$.

2.2 Multiway Partitioning Objectives

In most cases, the objective for multiway partitioning is to minimize the cutsize as mentioned in 2.1. Other multiway partitioning objectives have been proposed for different applications.

2.2.1 Ratio Cut [35]

It integrates the cutsize and block size balance within a single objective.

$$\text{Minimize } f(\pi_k) = \text{cutsize}(\pi_k) / \sum_{h=1}^{k-1} \sum_{l=h+1}^{k} w(B_h)\, w(B_l).$$

2.2.2 Scaled Cost [5]

It counts the number of times that a net is cut by a block, not the number of the cut nets.

$$\text{Minimize } f(\pi_k) = \frac{1}{M_c(k-1)} \sum_{h=1}^{k} \frac{|E(B_h)|}{w(B_h)}.$$

2.2.3 Absorption [32]

The absorption objective measures the sum of the fractions of nets "absorbed" by the blocks.

$$\text{Maximize } f(\pi_k) = \sum_{h=1}^{k} \Big(\sum_{n_j \in N \wedge n_j \cap B_h \neq \varnothing} \frac{|n_j \cap B_h| - 1}{|n_j| - 1} \Big).$$

A net n_j incident to block B_h adds absorption zero if n_j has only one cell in block B_h, and adds absorption one if n_j is completely contained in block B_h.

2.2.4 Density [19]

The density objective maximizes the sum of block densities, where the density of a block B_h is the number of the nets completely contained in B_h divided by the weight of B_h.

$$\text{Maximize } f(\pi_k) = \sum_{h=1}^{k} \frac{|\{n_j \in N \wedge |n_j \cap B_h| = |n_j|\}|}{w(B_h)}.$$

2.2.5 FPGA (Field-Programmable Gate Array) Partitioning [18]

The motivation of this objective is partitioning onto multiple devices, e.g., onto FPGAs for rapid prototyping or system emulation. A given device has strict upper bounds on both the number of cells that can be placed onto the device, as well as the number of nets

that can connect to other devices. For a given number of devices k, FPGA partitioning seeks to assign cells to devices such that $|B_h| < U_h$ ($|B_h|$ is the number of cells placed onto device k and U_k is the upper bound) with the following objective:

Minimize $f(\pi_k) = \max_{1 \le h \le k} |E(B_h)|$.

2.3 Calculate Initial Cost

We need to have a procedure to compute the initial cell costs and cell gains at the startup of our algorithm. The initial cost calculation procedure is exactly the same as the one used in [9]. Given an initial multiway partitioning $\pi_k = \{B_i \mid i = 1, 2, ..., k\}$, we have to calculate the internal cost and the external costs for each cell of the circuit for further obtaining the cell gains of each cell to move from its source block to $(k-1)$ target blocks respectively. We obtain the initial cutsize according to the definition 17 in section 2.1. The initial cost calculation algorithm is shown in Figure 1.

The **for** loop of lines 1-10 performs the same operations for each cell. The **for** loop of lines 2-3 initializes each cost of a cell c_i to zero. The cost of c_i depends only on the number of critical nets it has (a net is a critical net if it will be added to the cutset or removed from the cutset immediately after a move of a related cell), and these costs are computed as in Definition 14 and Definition 15. Thus, for each net n_j that is incident to c_i, we first check whether n_j is a critical net or not. The **for** loop of lines 2-3 is executed $\Theta(k)$ times since each cell has k costs. Lines 7-8 and line 10 take constant time. Line 6 requires $O(k)$ time because we should check $(k-1)$ blocks to find the B_t in the worst

case. The **for** loop of line 4-10 iterates $d(c_i)$ times. Since $\sum_{i=1}^{M_c} d(c_i) = M_p$, the initial cost calculation algorithm takes $O(M_p k)$ time.

```
1  for each cell c_i ∈ C, where c_i ∈ B_s do
2      for each block B_t do
3          C_i(s, t) = 0          /* Initialize the cost*/
4      for each net n_j incident to cell c_i do
5          if n_j(s) = 1 then
6              try to find a block B_t such that n_j(t) = |n_j| - 1
7              if such B_t has been found then
8                  C_i(s, t) = C_i(s, t) + w(n_j)
9          else if n_j (s) = |n_j| then
10                 C_i(s, s) = C_i(s, s) + w(n_j)
```

Figure 1: The initial cell cost calculation procedure

2.4 Update Cell Cost and Cell Gain

Once a cell is moved from its source block to its target block, its costs and the costs of its neighbors must be updated accordingly to indicate the effect of the move correctly. Also we need to update all the corresponding cell gains. The procedure for updating the cell costs and the cell gains is the same as the one described in [9].

We also adapted the bucket data structure, which is proposed in [27] and also used in [9], in our algorithm. The details on this structure are presented in section 4.7. Here we assume the bucket array size is the same as the one in FMS and PLM, that is $(2 \cdot G_{max} + 1)$, where G_{max} is the product of the maximum cell degree and the maximum net weight. We will show our modification on the bucket array size in section 4.7. After moving a cell c_i from its source block B_s to its target block B_t, the procedure for updating all cell costs and cell gains of c_i and those of its neighbors is shown in Figure 2.

/* $C_i(s,q), C_i(s,t)$ are the costs before the move */

1 recompute $g_i(t,q)$ for each $q \neq t$

2 **for** each net n_j incident to cell c_i **do**

3 **if** $n_j(s) = |n_j|$ **then**

4 **for** each cell $c_r \in n_j$ where $r \neq i$ **do**

5 $C_r(s, s) = C_r(s, s) - w(n_j)$ /* update $C_r(s,s)$ */

6 $g_r(s,q) = C_r(s,q) - C_r(s,s)$ for each $q \neq s$ /* update $g_r(s,q)$ */

7 **else if** $n_j(s) = |n_j| - 1$ **then**

8 find cell $c_r \in n_j$ such that $c_r \in B_q$ and $q \neq s$

9 $C_r(q, s) = C_r(q, s) - w(n_j)$ /* update $C_r(q, s)$ */

10 $g_r(q,s) = C_r(q,s) - C_r(s,s)$ /* update $g_r(q,s)$ */

11 $n_j(s) = n_j(s) - 1$

12 $n_j(t) = n_j(t) - 1$

13 **if** $n_j(t) = |n_j|$ **then**

14 **for** each cell $c_r \in n_j$ where $r \neq i$ **do**

15 $C_r(t,t) = C_r(t,t) + w(n_j)$ /* update $C_r(t,t)$ */

16 $g_r(t,q) = C_r(t,q) - C_r(t,t)$ /* update $g_r(t,q)$ */

17 **else if** $n_j(t) = |n_j| - 1$ **then**

18 find cell $c_r \in n_j$ such that $c_r \in B_q$ and $q \neq t$

19 $C_r(q,t) = C_r(q,t) + w(n_j)$ /* update $C_r(q,t)$ */

20 $g_r(q,t) = C_r(q,t) - C_r(t,t)$ /* update $g_r(q,t)$ */

Figure 2: The cell cost and cell gain update procedure

This procedure is invoked right after a cell is moved. We present the time complexity for this procedure under the assumption that the partitioning algorithm uses the cell locking mechanism, like FMS, PLM and our new algorithm.

We look at this procedure in the scope of a pass. For a net n_j, lines 4-6 are executed at most once since all the unlocked cells on n_j can only be in B_s, and, after a move involving a cell that is on n_j from B_s, we can no longer have all cells on the net n_j in B_s again in the same pass. For a net n_j, lines 8-10 are executed at most three times. If $n_j(B_s) = d(n_j) - 1$ and $n_j(B_l) = 1$ for some block B_l before a cell move, only B_s and B_l can again be a source block for a cell move involving a cell on the net n_j. Now, since B_l can have only one unlocked cell that is on n_j, we can have $n_j(B_l) = d(n_j) - 1$ at most once. Also, if $n_j(B_s) = d(n_j) - 1$ before a cell move of a cell on n_j from B_s, we can have $n_j(B_s) = d(n_j) - 1$ at most once again only if the only cell on n_j in B_l move to B_s. Lines 14-16 are executed at most once since all the cells of n_j are already in B_t, and B_t cannot be a target block again to

have $n_j(B_t) = d(n_j)$. For a net n_j, lines 18-20 are executed at most three times for similar reason stated above for lines 8-10. The **for** loop of lines 4-6 takes $O(kG_{max} \mid n_j \mid)$ time since we update $(k-1)$ gains for each cell on n_j and these gain updates can involve $(k-1)$ deletions from bucket lists, and each deletion takes $O(G_{max})$ time. Line 8 requires $O(k)$ time, the line 9 takes constant time, and line 10 takes $O(G_{max})$ time since it can involve only one deletion from a bucket list. The **for** loop of lines 14-16 takes $O(kG_{max} \mid n_j \mid)$ time because we update $(k-1)$ gains for each cell on n_j and these gain updates can involve $(k-1)$ deletions from bucket lists. Finally, line 18 requires $O(k)$ time, line 19 takes constant time and the line 20 takes $O(G_{max})$ time since it can involve only one deletion from a bucket list. Moreover, line 1 takes $O(kG_{max})$ time because it can involve $(k-1)$ deletions from bucket lists. In total, the cost update algorithm runs in time $O(M_p kG_{max})$ for at most M_c cell moves in a pass.

Chapter 3

Related Work

3.1 FM-based Iterative Improvement Algorithms

In 1970, Kernighan and Lin (KL) [22] proposed a well-known heuristic two-way graph partitioning algorithm that has became the basis for most of the subsequent partitioning algorithms in the research field. The algorithm KL starts with a balanced two-way partition, and it performs a number of passes until a local minimum cutsize is found. A pass consists of a number of pairwise cell swappings between the two blocks. A gain is defined as the change in cutsize due to the pairwise cell swapping. A positive gain means a decrease in cutsize, and negative gain an increase. To avoid thrashing during each pass, after a pair of cells is swapped, the pair of cells becomes locked until the start of the next pass. Therefore, each cell moves exactly once in each pass. KL iteratively swaps a pair of unlocked cells with the highest cell gain until all the cells become locked, and it returns the minimum cutsize observed during the pass. The next pass is executed using the solution from the current pass as its initial solution. The algorithm terminates when a pass fails to find a solution with lower cutsize than that of its initial solution. From the viewpoint of a pass, KL can climb out of local minima since it always swaps a pair of cells with the highest gain even if this gain is negative. However, KL only accepts a pass with positive gains. If we consider the whole algorithm, KL is still a greedy algorithm.

Schweikert and Kernighan [28] extended the algorithm KL to handle the two-way hypergraph partitioning problem.

Fiduccia and Mattheyses (FM) [12] presented a modified version of algorithm KL to enhance the solution quality. They introduced a new data structure (bucket array lists of cell gains) to achieve the linear run time per pass. They also proposed a single cell move instead of swapping a pair of cells in one move. This allows for more flexibility in selecting the candidate cells for movements.

Krishnamurthy (KR) [24] suggested that the lack of an "intelligent" tie-breaking scheme among many possible cell moves with the same gain could cause the FM to make "bad" choices. He adopted FM with a look-ahead scheme that looks ahead up to the r^{th} level of cell gains to choose a cell move. Krishnamurthy introduced a gain vector as a tie-breaking mechanism, which is a sequence of potential gain values corresponding to a sequence of possible moves in future. Note that the 1^{st}-level gain is identical to the cell gain in the algorithm FM. The r^{th} entry in the gain vector looks r moves ahead, and ties are broken lexicographically by the 1^{st}-level gains, the 2^{nd} -level gains, etc.

Sanchis (SA) [27] extended the FM with Krishnamurthy's look-ahead scheme to multiway partitioning. The algorithm SA (hereafter we call it FMS) is the first hypergraph multiway partitioning algorithm since all algorithms before it are for bipartitioning. FMS is extensively used as a benchmark in performance comparison for different algorithms. Hereafter, we call KL, FM, KR and FMS FM-based algorithms. They dominate both VLSI CAD research community and industry practice for several reasons. They are generally intuitive (the obvious way to improve a given solution is to

repeatedly make it better via cell moves), flexible in adapting to different optimization objectives, easy to implement, and relatively fast.

Afterwards many modifications directed to FM-based algorithms have been made to improve the solution quality, and the results from these modifications are encouraging.

3.2 Integration of the Cutsize and the Balance into Objective Function

Most of the partitioning algorithms deal with the cutsize and the balance of the partition separately, i.e., the objective function only includes the cutsize and the balance is considered as the constraint on a cell move. If a cell move violates the balance constraint, it is prohibitive to move even though it results in a significant reduction in cutsize. Park and Park [26] pointed out that the cell move operation is largely influenced by the balance constraint, and they proposed a cost function that comprises both cutsize and balance degree (that is the sum of all size differences for every pair of different blocks) with a positive weighting factor. They proved that a multiway partitioning with minimum cost corresponds to a balanced minimum cutsize as defined in FMS if the weighting factor is larger than the number of cells in a circuit. The FMS is then used to solve the multiway partitioning under this objective function. The experiment was conducted on circuits with 100, 200, 400 and 1000 cells. The experimental results showed their algorithm outperforms FMS in most of the cases.

3.3 Relaxed Locking Mechanism

A possible weakness for FM-based algorithms lies in the locking mechanism, e.g., a cell c is moved from block 1 to block 2 early in a pass, and one or more of its neighbor cells are later moved from block 2 to block 1, then c will be in the "wrong" block. To rectify this behavior, Hoffman [17] proposed a dynamic locking mechanism for bipartitioning that behaves like FM, except that, when c is moved out of a block j, each neighbor of the cell c in the block j becomes unlocked. This allows the neighbors of the cell c to also migrate out of the block j. The locking mechanism permits a maximum number of ten moves for each cell in a pass.

Dasdan and Aykanat [9] (DA) developed two multiway partitioning algorithms using a relaxed locking mechanism. The first one, PLM (Partitioning by Locked Move), uses the locking mechanism in a relaxed manner. It allows multiple moves for each cell in a pass by introducing the phase concept so that each pass may contain more than one phase and each cell has a chance to be moved only once in each phase. The second algorithm, PFM (Partitioning by Free Move), does not use the locking mechanism at all. A cell can be moved as many times per pass as possible based on its mobility value. The performance of the proposed algorithms was experimentally evaluated in comparison with Simulated Annealing algorithm (SAA) and FMS on some benchmark circuits. Their results outperform FMS significantly on multiway partitioning and their performance is comparable to that of SAA with much less run time.

3.4 Tie-breaking Strategies

Tie-breaking strategies play an important role in circuit partitioning. Even when gain vectors [24] are used, ties may still occur among cell gains. Hagen et al. [14] observed that Sanchis [27], and most likely Krishnamurthy [24], used random selection schemes. They [14] found that the LIFO (Last-In-First-Out) bucket organization is distinctly superior to the FIFO (First-In-First-Out) and the random bucket organization. The LIFO buckets result in a 36% improvement over the random buckets and a 43% improvement over the FIFO buckets for bipartitioning. They ascribe the success of the LIFO to its enforcement of "locality" in the choice of cell to move, i.e., cells that are naturally clustered together will tend to move sequentially.

Dutt and Deng [10] proposed a different kind of tie-breaking approach for bipartitioning based on probabilistic techniques. Instead of using a gain value that reflects only the immediate change in cutsize from moving a cell, their algorithm PROP uses a more global gain computation. Each cell has an associated probability for the event that the cell will actually be moved to the other block. PROP begins by assigning each cell with an initial probability of 0.95, and then the gains are recomputed based on a function of the current solution and the cell probabilities. As cells are moved, probabilities and gains are updated for the neighboring cells. Experiments showed that PROP significantly outperforms FM.

N. Woo and J. Kim [37] proposed the notion of the benefit that is a scalar value associated with each free cell. The selection of a cell to move is based on its benefit. If moving a cell results in decrease/increase of the cutsize, the move has a positive/negative

primary benefit. Sometimes moving a cell does not change the cutsize, but the move may contribute to the primary benefit of other free cells. In such case, the move has a secondary benefit. They used two parameters as weights of the primary benefit and the secondary benefit respectively. Therefore, the benefit captures look-ahead information and reduces the opportunity of many cells having the same gain. Their algorithm, called MP2, had been applied to partitioning technology-mapped circuits into multiple FPGA chips.

3.5 Cluster-oriented Algorithms

In recently years, many cluster-oriented iterative improvement algorithms have been proposed, and the experimental results are encouraging. Dutt and Deng [11] pointed out that the FM-based iterative improvement algorithms could only remove small clusters from the cutset while it likely locks big clusters in the cutset. They proposed two new iterative improvement bipartitioning algorithms called CLIP (Cluster-oriented Iterative-improvement Partitioner) and CDIP (Cluster-Detecting Iterative improvement Partitioner). The algorithms select cells to move in an attempt to move clusters that straddle the two blocks of a partition. They divided the cell gain into the initial gain, which is calculated before a cell movement, and the update gain, which is generated after a cell movement. By focusing on the update gain when choosing cells to move, they reported very successful results for bipartitioning experiments.

Cong et al. [6] defined loose net and stable net for the hypergraph bipartitioning problem. A net is defined to be loose if the cells on it are only locked at one block. They focused

on the status of nets instead of cells as often emphasized in the traditional algorithms. A cluster removal strategy is accomplished by gradually increasing the gains of the neighbors of the currently moved cell via loose nets until all of them are moved to one block. A net is defined to be stable if it has remained in the cutset throughout the entire run. They observed that more than 80% of the nets in the final cutset are stable and these nets trap FM-based algorithms into local minima. Based on this observation, they proposed a hill-climbing method called Stable Net Transition that enables FM-based algorithms to discover multiple sub-optimal solutions within a practical run time. Their algorithm was developed for bipartitioning and the experimental results are also encouraging.

Cong and Lim [7] proposed a multiway partitioning algorithm with pairwise movement. It starts with an initial multiway partition by adapting existing loose net removal scheme [6], and then it applies the bipartitioning heuristic (FM) to pairs of blocks concurrently to improve the quality of the overall multiway partitioning solution. It outperforms its counterpart recursive FM by up to 17.3% on MCNC (Microelectronics Center of North Carlolina) and large-scale ISPD98 benchmark circuits.

The FM-based algorithms are very easy to get trapped in local minima as the size of the input circuits grows. Kernighan and Lin [22] empirically found that the probability of terminating with the optimal solution in a single trial to be approximately $2^{-n/30}$ for the test circuits, where n is the number of nodes in an input circuit. To improve the performance of the FM-based algorithms, one solution is to coarsen (clustering) the input circuit before partitioning. This is accomplished by dividing the input circuit with n nodes

represented by a hypergraph $H(V, E)$ into c clusters, $P^c = \{C_1, ..., C_c\}, c >> k$ (k is the number of blocks). Then we build a contracted circuit represented by a coarsened hypergraph $H'(V', E')$ with the c clusters as nodes, i.e. $V' = \{C_1, ..., C_c\}$ and $E' = \bigcup_{e \in E} \{C_i \mid |e| > 1 \wedge e \cap C_i \neq \varnothing\}$. This coarsened hypergraph reduces the problem size from n to c. FM is then run once on $H'(V', E')$ to yield the bipartitioning P_1, and P_1 is projected to a new bipartitioning P_0 of $H(V, E)$. Finally, FM is run a second time on $H(V, E)$ using P_0 as its initial solution. This second run of FM is called a refinement step that aims at the improvement of a good initial solution. This is defined as the two-phase approach.

Shin and Kim [31] developed a hierarchical partitioning algorithm. At cluster-level, clusters (they define a cluster as a set of highly connected cells) are formed, and then the best solution of the cluster-level partitioning is used as the initial solution of the cell-level partitioning. At both levels, the gradual constraint-enforcing technique is used. It starts with loosened size constraints that are gradually tightened as iterations progress. It allows clusters or cells to move with more freedom among blocks during earlier iterations and thus may find a better solution.

The primal-dual multiway partitioning algorithm proposed by Yeh et al. [34] is also a two-phase approach. It groups highly connected cells into clusters and then condense these clusters into super nodes prior to the execution of the FM-based algorithm. A primal-dual iteration on cluster system and original system is performed sequentially. The primal-dual iteration alternates "primal" passes of cell-based moves with "dual" passes of net-based moves. The cell-based move is the same as that in FM. The net-based move may move all the cells of a net at once. The complexity of the gain computation causes a

dual pass to require around 9-10 times more run time than that for a primal pass. Hauck and Borriello [15] concluded that dual passes "are not worthwhile".

The two-phase approach can be extended to the multilevel paradigm by using as many phases as desired. The multilevel paradigm was independently studied by Bui and Jones [4] in the context of computing fill reducing matrix reordering, by Hendrickson and Leland [16] in the context of finite element grid partitioning, and by Hauck and Borriello [15] and Cong and Smith [8] for hypergraph bipartitioning. In the multilevel paradigm, we build a hierarchy of successively coarser hypergraghs $H_i(V_i, E_i)$ ($i = 0, 1, \ldots, m$) by applying the approach mentioned above on the original hypergraph $H_0(V_0, E_0)$, and generating $H_1(V_1, E_1)$, ..., $H_m(V_m, E_m)$. Then algorithm FM is then applied to $H_m(V_m, E_m)$, and the obtained solution is projected back to the previous level $H_{m-1}(V_{m-1}, E_{m-1})$. This process is continued until partitioning on $H_0(V_0, E_0)$ is performed. The multilevel paradigm offers two advantages over the two-phase approach. First, the multilevel partitioning enables coarsening to proceed more slowly, and thus gives the iterative engine more opportunities for refinement. Second, the multilevel partitioning can be extremely efficient if a fast clustering and refinement strategy is used. Alpert et al. [2] proposed a partitioning algorithm based on the multilevel paradigm. Two key ingredients have been added to improve the performance significantly: (1) algorithm CLIP (Cluster-oriented Iterative-improvement Partitioner) [11] is used within the FM implementation; (2) a matching based clustering is used that halts prematurely so that more than n/2 clusters are generated. This causes the multilevel coarsening to proceed more slowly.

Karypis et al. [21] presented a multilevel hypergraph bipartitioning algorithm *hMETIS* that operates directly on hypergraphs. They used powerful coarsening schemes and

developed multiphase refinement schemes. Wichlud and Aas [36] proposed a multilevel bipartitioning algorithm by taking edge-frequency information into account during the coarsening uncoarsening and the tie-breaking. Their algorithm has 34.6% improvement in average cutsize over that obtained by the algorithm in [2] for bipartitioning on 13 benchmark circuits.

3.6 Multiway Partitioning Algorithms by Simulated Annealing

The Simulated Annealing algorithm (SAA) has its origin in statistical mechanics. The interest began with the work of Kirkpatrick, Gelatt and Vecchi [23]. The SAA is based on the analogy between the annealing process of solids and the problem of solving combinatorial optimization problems. Given a neighborhood structure and a current solution, SAA picks a random neighbor of the current solution and moves to this new solution if it represents a downhill move. Even if the new solution represents an uphill move, SAA will move to it with probability $e^{-\Delta/T}$ and otherwise it will retain the current solution; here Δ is the cost of the new solution minus the cost of the current solution, and T is the current value of a temperature parameter that guides the optimization. To control the rate of convergence and the strategy for exploring the solution space, the user typically establishes a temperature schedule by which T varies, e.g., as a function of the number of moves made. One can show that SAA will converge to a globally optimum solution given an infinite number of moves and a temperature schedule that cools to zero sufficiently slowly.

Dasdan and Aykanat [9] adapted simulated annealing for multiway circuit partitioning, and the experimental results show that the solution quality of their PFM3 is comparable to that of SAA, but the run time of SAA is far larger than that of PFM3.

3.7 Network Flow Based Partitioning Algorithms

The max-flow min-cut technique was overlooked as a viable approach for circuit bipartitioning due to the following reasons: First, the two blocks obtained by it are not necessarily balanced; second, although a balanced cut can be achieved by repeatedly applying min-cut to the larger block; this method can possibly incur n max-flow computations, where n is the size of flow network; third, the traditional network flow technique works on graphs, not hypergraphs.

Recently, Yang and Wong [33] proposed a network flow based min-cut balanced hypergraph bipartitioning algorithm. They proposed a method for exactly modeling a hypergraph by a flow network, and then developed a balanced bipartitioning heuristic based on a repeated max-flow min-cut technique. The experimental results show that their algorithm outperforms SAA and PFM3 on 5 benchmark circuits (cell size from 478 to 3466). The average improvements of the cutsize over SAA and PFM3 are 24.5% and 19% respectively.

Chapter 4

The Proposed Hypergraph Multiway Partitioning Algorithm

4.1 Motivation

All FM-based iterative improvement algorithms are started with a random initial solution. Each cell has $(k-1)$ cell gains for $(k-1)$ directions. A cell with maximum cell gain for a particular moving direction is chosen from all possible movements that will not violate the balance constraint. The selected cell is then moved to the target block and is locked there for the current pass. The cell gains of all the affected neighbors of the moved cell are updated accordingly. The next cell is chosen in the same way from all the remaining free (unlocked) cells and is moved to its target block. The cell move process is repeated until all the cells are locked or there are no legal moves available due to the balance constraint.

Assume that there are q $(q \le M_c)$ cell moves all together. Then all the partial gain sums $S_p = \sum_{x=1}^{p} g_x(a_x, b_x),\ p = 1, 2, ..., q$, where $g_x(a_x, b_x)$ is the gain of moving the x^{th} cell from block a_x to block b_x given that the first $(x-1)$ cell moves have already been made, are calculated and the maximum partial gain sum S_β is chosen. This corresponds to a point

of minimum cutsize in the entire cell moving process. Only the cells moved up to the β^{th} cell move are made permanent so that the actually moved cells are the sequence of moving the first cell from block a_1 to block b_1, the second cell from a_2 to b_2, ... , the β^{th} cell from a_β to b_β, where $a_1, a_2, ..., a_\beta, b_1, b_2, ..., b_\beta \in \{1, 2, ..., k\}$, k is the number of blocks. The whole process is called a pass. A number of passes, which is called a run, is performed until the maximum partial gain sum is no longer positive. The number of passes cannot be known in advance. Then we say that the local optimum with respect to the initial solution is obtained.

For several reasons, the solution quality produced by a FM-based iterative improvement algorithm is often poor, especially for larger circuits. First, it has been criticized for its well-known shortsightedness on the way of choosing a cell to move. It is only based on local information (cell gain) of the immediate decrease in cutsize. For example, it may be better to move a cell with smaller gain, as it may allow many good moves later. Thus it tends to be trapped in local optimum, which strongly depends on the initial solution. Second, many cells have the same cell gain, especially for larger circuits. The FM-based iterative improvement approach has no insightful scheme to choose which of these cells to move. Only critical net information is used for the cell gain calculation. Speaking critical net we mean that the net will be added to the cutset, or it will be removed from the cutset, immediately after a move of a related cell.

After taking a close look at a particular cut net, we find that if one cell on the net is locked in a block, the only way to remove the net from the cutset is to pull all other cells on the net to the block where the locked cell has already been moved in. If two or more

cells of a net are locked in two or more blocks, then there is no way to remove that net from the cutset in the current pass. Due to the missing of dynamic net information, bigger clusters are very likely to be locked in the cutset. Third, a cell gain defined by FM ranges from $(-\max_{c_i}[d(c_i)])$ to $(\max_{c_i}[d(c_i)])$. A bucket structure consists of an array of $\{2 \cdot \max_{c_i}[d(c_i)] + 1\}$ entries to which cells with the same cell gain are linked. Since the entire $M_c(k-1)$ cell gains are distributed in this short range, many cells may have the same cell gain; a better tie-breaking scheme like the look-ahead capability proposed by Krishnamurthy [24] is needed. Forth, even though the uphill moves with negative gains, which are always the best ones among all legal moves, are accepted in each pass, the possibility for exploring broader solution space and finding better solution is limited as the result of lacking a better strategy to escape from a local optimum.

To enhance the solution quality for the iterative improvement algorithms, we must try to overcome the weak points mentioned above. This is just the motive for developing our new algorithm.

4.2 Net Gain Concept

Before getting into the details of our algorithm, we first give some definitions that are related to the description of the algorithm.

A net is called a *free net* if all cells on it are unlocked. If all the locked cells on a net are distributed in a single block, the net is called a *loose net*. In this case, the block in which the locked cells are located is called the *locked block* for that net, and all the other blocks

in which the free cells are located are called the *free blocks* for that net. If the locked cells are distributed in two or more blocks, the net is called the *locked net*.

In an attempt to remove big clusters from the cutset, more dynamic net information is needed. Here we introduce the concept of *net gain* for each loose net. If a cell is moved to a block and locked there, we use the values of its net gain to encourage its neighboring cells to be moved subsequently to the locked block where the moved cell is just locked in. The net that straddles the cut line is thus removed. Unlike most of the other FM-based iterative improvement algorithms in which the selection of the next cell to move is only based on its cell gain, our algorithm selects a cell based on both its cell gain and the sum of all net gains for all loose nets incident to the cell.

For each net, there is a net gain array (called net_gain) of size k associated with it. Initially, all nets are free and all the k elements are set to zero for each array. After a cell c is moved to a block B_L, it is then locked during the current pass. For each loose net, there is exactly one locked block, and the number of free blocks is between 1 and $(k-1)$. The elements of a net_gain array are only defined for a loose net since it makes no sense to have them for either locked nets or free nets.

As mentioned earlier, we use the values in the net_gain array to encourage the free cells in free blocks of a loose net to move to the locked block of the net. The *net gain* of a loose net n_α for one of its free blocks B_F is defined as follows:

$$\text{net\_gain}[n_\alpha][B_F] = \left\lceil \frac{\max\limits_{n_j}[d(n_j)]}{d(n_\alpha)} \cdot \frac{\sum\limits_{c \in S_L}[1 + d(c) - cut(c)]}{\sum\limits_{c \in S_F}[1 + d(c) - cut(c)]} \right\rceil ,$$

where $\max\limits_{n_j}[d(n_j)]$ is the maximum net degree in the given circuit;

$d(n_\alpha)$ is the degree of the net n_α;

S_L is the set of locked cells on net n_α in locked block B_L;

S_F is the set of free cells on net n_α in free block B_F;

$d(c)$ is the degree of the cell c (number of nets incident to cell c);

cut (c) is the number of nets incident to cell c that are in the cutset.

Basically, each element of a net gain array is a product of two fractional expressions:

$$\frac{\max\limits_{n_j}[d(n_j)]}{d(n_\alpha)} \quad \text{[a]}, \quad \text{and} \quad \frac{\sum\limits_{c \in S_L}[1 + d(c) - cut(c)]}{\sum\limits_{c \in S_F}[1 + d(c) - cut(c)]} \quad \text{[b]}.$$

The purpose of expression [a] is to give smaller cut nets higher chance to move. The smaller the net degree of n_α is, the bigger the value of expression [a] is. Expression [b] indicates that the more locked cells the net n_α has in its locked block B_L, or the fewer cells the net n_α has in its free block B_F, the higher value net_gain[n_α][B_F] has. It also indicates that for locked cells with more internal nets (which are not in the cutset) and for free cells with fewer internal nets, the net_gain[n_α][B_F] gets higher value. This approach

is more appropriate than that used by [6] for the bipartitioning problem since [6] does not distinguish the internal nets from the external nets, and it does not have a mechanism to remove this gain once the loose net becomes locked.

All the free cells of the loose net n_α in free block B_F have the same net_gain$[n_\alpha][B_F]$. The net_gain$[n_\alpha][B_F]$ encourages all the free cells currently in the free block B_F to move to the locked block B_L for ultimately removing the loose net n_α from the cutset. The selection of cell movements in our algorithm is based on the move gains of each cell. For each cell, there is a move gain array of size k (called move_gain) associate with it. Also, for each cell, we use a net_cell_gain array that has the same structure as the move_gain array. The net_cell_gain$[c][B_L]$ has the value

$$\sum_{n_\alpha \in S_c} \text{net\_gain}[n_\alpha][B_F],$$

where S_c is the set of all loose nets that are incident to the free cell c and share the same locked block. For each free cell c of the loose net n_α in the free block B_F, the *move gain* of cell c to the locked block B_L is defined as:

$$\text{move\_gain}[c][B_L] = \text{cell\_gain}[c][B_L] + \text{net\_cell\_gain}[c][B_L],$$

where cell_gain$[c][B_L]$ is calculated as the conventional FM-based algorithms (defined in Definition 16).

Now, we illustrate how to calculate net_gain and net_cell_gain for a given circuit. In Figure 3, we only show parts of cells and nets that are related to our explanation and omit the rest.

This is a 3-way hypergraph partitioning instance, in which net 1 is a free net, and net 2 is a loose net because cell 5 and cell 7 are locked cells in a single block (block 2). Net 3 is a locked block with locked cells (cell 3 and cell 7) in block 1 and block 3 respectively. We are only interested in the loose net 2. Free blocks 1 and 3 and locked block 2 are associated with the loose net 2. Assume that net 2 is the biggest net with degree 5 in the circuit.

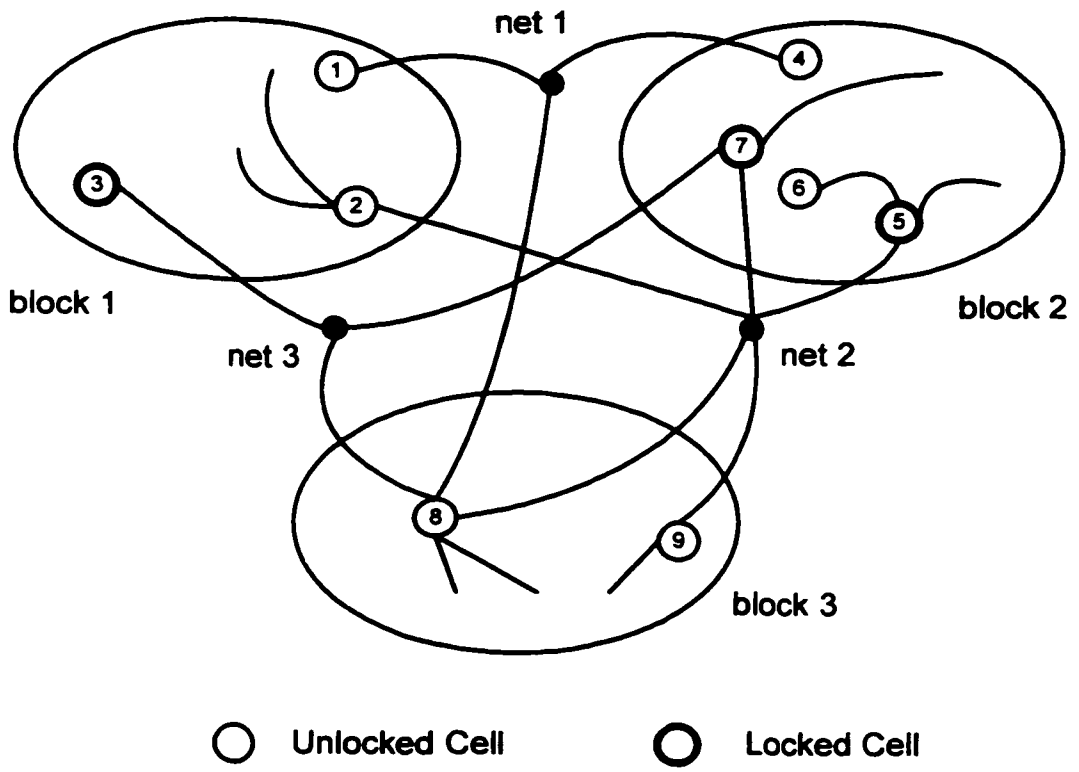


Figure 3: A hypergraph partitioning instance

Based on the expression of calculating the net_gain, we first calculate the numerator $\sum_{c \in S_L}[1 + d(c) - cut(c)]$. For net 2, there are two locked cells in locked block 2: cell 5

(degree = 3) and cell 7 (degree = 3). Among the three nets incident to cell 5, only one net (net 2) is in the cutset. Cell 7 has two nets (net 2 and net 3) in the cutset. So, $\sum_{c \in S_L}[1 + d(c) - cut(c)]$ is (1 + 3 – 1) + (1 + 3 – 2) = 5. The free block (block 1) of net 2 only has one free cell (cell 2 with degree = 3) and one net (net 2) is in the cutset. The denominator $\sum_{c \in S_F}[1 + d(c) - cut(c)]$ is (1 + 3 – 1) = 3. Therefore, we have: net_gain[2][1] = $\lceil \frac{5}{5} \times \frac{5}{3} \rceil = 2$.

Net 2 has two free cells (cell 8 and cell 9) in the other free block (block 3). The denominator is $\sum_{c \in S_F}[1 + d(c) - cut(c)]$ = (1 + 5 –3) + (1 + 2 – 1) = 5 and the numerator is the same as that for calculating net_gain[2][1]. We have:

$$\text{net\_gain}[2][3] = \lceil \frac{5}{5} \times \frac{5}{5} \rceil = 1.$$

Then for the free cell 2 of net 2, net_cell_gain[2][2] = net_gain[2][1] = 2; for free cells 8 and 9 of net 2, net_cell_gain[8][2] = net_cell_gain[9][2] = net_gain[2][3] = 1.

It is known that the range of each element of cell_gains is from ($-G_{max}$) to ($+G_{max}$) (G_{max} is equal to the product of the maximum cell degree and the maximum net weight) [9]. In the case of all net weights equal to one, this range is reduced to ($-\max_{c_i}[d(c_i)]$) to ($+\max_{c_i}[d(c_i)]$). Unlike the cell_gain, each element of a net_cell_gain array is always positive. From the expression for calculating the net gain, the maximum possible value of the net gain resulting from one loose net is $\{\max_{n_j}[d(n_j)]-1\} \cdot \max_{c_i}[d(c_i)]$. A free cell may

be on at most $\max_{c_i}[d(c_i)]$ loose nets at the same time, and these loose nets share the same locked block. Therefore, the maximum possible value max_net_gains of a net gain value is $\{\max_{n_j}[d(n_j)]-1\}\cdot\max_{c_i}[d(c_i)]^2$. Due to the introduction of net_cell_gain, the extended range for elements in a move_gain array ranges from $(-\max_{c_i}[d(c_i)])$ to $(\max_{c_i}[d(c_i)]$ + max_net_gains). That significantly reduces the opportunity of many cells having the same move gain, and makes the tie-breaking strategy such as look-ahead unnecessary.

It should be mentioned that in the process of calculating cell gain values, only the information of critical nets is used. We use the *net gain* concept to dynamically check the status of each net and to make the selection of the next cell to move more effective. The status of each net in the cutset is changed as follows: free → loose → locked or free → loose → disappear. Once a cell is moved, the status of the nets incident to the moved cell should be updated accordingly. If a net becomes locked, all elements of its net_gain array are set to zero immediately to avoid making wrong decisions in selecting cell moves.

4.3 How Net Gain Works (A Convincing Example)

The following simple example shows how the net gain concept is applied, and the advantage of embedding it in the selection of cell moves in our algorithm.

The simple circuit comprises 6 nets with 15 cells. We assume that all the cell weights and all the net weights have the value of 1. We solve the k-way partitioning with $k = 3$ here.

The number of cells in each block is limited in the range of 4 to 6 due to the balance constraint.

Figure 4 shows the initial solution with the cutsize of 5. At the beginning, all the values for the net_gain array and the net_cell_gain array are set to 0, and each move_gain value is set to the same as its corresponding cell_gain value. As defined previously, we select a candidate cell for moving based on its move_gain value. The following matrix (1) represents the 2-dimentional move_gain array, where each row corresponds to a cell and each column corresponds to a block:

$$\text{move\_gain} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{array} \begin{bmatrix} - & 0 & 0 \\ - & 0 & 0 \\ - & 0 & 0 \\ - & 0 & 0 \\ - & 1 & 0 \\ 0 & - & -1 \\ 1 & - & 0 \\ 0 & - & 1 \\ -1 & - & -1 \\ 0 & - & 0 \\ 0 & 0 & - \\ 0 & 0 & - \\ 0 & 0 & - \\ 0 & 0 & - \\ 0 & 0 & - \end{bmatrix} \quad (1)$$

In this case, cell 5, which has the highest move_gain value, is moved from block 1 to block 2 (Figure 4). Then, it is locked in block 2 for the current pass. This movement removes net 2 from the cutset, and hence reduces the cutsize from 5 to 4. Net 3 is the only other net incident to the moved cell 5, and it becomes a loose net. Based on the definition of a loose net, block 2 is the locked block, and block 3 is the free block for net 3. The net_gain value of net 3 for free block 3 is thus updated as follows:

$$\text{net\_gain}[3][3] = \left\lceil \frac{4}{4} \cdot \frac{(1+2-1)}{(1+1-1)+(1+2-2)} \right\rceil = 1.$$

Moreover, all the net_cell_gain values and the move_gain values corresponding to the free cells (cell 11 and cell 12) on net 3 for the free block 3 are updated according to the algorithm. The updated move_gain matrix is shown in (2).

At this stage, neither cell 11 nor cell 12 can be moved due to the balance constraint. The algorithm moves cell 7 from block 2 to block 1 (Figure 5), and locks the cell in block 1. The cutsize is then reduced to 3. Now, net 1 becomes the loose net. Since all the free cells of net 1 are in locked block 1, no further updating for this movement is needed. The next step is to move cell 8 to block 3 (Figure 5). The cutsize is further reduced to 2. Now, the move_gain matrix is shown in (3).

Without embedding the concept of net gain, as the conventional FMS, the cell_gain values for free cells (1, 2, 10, 11, 12, 13) have the same value of 0, and they are all legal moves (balance constraints are satisfied). Using the conventional approach, a cell is randomly selected for this situation. In the case of moving cell 10 from block 2 to block 1, the cutsize cannot be reduced further more.

Having the net_gain values as part of a cell's move_gain values, we obtain the move_gain matrix shown in (3). Both cells 11 and 12 have the same highest move_gain value of 1. We choose cell 11 to move to block 2 and lock it (Figure 6). (It is easy to see that either choice will lead to the same final solution.) Net 3 is still a loose net. Currently, there are two locked cells 5 and 11 in the locked block 2 and one free cell 12 in the free

block 3 for this loose net. Now, the net_gain[3][3] is $\lceil \frac{4}{4} \cdot \frac{(1+2-1)+(1+1-1)}{(1+2-2)} \rceil = 3$. The new move_gain matrix is shown in (4).

$$\text{move\_gain} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{array} \begin{bmatrix} - & 0 & 0 \\ - & 0 & 0 \\ - & 0 & 0 \\ - & 0 & 0 \\ - & - & - \\ -2 & - & -2 \\ 1 & - & 0 \\ 0 & - & 1 \\ -1 & - & -1 \\ 0 & - & 0 \\ 0 & 1 & - \\ 0 & 1 & - \\ 0 & 0 & - \\ 0 & 0 & - \\ 0 & 0 & - \end{bmatrix} \quad (2)$$

$$\text{move\_gain} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{array} \begin{bmatrix} - & 0 & 0 \\ - & 0 & 0 \\ - & -1 & -1 \\ - & -1 & -1 \\ - & - & - \\ -2 & - & -2 \\ - & - & - \\ - & - & - \\ -1 & - & -1 \\ 0 & - & 0 \\ 0 & 1 & - \\ 0 & 1 & - \\ 0 & 0 & - \\ -1 & -1 & - \\ -1 & -1 & - \end{bmatrix} \quad (3)$$

$$\text{move\_gain} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{array} \begin{bmatrix} - & 0 & 0 \\ - & 0 & 0 \\ - & -1 & -1 \\ - & -1 & -1 \\ - & - & - \\ -2 & - & -2 \\ - & - & - \\ - & - & - \\ -1 & - & -1 \\ 0 & - & 0 \\ - & - & - \\ 0 & 4 & - \\ 0 & 0 & - \\ -1 & -1 & - \\ -1 & -1 & - \end{bmatrix} \quad (4)$$

After moving cell 12 from block 3 to block 2 (Figure 6), the cutsize is reduced to 1. The optimal solution is obtained.

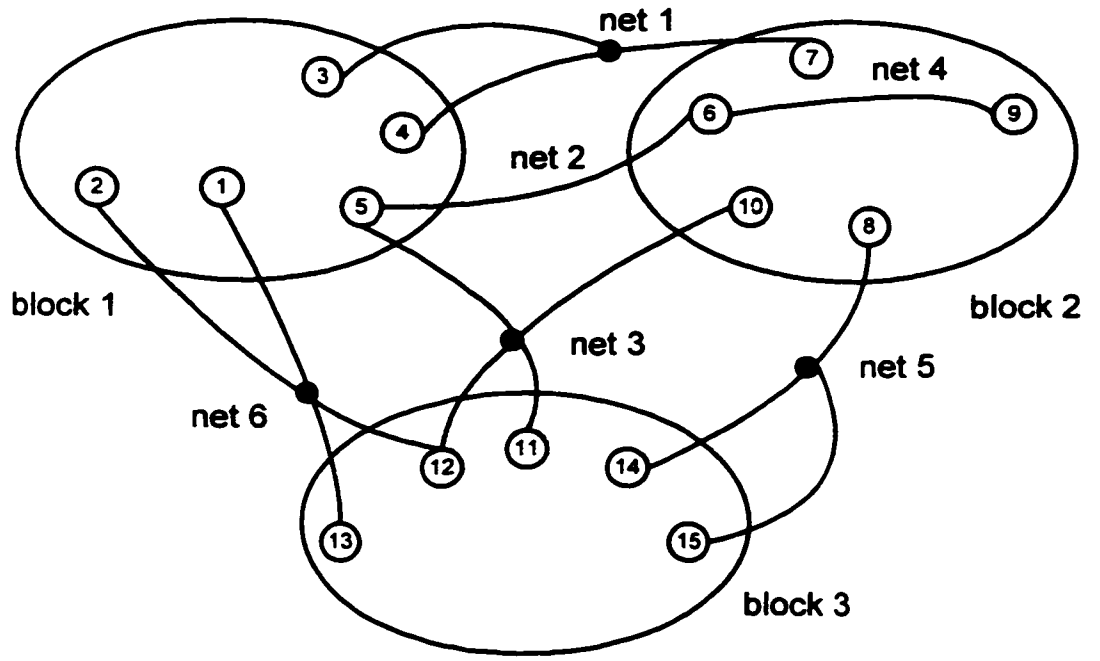


Figure 4: Example of using Net Gain (1)

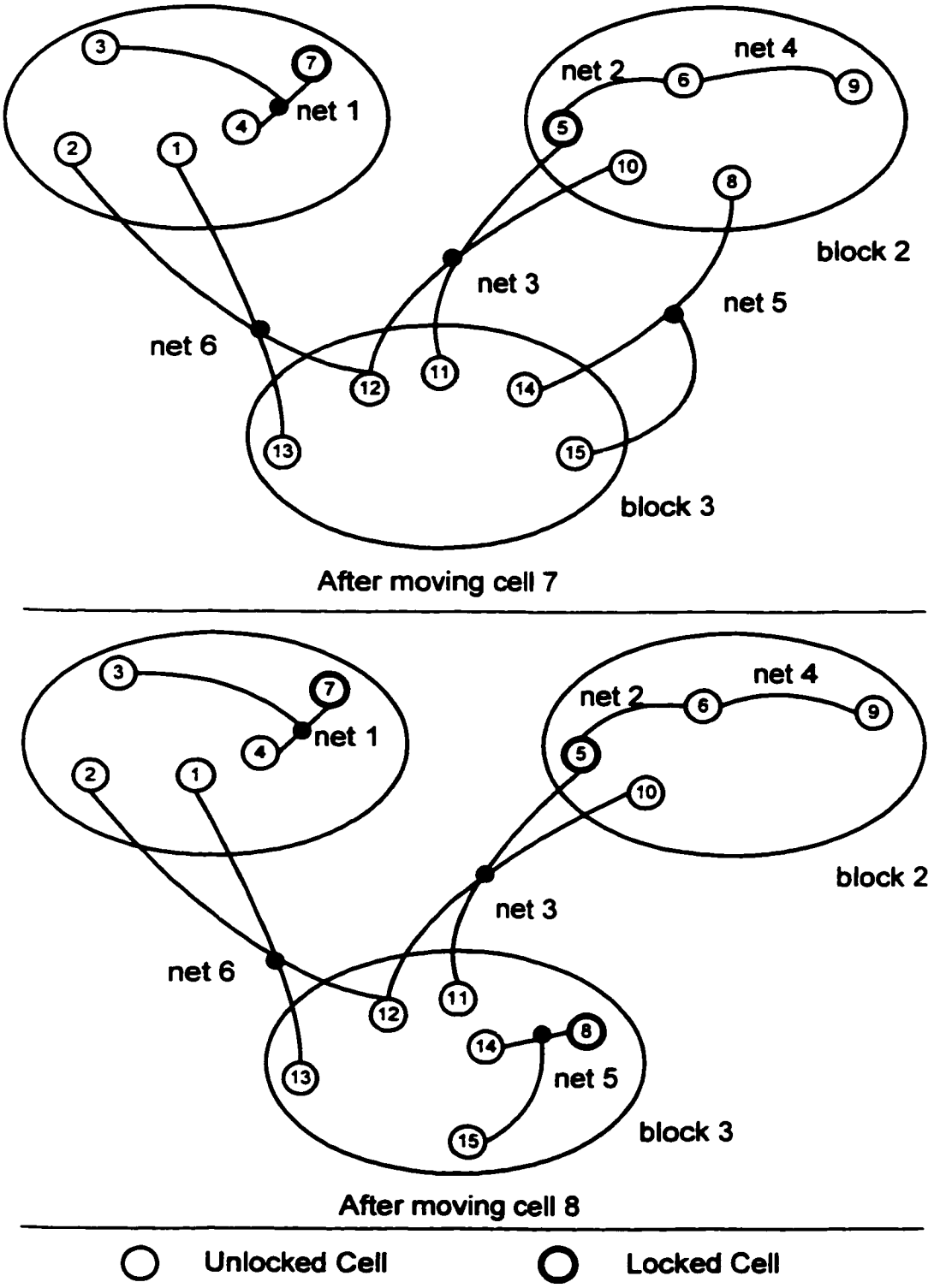
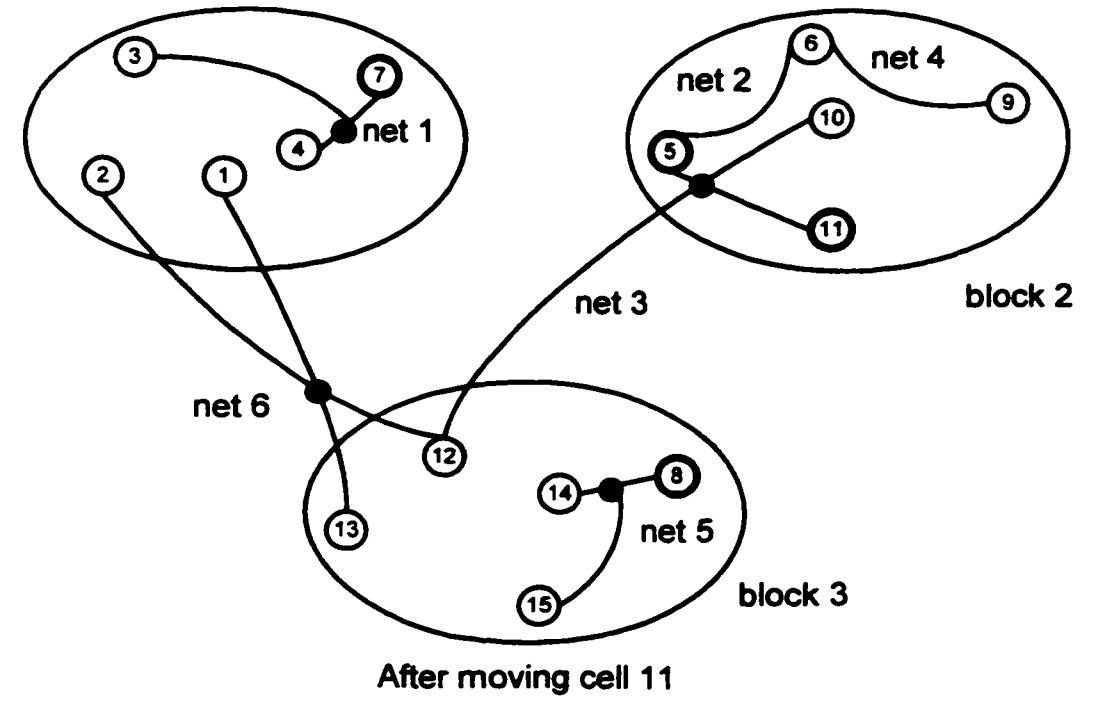
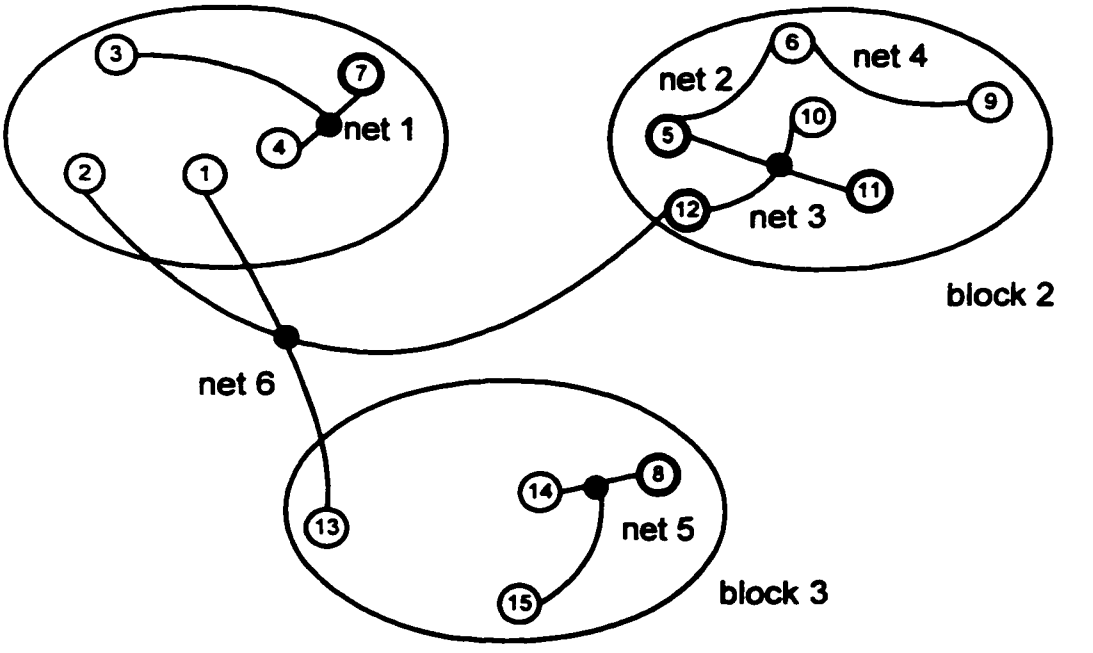


Figure 5: Example of using Net Gain (2)



After moving cell 11



After moving cell 12

Unlocked Cell Locked Cell

Figure 6: Example of using Net Gain (3)

4.4 Algorithm NGSP

Basically, the structure of our algorithm is similar to that of other FM-based iterative algorithms. The whole algorithm consists of a user specified number of runs. Each run comprises a number of passes that cannot be known in advance. We use the conventional locking mechanism in our algorithm. That is, each cell can only move at most once in each pass. Each pass has at most M_c iterations (cell moves). A local optimum is obtained at the end of each run. The algorithm outputs the best one from all the local optima at the end. The proposed algorithm with the *net gain* concept and a new solution perturbation scheme is abbreviated as NGSP (Net Gain Solution Perturbation) later.

In order to reduce the computational effort without significant degradation of the solution quality, we smoothly decrease the number of iterations from pass to pass by a fractional factor r. The experimental justification shown in the next section gives the evidence for indicating that most of the maximum partial gain sums are at the first half of the array of maximum partial gain sums of each pass; and, with the evolution from one pass to the next pass, the maximum partial gain sum is gradually moved to the start part of the array.

Additionally, to escape from being trapped in a local optimal solution, and to try to explore broader solution space, we perturb the current solution by the following scheme. Once a run is terminated, we find the common cut nets that are included in both the initial solution and the final solution of a run. These cut nets may be the obstacles for the solution being escaped from local optimum. We randomly force a percent amount of these cut nets to be removed from the current cutset by moving them one by one into the current smallest block if the balance constraint can be satisfied. The current solution

becomes the start point for the next run to explore new solution space. We also have experimental evidence to show the advantages of using the perturbation mechanism to improve the solution quality in next section.

Our algorithm consists of *NumOfRun* (given by user) runs. After *NumOfRun* runs are executed, it outputs the *finalCutsize* (minimum cutsize) and corresponding solution, and terminates.

The algorithm NGSP is shown in Figure 7 to Figure 9.

Algorithm NGSP

Input : *NumOfRun*: positive integer, used to define the number of runs;

r : floating-point number, $0 < r \leq 1$, used to decrease the number of moves from pass to pass;

p: floating-point number, $0 < p \leq 1$, used to choose a percentage of nets in the perturbation function;

function **main_function():**

Generate a random initial *k*-way partitioning as the initial solution;
finalCutsize = *localCutsize* = *currentCutsize*;
count = 0;
ratio=1;
start /* for each pass */
Compute cell_gain for each cell and initialize all cells as unlocked;
Initialize net_cell_gain and move_gain;
Build bucket list;
repeat
Choose a legal cell *c* with maximum move gain;
Make the cell move tentatively and lock it;

Update the cell_gain arrays and move_gain arrays of all affected cells;

Compute partial gain sum;

update_net_gain(*c*);

until the body has repeated $M_c \cdot ratio$ times or there are no legal moves available;

Find the maximum partial gain sum S_β;

if ($S_\beta > 0$) /* solution can be improved */

Make the first β cell moves permanent;

currentCutsize = *currentCutsize* - S_β;

localCutsize = *currentCutsize* ;

$ratio = ratio \cdot r$;

go to **start;**

else /* reached the local solution for current pass */

if (*finalCutsize* > *localCutsize*)

finalCutsize = *localCutsize*;

count = *count* + 1;

if *count* > *NumOfRun*

output the *finalCutsize* and corresponding solution, **stop;**

else

new_explore(*p*);

go to **start;**

Figure 7: The Algorithm NGSP

---

function **update_net_gain**(cell *c*)

/* updating net_gain arrays, net_cell_gain arrays and move_gain arrays*/

for each net n_a incident to the moved cell *c*

net_gain[n_α][1] = net_gain[n_α][2] = ...= net_gain[n_α][k] = 0

if net n_α is a loose net

for each free block B_F of n_α containing some free cells of n_α

Calculate net_gain[n_α][B_F];

for each free cell f of net n_α in block B_F

net_cell_gain[f][B_L] = net_cell_gain[f][B_L] + net_gain[n_α][B_F];

move_gain[f][B_L] = move_gain[f][B_L] + net_cell_gain[f][B_L];

Update bucket lists;

Figure 8: The update_net_gain Function

function **new_explore**(float p)

/* It is used to perturb the current local optimum solution for exploring new solution space. The perturbed solution is used as an initial solution for next run. */

commonNets = (cut nets in initial solution for the run) ∩ (cut nets in the current solution);

h = size of *commonNets*;

if $h = 0$

Output the *finalCutsize* and corresponding solution;

Stop.

else for ($j = 0$; $j \le p \cdot h$; j++) /* $0 < p \le 1$. */

Randomly take a net from *commonNets* and move all cells incident to it to the smallest block if the balance constraint is satisfied;

Figure 9: The new_explore Function

4.5 Experimental Justification for our Algorithm NGSP

We have conducted the following experiments on seven widely used ACM/SIGDA benchmark circuits: test06, primary2, avg_large, golem3, struct, biomed and industry2. We will give the detailed description of the benchmark circuits in Chapter 5. The following sections provide experimental evidence for the algorithm NGSP.

4.5.1 Decreasing the number of iterations from pass to pass to reduce the run time

As stated in the previous section, we decrease the number of iterations from pass to pass by a fractional factor r to reduce the overall run time needed by our algorithm.

The following experiments are to find where (at which moves) the maximum partial gain sum occurs for each pass. In these experiments, we take $r = 1$ (i.e., the maximum number of moves in each pass is equal to the number of cells) to see the exact number of moves before the maximum partial gain sums occur. The "move #"(move number) in Table 1 through Table 3 is bound by the number of cells. The experimental results for different values of k on three benchmark circuits are as follows. We use "mpgs" to denote the maximum partial gain sum.

Table 1: test06 for 10-way partitioning

pass #	1	2	3	4	5	6	7	8	9	10	11	12	13
mpgs	773	267	123	66	2	6	15	4	33	3	10	7	1
move #	1294	1215	1116	1086	1	39	14	914	157	55	21	20	1
pass #	14	15	16	17	18	19	20	21	22	23	24	25	26
mpgs	1	1	1	3	6	1	4	10	1	1	2	2	1
move #	1	1	1	60	41	2	26	71	10	2	21	10	4

Table 2: primary2 for 7-way partitioning

pass #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mpgs	1608	393	50	36	6	19	6	27	9	1	1	1	1	1	1	1	1
move #	2412	2293	1859	125	24	1665	14	1351	27	1	1	1	1	1	1	1	1
pass #	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**
mpgs	3	5	24	1	4	13	4	9	1	12	1	2	1	1	1	1	1
move #	15	44	81	5	240	97	60	8	6	94	4	2	1	1	1	17	3

Table 3: avg_large for 5-way partitioning

pass #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mpgs	16861	1042	1365	186	18	21	73	236	11	42	4	9	5	2	33	4	6
move #	20104	19109	17221	15693	82	50	14850	15708	60	244	37	23	5	1	137	4	5
pass #	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**
mpgs	6	2	5	10	2	2	4	1	2	7	3	5	3	1	1	37	4
move #	7	18	2	7	3	1	2	1	2	13	4	13	2	1	1	5673	3
pass #	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**
mpgs	15	12	2	7	3	3	2	11	6	9	4	4	13	1	8	7	2
move #	61	41	2	4	3	27	17	12	20	43	2	9	64	1	79	25	27
pass #	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**
mpgs	7	2	1	1	3	1	1	1	2	5	1	1	4	12	8	3	4
move #	4	2	4	3	4	4	2	1	61	3	1	1	37	104	9	22	43
pass #	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**	**85**
mpgs	6	1	1	1	1	2	1	2	2	1	1	1	5	9	21	2	1
move #	14	2	1	1	2	21	9	6	3	1	2	23	3	9	54	3	3
pass #	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**
mpgs	2	5	1	5	4	3	1	2	3	1	3	6	2	1	2	2	6
move #	2	3	2	10	7	5	3	2	2	1	3	10	1	1	7	3	9
pass #	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**
mpgs	4	1	5	5	2	2	2	2	5	2	2	2	1	2	2	2	1
move #	26	2	19	4	3	2	2	3	1	2	2	3	1	2	2	2	1
pass #	**120**	**121**	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**	**132**	**133**	**134**	**135**	**136**
mpgs	2	2	2	1	3	1	2	1	1	2	1	3	12	9	4	8	2
move #	2	2	2	1	4	7	3	1	1	4	2	61	14	7	91	10	2

pass #	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
mpgs	2	8	4	6	3	11	1	4	1	1	1	1	1	9	2	4	1
move #	12	5	7	3	2	48	5	18	1	1	1	3	1	10	37	7	2
pass #	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170
mpgs	1	1	3	1	6	2	2	4	1	1	1	1	1	1	1	10	1
move #	6	9	9	1	21	2	17	8	1	1	1	1	2	6	1	67	1
pass #	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187
mpgs	4	2	4	1	1	1	1	1	2	1	1	1	3	5	2	2	1
move #	2	4	6	2	1	1	1	2	2	3	2	1	9	14	2	4	39
pass #	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204
mpgs	1	1	5	4	1	1	3	4	4	6	2	2	5	1	6	2	1
move #	1	4	5	23	1	1	3	24	19	64	2	2	8	1	20	16	15

The experimental results show that for 81.8% (test06), 89% (primary2), 97.1% (avq_large) and 98.7% (golem3, table not shown) of the passes in a run, the maximum partial gain sums occur in the first half of the cell moves. With the increase of the number of cells in the circuits, the chance is significantly increased for the maximum partial gain sum to occur in the first half of the cell moves. With the evolution from pass to pass, the location of the maximum partial gain sum tends to be smaller and closer to the initial part of the cell moves. This is the reason why we introduce a parameter r to reduce the number of iterations from pass to pass. This strategy makes our algorithm very efficient for solving large circuit partitioning problems.

4. 5. 2 Improving solution quality with the perturbation mechanism

To show the evidence of the improvement in the solution quality by using the perturbation mechanism, we execute our algorithm s times (s initial solutions) with the

perturbation mechanism, each execution includes p runs; then we execute the algorithm ($s \cdot p$) times ($s \cdot p$ initial solutions) without the perturbation mechanism. We take the average cutsize over the s solutions for the first case, and the average cutsize over ($s \cdot p$) solutions for the second case. We set $r = 1$ and $p = 0.1$ for all the experimental cases. The experimental results on benchmark circuits struct ($k = 10$), primary2 ($k = 10$), biomed ($k = 7$), avq_large ($k = 5$) and golem3 ($k = 2$) are shown in Table 4 through Table 8 respectively.

Table 4: struct for 10-way partitioning

Experimental case	(1) With perturbation (each execution includes 5 runs)	(2) Without perturbation
Execution times	10	100
Average cutsize	156.8	183.5
Total run time (sec.)	86	137
Improvement of (1) over (2) in cutsize (%)	14.6	

Table 5: primary2 for 10-way partitioning

Experimental case	(1) With perturbation, each execution includes 10 runs	(2) Without perturbation
Execution times	10	100
Average cutsize	575.4	621.7
Total run time (sec.)	121	141
Improvement of (1) over (2) in cutsize (%)	7.4	

Table 6: biomed for 7-way partitioning

Experimental case	(1) With perturbation, each execution includes 5 runs	(2) Without perturbation
Execution times	10	80
Average cutsize	335.7	363.9
Total run time (sec.)	401	467
Improvement of (1) over (2) in cutsize (%)	7.7	

Table 7: avg_large for 5-way partitioning

Experimental case	(1) With perturbation, each execution includes 5 runs	(2) Without perturbation
Execution times	10	50
Average cutsize	889.8	999.9
Total run time (sec.)	4086	4633
Improvement of (1) over (2) in cutsize (%)	11	

Table 8: golem3 for bipartitioning

Experimental case	(1) With perturbation, each execution includes 3 runs	(2) Without perturbation
Execution times	10	30
Average cutsize	1799	1963.2
Total run time (sec.)	16391	23061
Improvement of (1) over (2) in cutsize (%)	8.3	

We conclude from Table 4 through Table 8 that the improvement of average cutsize for the algorithm with perturbation over that without perturbation is from 7.4% to 14.6% with almost the same run time. Therefore, the perturbation mechanism is necessary for improving the performance of our algorithm.

4.6 Parameter Settings

In our algorithm, the solution quality and the run time depend on the following three parameters.

4. 6. 1 The parameter *r*

As we have pointed out that *r* is used to reduce the run time by smoothly decreasing the number of iterations from pass to pass without significant degradation of the solution quality. It should be combined with the perturbation mechanism to make the algorithm more efficient and effective.

Table 9 through Table 12 show the influence of using different values for the parameter *r* on both the solution quality (cutsize) and the run time. In these experiments we run our algorithm 30 times for each value of *r*, and we take the average for each case.

Table 9: biomed for 7-way partitioning

r	average cutsize	average run time (sec.)	increase of average cutsize with respect to $r = 1$	ratio of average run time to that of $r = 1$
1	338.9	10.1	0	1
0.95	339.12	9.43	0.06 %	0.93
0.90	340.93	7.93	0.6 %	0.78
0.80	353.53	6.73	4.2 %	0.66
0.65	352.5	6.1	3.9 %	0.6

Table 10: industry2 for 10-way partitioning

r	average cutsize	average run time (sec.)	increase of average cutsize with respect to $r = 1$	ratio of average run time to that of $r = 1$
1	1564.2	92.13	0	1
0.98	1599.27	65.65	2.1 %	0.71
0.95	1616.8	47.73	3.2 %	0.52
0.90	1630.5	33	4 %	0.35
0.85	1720.8	28.7	9 %	0.31

Table 11: avg_large for 5-way partitioning

r	average cutsize	average run time (sec.)	increase of average cutsize with respect to $r = 1$	ratio of average run time to that of $r = 1$
1	861.03	203.13	0	1
0.98	861.9	156.3	0.1%	0.77
0.95	919.7	144.4	6.3 %	0.71
0.90	946.5	119.9	8.9 %	0.32
0.85	992.9	120.3	13.2 %	0.29

Table 12: golem3 for 5-way partitioning

r	average cutsize	average run time (sec.)	increase of average cutsize with respect to $r = 1$	ratio of average run time to that of $r = 1$
1	4045.1	1343.3	0	1
0.98	4176.8	522.7	3.1 %	0.39
0.95	4623.6	321.4	12.5 %	0.24
0.90	4764.9	246.8	15 %	0.18
0.85	4933.3	225.9	18 %	0.17

Each table of Table 9 through Table 12 indicates that the average cutsize increases if we decrease the value for r. The tables also show that the run time is reduced with the increase of the value for r. Comparing different tables, we see that the larger a benchmark circuit is, the bigger the influence of reducing the same amount on r has on both the average cutsize and the average run time. For example, when r reduces from 1 to 0.90, the increase of the average cutsize with respect to $r = 1$ are 0.6% (for biomed), 4% (for industry2), 8.9% (for avq_large) and 15% (for golem3); and the ratios of average run time to that of $r = 1$ are 0.78 (for biomed), 0.35 (for industry2), 0.32 (for avq_large) and 0.18 (for golem3). For the benchmark circuit golem3, there are more than 300 passes in a run. If we choose $r = 0.98$, the number of iterations for the 35^{th} pass is nearly half of that

for the first pass. The cutsize is increased by 3.1 % and the run time is reduced by 41% of that of $r = 1$. From this observation, we need to choose larger r for large benchmarks to avoid high degradation of the solution quality. Also we need to use the perturbation mechanism to guarantee the solution quality with reasonable run time.

4.6.2 The number of perturbations: *NumOfRun*

Our algorithm uses the perturbation mechanism to escape from local optima and to explore broader solution space. Theoretically, the solution quality is enhanced if we increase the value for the parameter *NumOfRun*. However, the improvement rate of the cutsize tends to be decreased while the value of *NumOfRun* is increased. The experimental evidence for bipartitioning on benchmark circuit primary2 in Table 13 is used for illustration. We set $r = 0.98$, $p = 0.2$ and the results are the averages over 20 runs. Table 13 shows that the improvement (%) in cutsize for one unit increase in run time tends to be declined with the increase of the number of perturbations. It implies that as the increase of the value of *NumOfRun*, we have to spend more run time to obtain the same percent of improvement in cutsize. The appropriate *NumOfRun* is chosen based on the tradeoff between the solution quality and the run time.

Table 13: The cutsize and the run for different number of perturbations

NumOfRun	average cutsize	(1) improvement of cutsize for current *NumOfRun* over that of previous *NumOfRun* (%)	average run time (sec.)	(2) ratio between run times for current *NumOfRun* and previous *NumOfRun*	(1)/(2) the improvement in cutsize for one unit increase in run time (%)
1	194.85	-	3.15	-	-
2	173.6	10.9	5.2	1.65	6.6
4	163.85	5.6	8.3	1.59	3.52
8	155.15	5.3	13.8	1.66	3.19
16	146.1	5.8	25.8	1.87	3.10

4.6.3 The perturbation ratio: *p*

The perturbation ratio p ($0 < p \leq 1$) plays an important role in the perturbation mechanism. To escape from a current local optimum, we force up to p of the common cut nets (those nets are in both the initial and the final solutions) of a run out of the cutset and start a new run. A small p will lead to more thorough searches around current solution space, whereas a large p will enlarge the search range into a broader solution space with more run time. We find that the appropriate p values range from 0.05 to 0.20.

As an example to show how the parameters r, p and *NumOfRun* are determined, the experimental results of settings the parameters for the 5-way partitioning on benchmark circuit industry2 are shown in Table 14 through Table 17. We execute the algorithm 20 times for each case and take the averages over these 20 runs. The parameters r, *NumOfRun* and p are to be determined. First, we temporally set *NumOfRun* = 3, $p = 0.05$ and use different values of r. The experimental results are shown in Table 14.

Table 14: Choose the parameter *r*

r	average cutsize	best cutsize	average run time (sec.)
1	857.85	741	196.15
0.98	894.35	683	102.85
0.95	922.5	763	71.4
0.90	957.35	778	56.15
0.85	1006.4	829	50.7

Based on the tradeoff between the cutsize and the run time from Table 14, we choose r = 0.98. Although the cutsize of 857.85 for $r = 1$ is 4% better over the cutsize of 894.35 for r = 0.98, its run time is almost twice of that for the later case. We then need to determine p

by set $r = 0.98$ and *NumOfRun* = 3. The experiment results for different values of p are shown in Table 15.

Table 15: Choose the parameter p

p	average cutsize	best cutsize	average run time (sec.)
0.025	896.55	683	102.65
0.05	894.35	683	102.85
0.10	912.35	722	111.25
0.15	938.45	813	107.65

From Table 15, we choose the value for p ($p = 0.05$) that gives the best average cutsize. The last parameter to be determined is *NumOfRun*. Table 16 shows the results for *NumOfRun* = 2, 3, 4, and 5 by setting $r = 0.98$ and $p = 0.05$. We choose *NumOfRun* = 5, and get the cutsize of 842.8 with 148.15 seconds of run time which outperforms the case of $r = 1$, $p = 0.05$ and *NumOfRun* = 3 both in cutsize and run time (Table 14).

Table 16: Choose the parameter *NumOfRun*

NumOfRun	average cutsize	best cutsize	average run time (sec.)
2	953.75	780	86.55
3	894.35	683	102.85
4	871.05	740	128.05
5	842.8	714	148.15

4.7 Data Structure and Complexity Analysis

We use dynamic lists to store a circuit. One list stores the cell information, one for the net information, and one for the block information. The bucket array data structure [9] is also used in our algorithm. Basically, a bucket array (see Figure 1) has an array of pointers and each of these pointers points to a doubly linked list. An upper bound of the bucket array size is set to be a user-defined parameter u, and, as explained in section 4.2, u falls in the

range of $[\max_{c_i}[d(c_i)],\ \max_{c_i}[d(c_i)]+\{\max_{n_j}[d(n_j)]-1\}\cdot\max_{c_i}[d(c_i)]^2]$. In partitioning large circuits, we use the parameter u to overcome the limitation on system memory and to reduce the run time. The size of a bucket array is therefore $(\max_{c_i}[d(c_i)]+u+1)$. There are $(k-1)$ moving directions for each cell, and there are k blocks in the partition. Therefore, we need to have $k(k-1)$ such bucket arrays.

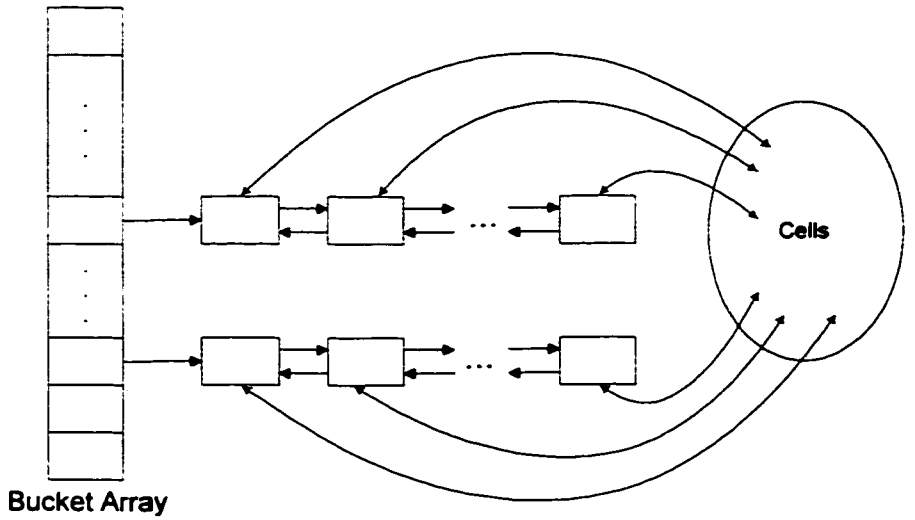


Figure 10:The bucket data structure

Based on the experimental results from [15], we also employ the LIFO (Last In First Out) structure for the doubly linked list. That means the insertion or deletion is done at the head of each doubly linked list. For each bucket array, we store the maximum array index at which it contains a non-empty doubly linked list. This is to ensure constant time access to the best moves in each bucket array. If there is more than one move with the same maximum move gain, we select the one at the head of a doubly linked list. To find a move with maximum move gain, we search all the $k(k-1)$ bucket arrays, and, select the

first such move encountered during the process. Finding a node among $k(k-1)$ bucket arrays need $O(k^2)$ time. A single insertion or deletion can be done in constant time on the bucket list. If a removal empties a doubly linked list at the maximum index on a bucket array, we have to spend $O(u)$ time to update the variable that stores the maximum non-empty index.

We look the update_net_gain function first. The outer **for** loop in this function takes $O(\max_{c_i}[d(c_i)])$ time since a cell may have at most $\max_{c_i}[d(c_i)]$ nets on it. Initializing net gain values needs $O(k)$ time since there are k net gain values for each net. In case a net is a loose net, we need to recompute the net gain values. It needs $O(\max_{n_j}[d(n_j)])$ to compute each net gain value and we have to compute it at most $O(k)$ times. The most inner **for** loop that updates move gain array takes $O(\max_{n_j}[d(n_j)])$ time since its body takes constant time. Therefore, it takes $O(k\max_{n_j}[d(n_j)])$ time to calculate the net gain array and update the move gains and the bucket lists. The update_net_gain function can be done in $O(k\max_{c_i}[d(c_i)]\cdot\max_{n_j}[d(n_j)])$ time.

As other FM-based approaches, we need to compute the cell gain and initialize move gain for each initial solution. As stated in Chapter 2, this step takes $O(M_p k)$ time. Creating the bucket array lists can be done in $\Theta(k^2 u)$ since there are $k(k-1)$ different possible moves and each bucket array has $(\max_{c_i}[d(c_i)]+u+1)$ doubly linked lists. One insertion can be done in constant time on the bucket array. Therefore, inserting all cell nodes into the bucket array lists takes $O(M_c k)$ time. Selecting a legal candidate from $k(k-1)$ arrays needs $\Theta(k^2)$. Locking the selected cell takes constant time, and removing cell nodes from

the bucket array lists takes $O(ku)$. By adapting the locking mechanism, the gain updating procedure after each cell move needs $O(\frac{M_p ku}{M_c})$ time. Since a program structure is employed inside the repeat loop that tracking the partial gain sum subsequence, finding the maximum partial gain sum can be done in constant time. The **repeat** loop takes $O(M_c k^2 + M_p ku + M_c k \max_{c_i}[d(c_i)] \cdot \max_{n_j}[d(n_j)])$.

Overall, NGSP has the complexity of $O(k^2 u + M_c k^2 + M_p ku + M_c k \max_{c_i}[d(c_i)] \cdot \max_{n_j}[d(n_j)])$ per pass. The total number of passes for one run can not be known in advance though the maximum number of passes recorded in our experiments is around 300.

Chapter 5

Experimental Studies

This chapter presents the details of the experimental framework and lists the experimental results. We conduct the experimental studies for our algorithm in comparison with FMS, DA (including six versions: PLM1, PLM2, PLM3 and PFM1, PFM2, PFM3) algorithms. Like [9], the level parameter of FMS is set to one. We chose PLM3 and PFM3 among all the algorithms of DA for comparison since they produced better results than other versions of PLM and PFM respectively in [9]. The performance comparisons have been done on seven widely used ACM/SIGDA benchmark circuits. All the algorithms are implemented using the C++ programming language and all experiments are done on a 433MHz Pentium Celeron based Windows NT 4 workstation with 128M physical memory.

Like [9], to make a fair comparison, all algorithms are performed with a random initial solution and the same balance criterion among all blocks (the balance tolerance of 0.1). We assume for simplicity and comparison with other algorithms that all weights for cells and nets are set to 1. The results produced by our implementation of the algorithms PLM, PFM and FMS are consistent (<1% difference) with the ones produced from the original implementation done by Ali Dasdan[9].

5.1 Benchmark Circuits

Table 1 shows the seven benchmark circuits that are used for our performance comparisons. Among these circuits, the number of cells ranges from 1752 to 103048 and the circuit density ranges from 0.00449 to 0.07983.

Table 17: The Benchmark Suits

Benchmark Circuit	M_c	M_n	M_p	avc.$d(c)$	avc.$d(n)$	max.$d(c)$	max.$d(n)$	D
test06	1752	1641	6638	3.79	4.05	6	388	0.079830
struct	1888	1888	5375	2.85	2.85	4	16	0.004490
primary2	3014	3029	11219	3.72	3.70	9	37	0.008204
biomed	6417	5711	20912	3.26	3.66	6	860	0.062038
industry2	12142	12949	47193	3.89	3.64	12	584	0.011770
avg_large	25678	25384	82751	3.29	3.26	7	4042	N/A
golem3	103048	144949	338419	3.28	2.33	22	39	N/A

5.2 Performance Comparisons

The results obtained by FMS, PLM3, PFM3, and NGSP on seven benchmark circuits for four partitions (k = 2, 5, 7, 10) are shown in Table 18 (for the average cutsize and the standard deviation) and Table 19 (for the best cutsize and the worst cutsize) respectively. The bold values are the best one in each row. For all benchmark circuits in this study, we run FMS 100 times. We run PLM3, PFM3 and NGSP 50 times for each of the benchmark circuits test06, struct, primary2 and biomed. All the algorithms, except FMS, are executed 20 times for the benchmark circuits industry2 and avg_large. For the largest benchmark circuit golem3, we execute PLM3, PFM3 and NGSP 10 times each.

Table 18: The average cutsize and (the standard deviation)

Benchmark	k	FMS	PLM3	PFM3	NGSP
test06	2	82.7 (10.6)	79.2 (6.2)	89.1 (9.4)	67.7 (7.0)
	5	298.2 (24.8)	207.2 (19.6)	158.4 (20.0)	133.0 (8.7)
	7	336.2 (20.7)	240.2 (22.2)	183.8 (25.7)	139.74 (9.01)
	10	375.5 (16.9)	264.4 (19.0)	210.7 (28.2)	177.06 (12.59)
struct	2	51.3 (4.3)	49.9 (3.7)	50.6 (11.2)	34.9 (2.9)
	5	311.3 (27.3)	209.2 (23.9)	151.1 (27.7)	91.2 (5.7)
	7	400.4 (30.1)	309.0 (31.7)	235.1 (33.4)	116.4 (7.0)
	10	503.4 (30.0)	421.7 (26.0)	339.7 (33.6)	155.3 (11.1)
primary2	2	272.2 (40.0)	257.7 (44.6)	240.8 (28.5)	159.0 (22.4)
	5	874.4 (27.0)	738.7 (30.5)	512.7 (24.0)	430.41 (18.4)
	7	952 (24.2)	829.5 (26.3)	615.9 (29.3)	500.74 (17.72)
	10	1028.9 (21.3)	876.4 (23.9)	746.5 (23.3)	584.2 (13.7)
biomed	2	128.8 (47.0)	174.6 (17.6)	192.8 (25.3)	87.7 (4.6)
	5	714.4 (56.5)	573.4 (36.1)	487.5 (19.0)	263.6 (19.7)
	7	845.9 (44.6)	686.3 (33.1)	588.4 (23.2)	324.2 (15.7)
	10	940.5 (29.7)	817.7 (29.6)	729.3 (23.6)	378.9 (12.6)
industry2	2	633.9 (154.4)	624.4 (173.1)	690 (88.1)	232.9 (41.2)
	5	2750.4 (109.5)	2002.9 (113.9)	1368.4 (104.1)	842.8 (72.63)
	7	2995.5 (78.9)	2156.3 (105.8)	1656.7 (106.7)	1206.35 (113.76)
	10	3091.3 (77.8)	2306.2 (57.1)	1750.2 (104.7)	1487.55 (78.09)
avg_large	2	803.4 (158.6)	880.2 (77.9)	611.3 (49.9)	412.9 (96.92)
	5	3992.5 (114.4)	2215.6 (79.2)	1478.4 (69.1)	835.3 (64.3)
	7	4608.4 (78.8)	2816.6 (66.1)	1975.6 (62.9)	948.3 (42)
	10	5081.3 (88.39)	3661.1 (101.6)	2758.9 (131.1)	1200.45 (53.36)
golem3	2	3299.8 (289.7)	4012 (301.0)	3208.3 (198.7)	1607.1(170.74)
	5	23492 (416.9)	9846.8 (545.3)	5931*	3878.9 (231.01)
	7	27177 (453.9)	12320 (411)	N/A	4552.9 (241.38)
	10	29010 (328.8)	N/A	N/A	5264.4 (155.77)

* We only execute one time with run time = 28223 seconds.

Table 19: The best cutsize and (the worst custsize)

Benchmark	k	FMS	PLM3	PFM3	NGSP
test06	2	**60** (109)	68 (89)	69 (109)	**60 (82)**
	5	238 (347)	170 (249)	113 (213)	**110 (149)**
	7	289 (372)	191 (287)	133 (240)	**121 (160)**
	10	335 (408)	217 (302)	151 (2622)	**150 (202)**
struct	2	43 (63)	43 (55)	**33** (75)	**33 (45)**
	5	226 (364)	137 (252)	90 (233)	**82 (105)**
	7	334 (492)	228 (378)	154 (312)	**99 (139)**
	10	439 (595)	367 (485)	267 (409)	**134 (176)**
primary2	2	180 (373)	173 (353)	171 (311)	**139 (215)**
	5	809 (933)	665 (804)	463 (570)	**381 (463)**
	7	879 (1024)	772 (882)	536 (683)	**459 (533)**
	10	977 (1083)	810 (918)	703 (797)	**549 (611)**
biomed	2	**83** (279)	130 (215)	140 (231.3)	**83 (104)**
	5	574 (841)	497 (637)	428 (531)	**222 (307)**
	7	682 (935)	612 (744)	534 (645)	**292 (366)**
	10	784 (997)	797 (894)	683 (790)	**352 (404)**
industry2	2	282 (1067)	386 (941)	454 (845)	**190 (354)**
	5	2485 (2963)	1710 (2176)	1175 (1583)	**714 (967)**
	7	2809 (3154)	1958 (2361)	1481 (1835)	**952 (1354)**
	10	2847 (3276)	2205 (2408)	1582 (1921)	**1309 (1650)**
avg_large	2	359 (1112)	768 (1019)	519 (696)	**185 (476)**
	5	3593 (4205)	2026 (2318)	1394 (1580)	**712 (973)**
	7	4438 (4816)	2745 (2933)	1887 (2073)	**887 (1059)**
	10	4831 (5272)	3501 (3811)	2615 (2895)	**1103 (1289)**
golem3	2	2624 (3886)	3491 (4542)	2972 (3607)	**1417 (1927)**
	5	22606 (24388)	8999 (10931)	5931*	**3396 (4166)**
	7	26409 (28290)	11727 (12944)	N/A	**4125 (4815)**
	10	28577 (29580)	N/A	N/A	**5022 (5478)**

*We only execute once with run time = 28223 seconds.

For the large benchmark circuit golem3, the cutsize for PLM3 with $k = 10$, the cutsize for PFM3 with $k = 7$ and $k = 10$, are not available due to the unacceptable run time which is much more than 28800 seconds (8 hours) for one execution.

We can conclude from Table 18 and Table 19 that for all the 28 instances (7 benchmark circuits multiplied by 4 different partitions), NGSP always significantly outperforms FMS, PLM3 and PFM3 in terms of the solution quality (best cutsize, average cutsize and worst cutsize). For standard deviation, NGSP also gets the minimum values for most (86%) of the instances. It implies that NGSP is the most stable one among all the four algorithms since its final solutions do not heavily depend on the initial solutions.

Table 19 shows that for all the 28 instances, NGSP always has the minimum best cutsize, and the gaps in the best cutsize between NGSP and FMS, PLM3 or PFM3 are dramatically increased with the increase in the circuit size. For example, for benchmark circuits biomed, industry2, avq_large and golem3, the worst cutsizes produced by NGSP is even smaller than the corresponding best cutsizes produced by FMS, PLM3 and PFM3 for all instances, except for the instance $k = 10$ on circuit industry2, where the worst cutsize of NGSP is 1650 and the best cutsize of PFM3 is 1582. For the instance of $k = 10$ on circuit avq_large, the best cutsize produced by FMS, PLM3 and PFM3 is 3.74, 2.71 and 2.02 times of the worst cutsize produced by NGSP respectively.

Derived from the values in Table 18, the average improvements of the average cutsize produced by NGSP over FMS, PLM3 and PFM3 for $k = 2, 5, 7, 10$ are shown in Table 20. From Table 20, we also observe that the solution quality of both PLM3 and PFM3 is

worse than that of FMS for bipartitioning. This observation is consistent with that in [9]. However, NGSP beats FMS in cutsize significantly for bipartitioning.

Table 20: Quality Comparison: Avg. improvements(%) of the avg. cutsize

k	FMS	PLM3	PFM3
2	40.91	44.03	41.79
5	67.51	52.67	33.84
7	65.87	49.63	36.20
10	62.12	44.23	35.22

Based on Table 19, the average improvements of the best cutsize of NGSP over FMS, PLM3 and PFM3 for $k = 2, 5, 7, 10$ are shown in Table 21. It shows that, in terms of the best cutsize, FMS also beats PLM3 and PFM3 for bipartitioning. But the rank of the best cutsize among these three algorithms is PFM3, PLM3 and FMS for multiway partitioning.

Table 21: Quality Comparision: Avg. improvements(%) of the best cutsize

k	FMS	PLM3	PFM3
2	25.3	39.55	35.31
5	66.53	50.97	29.15
7	65.94	52.48	31
10	62.45	64.31	32.6

Table 22 shows the average run time for different k values on seven benchmark circuits. Table 23 shows the total amount of run times required by FMS (28 instances), PLM3 (27 instances), PFM3 (26 instances) and NGSP (28 instances), and Table 24 shows the ratio of total amount of run time required by FMS, PLM3 and PFM3 with respect to NGSP. It can be seen that FMS takes the smallest run time (around 0.62 times of that for NGSP), PFM3 is far larger than NGSP (8.75 times of NGSP) and PLM3 takes 2.67 times of that needed by NGSP. For 10-way partitioning on benchmark circuit industry2, the run time of PFM3 is 28 times that of NGSP. For 5-way partitioning on benchmark circuit golem3, the run time of PFM3 is 24.77 times that of NGSP. It should be pointed out that the run time for large benchmark circuits is also affected by the limitation of physical memory installed in the test machine since hard disk swapping is needed.

Table 22: Average run times (seconds)

Benchmark	*k*	FMS	PLM3	PFM3	NGSP
test06	2	0.35	0.80	1.48	9.98
	5	0.47	11.08	27.68	11.96
	7	0.56	28.70	67.52	18.9
	10	0.70	76.36	165.32	22.76
struct	2	0.35	0.62	1.56	2.70
	5	0.50	10.90	24.44	4.28
	7	0.57	27.32	59.28	5.48
	10	0.76	59.18	158.10	7.08
primary2	2	1.02	2.60	4.58	5.80
	5	1.03	25.08	72.68	28.74
	7	1.28	66.04	159.70	35.96
	10	1.52	182	309.24	20.60
biomed	2	2.02	3.30	13.78	14.30
	5	2.43	48.48	98.68	30.56
	7	2.75	131	211.42	38.74
	10	3.58	373	631.24	53.40
industry2	2	5.12	12.18	43.15	72.6
	5	6.34	122.50	514.90	148.15
	7	8.71	327.54	1396.75	126.35
	10	11.05	818.21	4507.95	160.95
avg_large	2	10.91	18.40	99.95	256.7
	5	14.79	195.80	989.20	356.55
	7	18.42	532	2419.60	509.45
	10	20.85	3073	5508.25	649.15
golem3	2	82.60	147	1701	848.9
	5	145.05	2715	28223	1139
	7	160.03	7200	N/A	1455
	10	228.5	N/A	N/A	2149.9

Table 23: Total run time (seconds)

FMS	PLM3	PFM3	NGSP
73226 (28)	294935.8 (27)	709170 (26)	81039 (26), 95589 (27), 117079 (28)

Table 24: The ratio of total run time with respect to NGSP

FMS	PLM3	PFM3	NGSP
0.625	2.67	8.75	1

The previous experimental results show that our algorithm significantly outperforms DA (both PLM and PFM) in solution quality with much less run time. Although FMS is the fastest algorithm, its very poor solution quality cannot be accepted. The large run time for PLM3 and PFM3 makes them unable to solve the multiway partitioning problem for large circuits (for example, golem3).

Chapter 6

Conclusion

In this thesis, we have proposed an effective multiway partitioning algorithm called NGSP. We introduced the concept of net gain and embeded it in the selection of cell moves. If a cell is moved to a block and locked there, we used the net gain values to encourage its neighboring cells to be moved subsequently to the block where the moved cell is just locked in. The net that straddles the cut line is thus removed. Unlike the conventional FM-based iterative improvement algorithms in which the selection of the next cell to move is only based on its cell gain, our algorithm selects a cell to move based on both its cell gain and the sum of all net gains for those loose nets incident to the cell (we call it move gain). Due to the introduction of the net gain, the extended range of the move gain reduces the opportunity of many cells having the same value and makes the tie-breaking strategy such as look-ahead unnecessary.

Based on our experimental justification, we find that, with the evolution from pass to pass, the location of the maximum partial gain sum tends to be closer to the initial part of the cell moves. Therefore, we smoothly decrease the number of iterations from pass to pass to reduce the computational effort without significant degradation of the solution quality. This enables our algorithm to partition large benchmark circuits with reasonable run time.

To escape from local optima and to search for broader solution space, we proposed a perturbation mechanism. Once a run is terminated, the cut nets that are included in both the initial solution and the final solution of the current run may be the obstacles for the solution being escaped from local optima. We randomly force some of these cut nets to be removed from the current cutset. The perturbed solution becomes the initial solution for the next run to explore new solution space. Our experimental justification shows that this perturbation mechanism is necessary for improving the performance of our algorithm.

Compared with the multiway hypergraph partitioning algorithms proposed by Dasdan and Aykanat [9], our algorithm significantly outperforms theirs in terms of average cutsize and run time: the average improvement over theirs PLM3 and PFM3 are 47.64 % and 36.76% with only 37.17% and 9.66% run time respectively. Although FMS is a very fast algorithm, its very poor solution quality cannot be accepted. The large run time for PLM3 and PFM3 makes them unable to solve multiway partitioning for large circuits.

Our ongoing research will focus on seeking for an adaptive scheme to make the reduction of the number of moves more efficiently and try to find a refined perturbation mechanism based on the improvement amount between two consecutive runs to guide the searching for the solution space more effectively.

References

[1] C. J. Alpert and A. B. Kahng, Recent directions in netlist partitioning, *Integration, the VLSI Journal,* vol. 19, pp. 1-81, 1995.

[2] C. J. Alpert, J. H. Huang and A. B. Kahng, Multilevel circuit partitioning, *Proc. ACM/IEEE Design Automation Conf.,* pp. 530-533, 1997.

[3] C. Berge, Graphs and Hypergraphs, *American Elsevier, New York,* 1976.

[4] T. Bui and C. Jones, A heuristic for reducing fill in sparse matrix factorization, *6th SIAM Conf. Parallel Processing for Scientific Computing,* pp. 445-452, 1993.

[5] P. K. Chan, M. D. F. Schlag, and J. Z. Zien, Spectral k-way ratio-cut partitioning and clustering, *IEEE Trans. on Computer-Aided Design,* vol. 13, no.8, pp.1088-1096, 1994.

[6] J. Cong, P. Li, S. Lim, T. Shibuya and D. Xu, Large scale circuit partitioning with loose/stable net removal and signal flow based clustering, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design,* pp. 441-446, Nov. 1997.

[7] J. Cong and Lim, Multiway partitioning with pairwise movement, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design,* pp. 512-516, 1998.

[8] J. Cong and M. L. Smith, A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design, *Proc. ACM/IEEE Design Automation Conf.,* pp.755-760, 1993.

[9] A. Dasdan and C. Aykanat, Two novel multi-way circuit partitioning algorithms using relaxed locking, *IEEE Trans. on Computer-Aided Design,* vol.16, no.2, pp. 169-178, 1997.

[10] S. Dutt and W. Deng, A probability-based approach to VLSI circuit partitioning, *Proc. ACM/IEEE Design Automation Conf.,* pp.100-105, 1996,

[11] S. Dutt and W. Deng, VLSI circuit partitioning by cluster-removal using iterative improvement techniques, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp.92-99, Nov. 1996.

[12] C. M. Fiduccia and R. M. Mattheyses, A linear-time heuristic for improving network partitions, *Proc. ACM/IEEE Design Automation Conf.*, pp.175-181, 1982.

[13] M. R. Gray and D. S. Johnson, Computers and Intractability: A Guide to the theory of NP-completeness, *W. H. Freeman, San Francisco*, CA, 1979.

[14] L. W. Hagen, D. J. Huang and A. B. Kahng, On implementation choices for iterative improvement partitioning algorithms, *Proc. European Design Automation Conference*, pp. 144-149. 1995.

[15] S. Hauck and G. Borriello, An evaluation of bipartitioning techniques, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 849-866, 1997.

[16] B. Hendrickson and R. Leland, A multilevel algorithm for partitioning graphs, *Technical Report, SAND93-1301, Sandia National Laboratories*, 1993.

[17] A. G. Hoffmann, The dynamic locking heuristic –a new graph partition algorithm, *Proc. IEEE Int. Symp. on Circuits and Systems*, pp.173-176, 1994.

[18] D. J. H. Huang and A. B. Kahng, Multi-way system partitioning into single or multiple type FPGAs, *Proc. ACM/SIGDA International Workshop on Field-Programmable Gate Arrays*, pp. 140-145, 1995.

[19] D. J. H. Huang and A. B. Kahng, When clusters meet partitions: new density-based methods for circuit decomposition, *Proc. European Design and Test Conf.*, pp. 60-64, March, 1995.

[20] F. M. Johannes, Partitioning VLSI circuits and systems, *Proc. ACM/IEEE Design Automation Conf.*, pp. 83-87, 1996.

[21] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, Multilevel hypergraph partitioning: Applications in VLSI domain, *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp.69 –79, March, 1999.

[22] B. W. Kernighan and S. Lin, An efficient Heuristic procedure for partitioning graphs, *Bell System Tech. Journal*, vol. 49, pp.291-307, Feb. 1970.

[23] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by simulated annealing, Science, 220, pp. 671-680, 1983.

[24] B. Krishnamurthy, An improved min-cut algorithm for partitioning VLSI networks, *IEEE Trans. on Computers*, vol.33, no.5, pp.438-446, 1984.

[25] B. Mobasher, N. Jain, E. H. Han and Srivastava, Web mining: Pattern discovery from world wide web transactions, *Technical Report TR 96-50, Department of Computer Science, University of Minnesota, Minneapolis*, 1996.

[26] C. I. Park and Y. B. Park, An Efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no.11, pp.1686-1694, 1993.

[27] L. A. Sanchis, Multiple-way network partitioning, *IEEE Trans. on Computers*, vol.18, no.1, pp.62-81, 1989.

[28] D. G. Schweikert and B. W. Kernighan, A proper model for the partitioning of electrical circuits, *Proc. 9th Design Automation Workshop*, pp.57-62, 1972.

[29] S. Shekhar and R. Aggarwal, Clustering roadmaps for routing: hypergraph approach, *Technical Report, Department of Computer Science, University of Minnesota*, 1997.

[30] S. Shekhar and D. R. Liu, Partitioning similarity graphs: A framwork for declustering problems, *Information System Journal*, vol. 21, no. 4, pp. 475-496, 1996.

[31] H. Shin and C. Kim, A simple yet effective technique for partitioning, *IEEE Trans. Very Large Scale Integration System*, vol.1, no. 3, pp.380-386, Sept. 1993.

[32] W. Sun and C. Sechen, Efficient and effective placement for very large circuits, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 170-177, 1993.

[33] H. H. Yang and D. F. Wong, Efficient network flow based min-cut balanced partitioning, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol.15, no.12, pp. 1533-1539, 1996.

[34] C. W. Yeh, C. K. Cheng and T. Y. Lin, A general purpose, multi-way partitioning algorithm, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no.12, pp. 1480-1488, 1994.

[35] C. W. Yeh, C. K. Cheng and T. Y. Lin, A probabilistic Multi-Commodity Flow Solution to Circuit Clustering Problems, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 428-431, 1992.

[36] S. Wichlund and E. J. Aas, On multilevel circuit partitioning, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 505-511, 1998.

[37] N. Woo and J. Kim, An efficient method of partitioning circuits for multiple-FPGA implementation, *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pp. 202-207, 1993.

Appendix A

Algorithm PLM

Initialize bucket list pointers;

repeat /* for each pass */

for each of N_{out} phases **do**

Compute initial costs of cells, and initialize cells as unlocked;

Insert cells into bucket lists on the basis of their move gains;

repeat

Select a legal cell with maximum gain to move;

Delete that cell from bucket list and lock it;

Update the gains of affected cells;

until N_{in} times or there are no legal moves

if $N_{in} < n$ **then**

Free bucket list nodes for remaining unlocked cells;

Find maximum partial gain sum S_p and corresponding subsequence of cell moves;

if $S_p > 0$ **then**

Make the first p moves (corresponding to S_p) permanent;

Decrease the cutsize by S_p;

until $S_p \leq 0$

Version of PLM

name	number of iterations per pass N	N_{in}
PLM1	n	$n/2$
PLM2	nk	$n/2$
PLM3	nk^2	$n/2$

* $N = N_{in}\ N_{out}$, n is the number of cells in circuit, k is number of blocks.

Algorithm PFM

Initialize bucket list pointers;

repeat /* for each pass */

Compute initial costs of cells, and initialize cells as unlocked;

Insert cells into bucket lists on the basis of their move gains;

repeat

Select a legal cell with maximum gain and make the move;

Increment move count of the cell;

Update the gains and mobility values of all affected cells;

until N times or there are no legal moves

Find maximum partial gain sum S_p and corresponding subsequence of cell moves;

if $S_p > 0$ **then**

Make the first p moves (corresponding to S_p) permanent;

Decrease the cutsize by S_p;

Free all bucket list nodes;

until $S_p \leq 0$

Version of PFM

name	number of iterations per pass N	R
PFM1	n	2
PFM2	nk	8
PFM3	nk^2	128

* n is the number of cells in circuit, k is number of blocks.

Mobility = $\left\lfloor R(2 \cdot G_{\max} + 1) \cdot \frac{1}{1 + n_i^{0.5} . \exp^{-4.6 \cdot g_i(s,t)/G_{\max}}} \right\rfloor$, where n_i is the move count of cell c_i; $g_i(s, t)$ is the gain of cell c_i moving from block s to block t; G_{max} is product of maximum cell degree and maximum net weight, in case of net weight = 1, $G_{max} = \max_i [d(c_i)]$.

Appendix B

Benchmark file format

The benchmark avg_large and avg_small were obtained from Lars Hagen (lars@cadence.com). All the other benchmark files were obtained from the VLSI CAD Laboratory (ABK GROUP) at the Computer Science Department of University of California, Los Angeles. (abk@cs.ucla.edu)

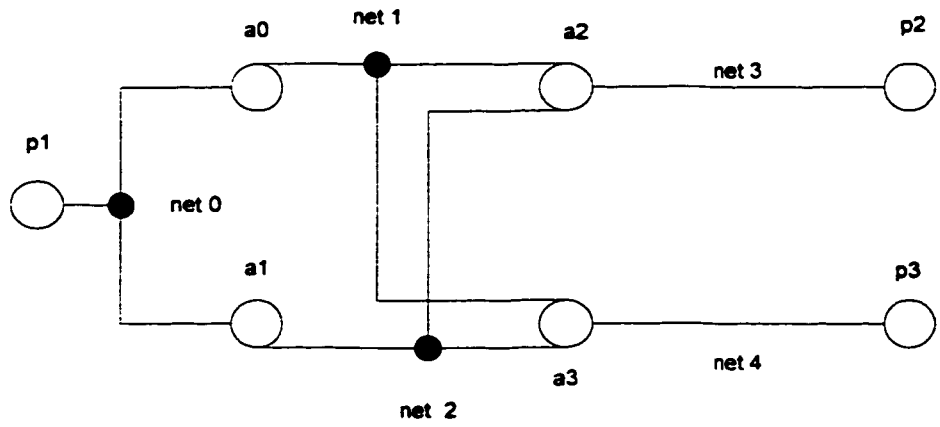
All the benchmark files are of the same format. Each file has a header with the five entries.

They are

ignored
number_of_Pins
number_of_Nets
number_of_Modules
pad_offset

The list of nets follows. Each net is simply a subset of modules that are either cells or pads. Cells are numbered from 0 to pad offset (inclusive). Pads are numbered from 1 to (#Modules - pad_offset - 1). Cells are prefaced by an 'a', pads by a 'p'. The beginning of each net is denoted by a 's'.

The following is an example circuit that has 4 cells, 3 pads, 5 nets and 13 pins.



The .net file for the above circuit is given by

0

13

5

7

3

p1 s 1

a0 1

a1 1

a0 s 1

a2 1

a3 1

a1 s 1

a2 1

a3 1

a2 s 1

p2 1

a3 s 1

p3 1

Implementation Issues

As stated in the thesis, in calculating the cutsize of a partitioning, we set the weight for all the nets to 1. In order to make those algorithms adaptable to the general form of cutsize, we need to modify the function of calculating cell cost. The range of bucket array is also needed to be expanded accordingly.

As stated in Chapter 4, we implemented all four algorithms used in this study (FMS, PLM, PFM and NGSP) using the C++ programming language.

We divide the basic information needed in a hypergraph partitioning algorithm into two categories: static and dynamic. The first is the information come from a benchmark file directly and would not be changed during the execution of an algorithm. The static information includes the cell weights, the net weights and the net lists (each stores the cells that are incident to the net). The information that is computed and/or changed during the execution of an algorithm is called dynamic information. It includes the cell gains, cell status (locked or not), block lists (each stores the cell numbers that are in the block) and net status.

We developed the following classes as the basic building block of all the algorithms in this study: class CellNode, class doubleLL, class aCell, class aNet, class aBlock and class bucket.

The class cellNode is used as a common node of storage for all kinds of linked (or doubly linked) list structure in our implementations.

```
class cellNode {
```

```
public:
  long int cellnum;  // cell number
  long info;         // stores information based on needs
  long index;     // used in the bucked list structure for storing gain index
  cellNode *next, *prev;

  cellNode(int i):cellnum(i),info(-1),index(-1),next(NULL), prev(NULL){}
  cellNode():cellnum(-1),info(-1),index(-1),prev(NULL),next(NULL){}
  cellNode(const cellNode& c):cellnum(c.cellnum),info(c.info),
        index(c.index),prev(c.prev), next(c.next){   }
  inline long GetCellNumber()const { return cellnum; }  // returns the cell number
  inline long GetInfo() const { return info; }   // returns the info field
  void Set(long cid, long Info) {cellnum=cid; info=Info;}   // resets the fields
private:
      cellNode& operator=(const cellNode& rhs);  // disable = operation
};
```

---

declaration of the class cellNode

The following is the declaration for the class doubleLL (doubly linked list). This class provides features that allow the user to use either the FIFO or the LILO approach.

```
class doubleLL {
public:
```

```
    cellNode headDummy,tailDummy;  /* dummy nodes */
    cellNode* head;   /* points to the first node of the list */
    cellNode* tail;    /* points to the last node of the list */
    unsigned long length; // not includes the head and tail dummies
    unsigned long currentindex; // the index of current node
    cellNode* currentPtr; // points to the current node

    doubleLL(); // default constuctor
    doubleLL(const doubleLL&); // copy constructor
    void reset(); // release the list
    void push_back(cellNode&); // insert a node to the back
    void push_front(cellNode&); // insert a node to the front
    void push_back(cellNode*); // insert a node to the back
    void push_front(cellNode*); // insert a node to the front
    void remove(cellNode*);      // remove node from the list
    cellNode* removehead();     // remove and return the first non-dummy node
    cellNode* removetail();     // remove and return the last non-dummy node
    cellNode* GetHead();        // returns a pointer that points to the head node
    cellNode* GetTail();      // returns a pointer that points to the tail node
    void resetcur();          // resets the current node to be the first node
    unsigned long size() const { return length; } // returns the size of the list
private:
    doubleLL& operator=(const doubleLL&); // disable = operation
```

};

---

declaration of the class doubleLL

As described in Chapter 4, the class Bucket represents a single bucket array of lists in our programs. We have two data members in this class (*maxindex* and *minindex*) that always store the current highest and lowest indexes at which the doubly linked lists are not empty. In the case of the bucket list being empty, both of the two data members have the same value of –1. Every time a node is inserted into one of the doubly linked list on the bucket array, or is removed from one of the doubly linked lists, these two data members are checked and updated if necessary. The purpose of the *maxindex* is to allow algorithms to find a cell movement that has the maximum gain value quickly. The data member *minindex* is there to reduce the time needed for a full bucket list search. Recent algorithms, like PFM and NGSP, create and use bucket array based on user defined gain values; that significantly increase the range of the gain value of a bucket list. This approach does help to improve the overall performance of these algorithms.

```
class Bucket {
public:
        long lb, hb; // lower , higher bound
        long size;   // bucket size
        long maxindex;  // the maximum index at which the list is not empty
        long minindex;  // the minimum index at which the list is not empty
        cellNode* cur;  // points to the current node of the bucket list
        long curindex;   // current index
        doubleLL* bk;  // dynamic list of doubly linked lists
```

```
public:
        Bucket():lb(0),hb(0),size(0),maxindex(-1),
                minindex(-1), cur(NULL),curindex(-1),bk(NULL){}
        ~Bucket() { if (bk) delete [] bk;}
        long GainToIndex(long gain) const;   // convert gain into array index
        long IndexToGain(long ind) const;   // convert array index into gain
        long MaxGain() const;  // return the non-empty max. gain; -1: empty buket list
        void AddFrontAt(long gain, cellNode& nnode); // insert a cellNode at gain
        void AddEndAt(long gain, cellNode& nnode);  // // insert a cellNode at gain
        cellNode* RemoveFrontAt(long gain);
        cellNode* RemoveEndAt(long gain);
        void RemoveNodeAt(long gain, cellNode* nnode);

        void Updatemax(long);  // recompute the maximum nonempty index
        void Updatemin(long);  // recompute the minimum nonempty index
        void ResetAt(long g); // clear the doubly linked list at gain g
        long int SizeAtGain(long gain) {return bk[gain-lb].length; }
        void clear();  // empty the bucket list
        void resize(long,long); // reset the siz eof the bucket list
        long ResetCur(); // reset the current pointer to the first of node with highest gain
        long CurNext();  // move the current pointer to the next node
        bool IsEmpty(); // whether the bucket is empty or not
        doubleLL& operator[](long int i);
};
```

declaration of the class Bucket

An object of class *aCell* is used to represent a cell in a circuit. A unique cell number identifies a cell in a circuit. The vector *netOnCell* stores the lists of nets that are incident to the cell. The pointer *bNode* points to a dynamic list of *cellNodes*. The size of this dynamic list is set to be the number of blocks for each execution of the algorithm. One

cellNode object corresponds to a possible moving direction (block) and it is inserted into the corresponding bucket list based on the gain value computed during the execution of the algorithm.

```
class aCell {
public:
        static long c_total; // total number of cells
        cellNode cNode;          // link to the block where the cell is located
        bool locked; // whether the cell is locked or not
        cellNode* bNode;  // list of nodes to be inserted into the bucket lists
        vector<long int> netOnCell;      // nets on the cell
        aCell():cNode(c_total++),locked(false),bNode(NULL){}
        ~aCell();
        void output(ostream& os=cout);  // outputs the cell information
        long GetBlock(){return cNode.info;} // returns the block numebr
        static long GetTotal() { return c_total;} // returns the total number of cells
};
```

declaration of the class aCell

An object of class *aNet* represents a net in a circuit. Same as a cell, each net in a circuit has a unique net number.

```
class aNet {
public:
        static long n_total;  // total number of nets in a circuit
        static long cut_size;  // the number of blocks in which the net has cells
        const int MAX_PART=25; // maximun number of partitions
        long nid;  // net number
```

```
        int nWeight;  // weight of the net
        bool isCut;  // whether the net is in cutset or not
        vector<long int> cellOnNet;  // list of cells on net
        set<long, less<long> > blocksOnNet;  // set of blocks in which the net has cells
        long dist[MAX_PART]; // the number of cells in each partition(distribution)

        aNet();
        static void IncreaseCSize() {cut_size++;}
        inline void DecreaseCSize();
        void ACellToBlock(int); //  one of its cell moved to a block
        bool ACellLeaveBlock(int);  // one of its cell left from a block
        long size() { return cellOnNet.size(); } // return the size of the net
        void output(ostream& os=cout);
};
```

declaration of the class aNet

The class aBlock is to represent a block in the process of partitioning. The doubly linked list links the *cellNodes* object from the cell list.

```
class aBlock {
public:
        static int b_total;
        long bid; // block id
        doubleLL cellsIn;
        aBlock():bid(b_total++){}
};
```

declaration of the class aBlock