Fast Placement Algorithms for Grids in Two and Three Dimensions

Stefan Thomas Henning Obenaus
School of Computer Science
McGill University, Montreal

September 2000

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Abstract

Topic of this dissertation is the fast placement of hypergraph nodes into multi-dimensional grids. Hypergraphs can be used to model electronic circuits and network topologies, and grids can model the underlying architectures of electronic integrated circuits and optical systems. Thus, placement of hypergraphs into grids is an important problem as it may serve, for example, to reduce interconnection wire length in integrated circuits, or to minimize communication delays incurred in optical networks by routing around unnecessary bends. Fast placement techniques that complete in near-linear run time are desirable as circuit and network topologies are expected to continue to grow in size exponentially. The term "fast placement" thus refers to (a) the run time of the placement algorithm, and (b) the lower transmission delays in the resulting placement.

Original research contributions are provided in chapters 1, 4, and 5.

In chapter 1, the extension of relevant combinatorial grid placement problems to hypergraphs and multi-dimensional grids constitute a modest original contribution.

Entirely original work is presented in chapter 4, where we introduce a novel force-directed iterative placement algorithm for two- and three-dimensional placements. This algorithm is designed to produce good minimum-wire-length placements for large circuits, for which no comparison results exist. The three-dimensional implementation of our placement algorithm pioneers the 3-D placement field in which previously no efficient algorithms had been published. In order to avoid operating in a vacuum, we were forced to create a comparison algorithm based on an accepted standard placement technique. With this reference placer, we generated the first published 3-D placement results, and 2-D placement results for large benchmark circuits for which no published comparison results exist. Our placement algorithm out-performs the reference placer substantially in both run time and wire length. Further, we use our algorithm to present some experimental evidence of the estimated wire-length savings when utilizing the third dimension.

Our final original contribution is a method presented in chapter 5 for efficiently placing a modern network topology, the star graph, into multi-dimensional grids such that all star graph neighbours are joined by a common grid line. The basic placement technique, originally published in [Obe95], is made efficient by compacting and contracting the bendless embedding in an effective manner.

Résumé

Cette thèse porte sur le positionnement rapide des nœuds d'hypergraphes sur des grilles multidimensionnelles. Les hypergraphes peuvent être utilisés afin de modéliser des circuits électroniques ainsi que des topologies de réseaux, tandis que les grilles peuvent servir à modéliser l'architecture sous-jacente de circuits électroniques intégrés et de systèmes optiques.

Le positionnement d'hypergraphes sur des grilles est donc un problème important puisqu'il peut servir, par exemple, à réduire la longueur des fils d'interconnexion de circuits intégrés, ainsi qu'à minimiser les délais de communication engendrés par des détours inutiles dans les réseaux optiques. Le développement de techniques de positionnement rapide dont le temps d'exécution est quasi linéaire semble souhaitable puisqu'il est très probable que la dimension topologique des circuits et des réseaux continue de croître de façon exponentielle. Le positionnement rapide fait donc référence à deux choses, soit (a) le temps d'exécution de l'algorithme de positionnement, ainsi que (b) la réduction des délais de transmission découlant de ce positionnement.

Les contributions scientifiques originales se trouvent dans les chapitres 1, 4, et 5.

L'extension de problèmes combinatoires de positionnement de grilles en problèmes d'hypergraphes et de grilles multidimensionnelles, présentée dans le chapitre 1, constitue une contribution originale modeste.

Le travail présenté dans le chapitre 4 représente une contribution véritablement originale. Nous y décrivons un nouvel algorithme de placement itératif basé sur le comportement des forces pouvant être utilisé pour des positionnements bidimensionnels ainsi que tridimensionnels. Cet algorithme permet de trouver de bons positionnements utilisant une longueur minimale de fil pour les grands circuits pour lesquels il n'existe pas de résultats pouvant servir de référence. L'implémentation tridimensionnelle de notre algorithme de positionnement est une première dans le domaine du positionnement tridimensionnel puisque jamais un algorithme efficace n'a été publié. Afin d'avoir un point de comparaison, nous avons dû créer un algorithme de référence basé sur une technique de positionnement standard approuvée. Cette référence nous a permis de produire les premiers résultats de positionnement tridimensionnel publiés. Elle nous a aussi permis d'obtenir des résultats de positionnement bidimensionnel pour les grands circuits de référence pour lesquels aucuns résultats pouvant servir de référence n'ont été publiés. Notre algorithme de positionnement est beaucoup plus performant que l'algorithme de référence au niveau du temps d'exécution et de la minimisation de la longueur de fil. De plus, nous utilisons notre

algorithme afin de démontrer expérimentalement l'économie de longueur de fil estimée qui découle de l'utilisation de la troisième dimension.

Notre contribution originale finale se trouve dans le chapitre 5. Nous y présentons une méthode permettant de placer de façon efficace une topologie de réseau moderne, le graphe étoilé, sur une grille multidimensionnelle de telle sorte que tous les graphes étoilés voisins soient raccordés par une ligne commune. La technique de placement de base que nous avons publiée dans [Obe95], est améliorée grâce à la compaction et à la contraction efficaces des insertions de graphes étoilés sans inflexions.

Acknowledgements

Without the support and guidance of my supervisor Professor Ted Szymanski, this thesis would have been impossible to complete. Ted always encouraged me to pursue an interesting idea, and help me find an application for it. I am also grateful for his financial support and for not making me stay in youth hostels on trips to conferences.

My co-supervisor Professor Luc Devroye also deserves my thanks for helping me take a different look at problems, and making me tidy-up my formalism and math.

Professor Sue Whitesides deserves special acknowledgement for making me lay a foundation before building a house.

For translating the abstract into French, I am grateful to Nancy St-Onge.

Further, I want to thank the Government of Quebec for providing me with financial support through their FCAR scholarship program.

The most heartfelt gratitude, I owe to my wife and love Minnie. She stuck with me and did not move to New York City to become rich and famous.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce the concept of *placement* and examine underlying and related problems. As notational aids, we will make use of hypergraphs and grids. This notation and the presented problems will be illustrated in section 1.4 using a running example. The running example consists of a real electronic circuit and a fictitious, yet realistic, model of a field-programmable gate array (FPGA), that can house the circuit. The relevant placement and routing steps and combinatorial problems are illustrated using this example.

1.1 What is Placement?

Placement is the art of assigning coordinates to nodes of a hypergraph.

Hypergraphs, cf. definition 2 on page 4, can be, for example, computer network nodes interconnected by a network topology [Tan96], or electronic circuit cells forming a VLSI circuit [Wol94], or graphs in graph theory [Ber85]. The coordinates to be assigned might represent floors and rooms in an office building that is to be networked, or they might be positions on a silicon die which is to house an integrated circuit.

Placement thus forms one part of an embedding process that involves placing interconnected nodes and routing the interconnections between the placed nodes. In VLSI chip design this embedding process is generally split into two processes known as *place* and *route*, and in graph theory as *graph drawing*.

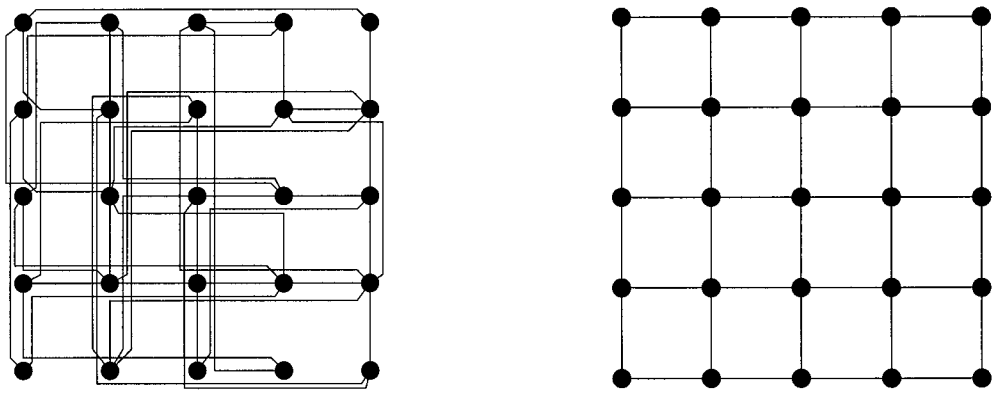


Figure 1.1: Bad (left) and good placement of a 5×5 mesh.

Before we formally define a placement in definition 4 on page 11, we first state qualitatively what a placement is used for. Thus, the mission statement for placement can be defined as follows:

Definition 1 *Purpose of Placement*
The purpose of determining a node placement is to enable the creation of a good routing of interconnections.

What constitutes a "good" routing is in the eye of the beholder and we will look at several popular interpretations in detail. Within this dissertation we use the term *embedding* to refer to the process that maps hypergraphs into another structure such as a grid. The part of the *embedding* that assigns coordinates to the interconnected nodes is referred to as the *placement.* The part that routes the interconnections between the placed nodes is called the *routing.* Thus, placement + routing = embedding. Formal definitions of hypergraphs and grids are provided in section 1.4.

1.2 Impact of Placement

Choosing one placement over another can have a profound impact on the desired parameters of an embedding. Poor placement can lead to excessively long or twisted

Figure 1.2: Bad (left) and good routing of a square with a good placement.

interconnections. The example in figure 1.1 illustrates the effect of a good versus a random placement.

For example, when designing a chip, choosing a bad placement can lead to long interconnections. Since the sizes of a circuit's components are usually fixed, an increase in area required for longer wiring leads to an increased chip area [SM91], and increased chip size translates to higher manufacturing costs.

In different research areas different placement and routing parameters are important. While the total wire length may be important in chip design, it may be of no consequence in other areas. In orthogonal graph drawing, for example, the number of bends in the drawn edges may be a factor that is to be minimized [DBETT99].

1.3 Distinction Between Place and Route

By definition 1, the purpose of placement is to provide for a good routing. Thus, a good routing implies a good placement. However, a good placement does not guarantee that every routing will be good, or that a good routing is easy to find.

For example, suppose our measure for a good routing is the total length of the routed edges along horizontal and vertical grid lines, then figure 1.2 shows how a good placement can have a bad routing.

In the next section it is outlined how placement and routing are often both NP-complete, cf. section 1.4, and hence difficult problems. As both problems, routing and placement, are difficult problems by themselves, it is even more difficult to solve them together, and hence they are often treated as two sequential problems: placement followed by routing. However, we note that approaches that solve both aspects together exist [SKK+98]. In the more common split place-and-route ap-

proach, a placement is determined first, using estimates on the projected routing, and then, while all nodes are fixed in their positions, an actual routing is performed, e.g. [Ban94, SY95, Wol94]. This dissertation focuses on the placement phase.

1.4 Underlying Combinatorial Optimization Problems

In this section, some of the fundamental problems relevant to embeddings in general, and placements in particular are outlined. Some of these problems have previously been discussed in the context of optical interconnects and networks in [SO98].

As the title of this thesis indicates, we will only consider placements and embeddings of hypergraphs into *grids*. Within this dissertation, using grids means placement coordinates are integers. First, we define the hypergraph and the grid. Then, we discuss individual embedding problems and their impact on placement strategies.

Hypergraphs have the main property that a number of interconnections, called edges, each connect a number of nodes:

Definition 2 *Hypergraph*
A hypergraph $H(V, E)$ *is a set of* nodes, *sometimes called* vertices, V *and a set of* edges E *such that* $\forall\, e \in E \;\; e \subseteq V$.

The interconnectivity of electronic circuits can be modelled by hypergraphs. In the context of electronic circuits, the terms *net* and *cell*, are often used in place of hypergraph edge and node. Additionally, the term *pin* in an electronic circuit is used to refer to a connection between a node to an edge. As a concrete example we will look at an 8-to-4 *Daisy Chain Concentrator* [SS98]. An n-to-m Daisy Chain Concentrator is an efficient way of concentrating the signals from n synchronized inputs to m outputs. An 8-to-4 Daisy Chain Concentrator takes the signals from 8 inputs and forwards them to 4 outputs. Signals are only blocked if more than 4 inputs are active at the same time. Figure 1.3 shows the schematic of the control plane of an 8-to-4 Daisy Chain Concentrator. This control plane controls the flow of data in the data plane, which is not shown.

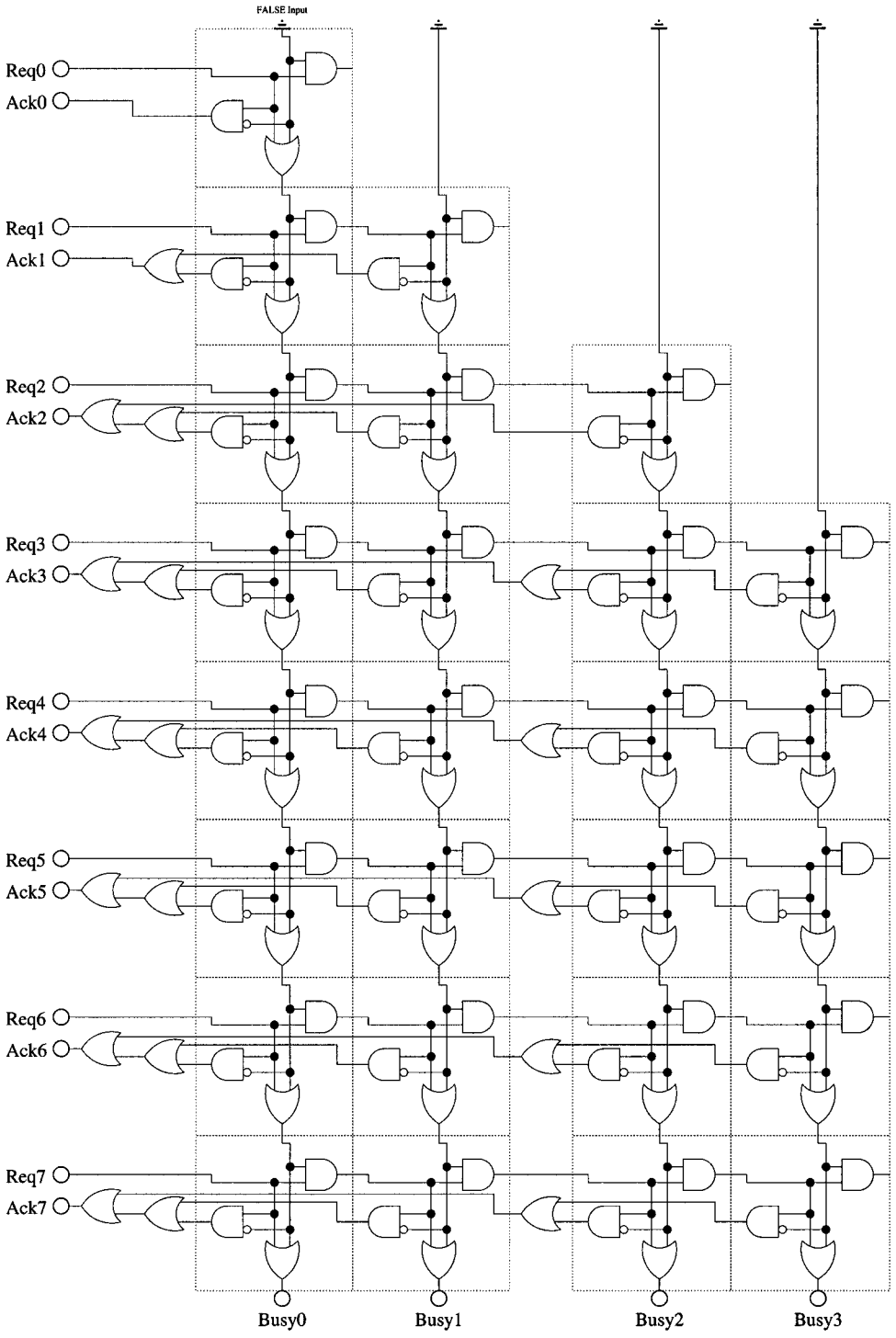


Figure 1.3: 8-to-4 Daisy Chain Concentrator schematic [SS98]: 8 inputs at the left are forwarded to 4 outputs at the bottom.

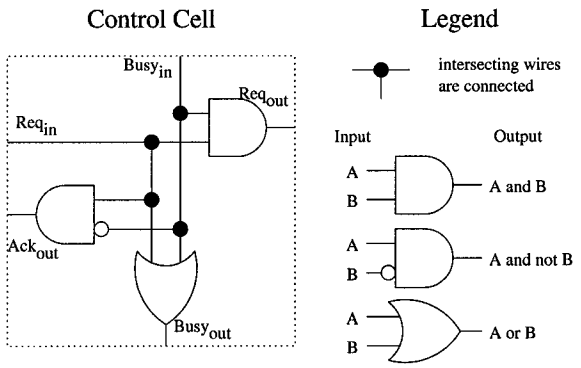


Figure 1.4: Control cell schematic [SS98]: This is the building block for a Daisy Chain Concentrator.

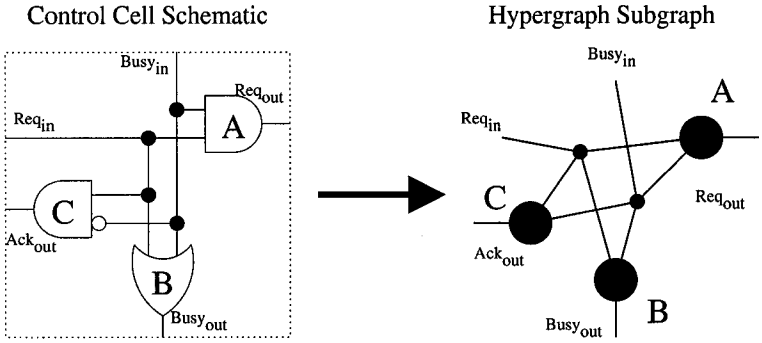


Figure 1.5: Conversion of a control cell schematic [SS98] to hypergraph representation: Only the structure of interconnections is preserved.

The control plane of the 8-to-4 Daisy Chain Concentrator is made up from eight rows and four columns of Daisy Chain Concentrator control cells, cf. figure 1.4. Each control cell forwards an input request to the next control cell in the row only if the output in the cell's column is already busy.

In order to focus on the various combinatorial optimization problems associated with the embedding problem, we transform the electronic schematic into hypergraph representation. In figure 1.5 we illustrate this transformation for a Daisy Chain Concentrator control cell. The structure of interconnections is preserved while the gate types are ignored. Individual hypergraph nodes are labelled such that this transfor-

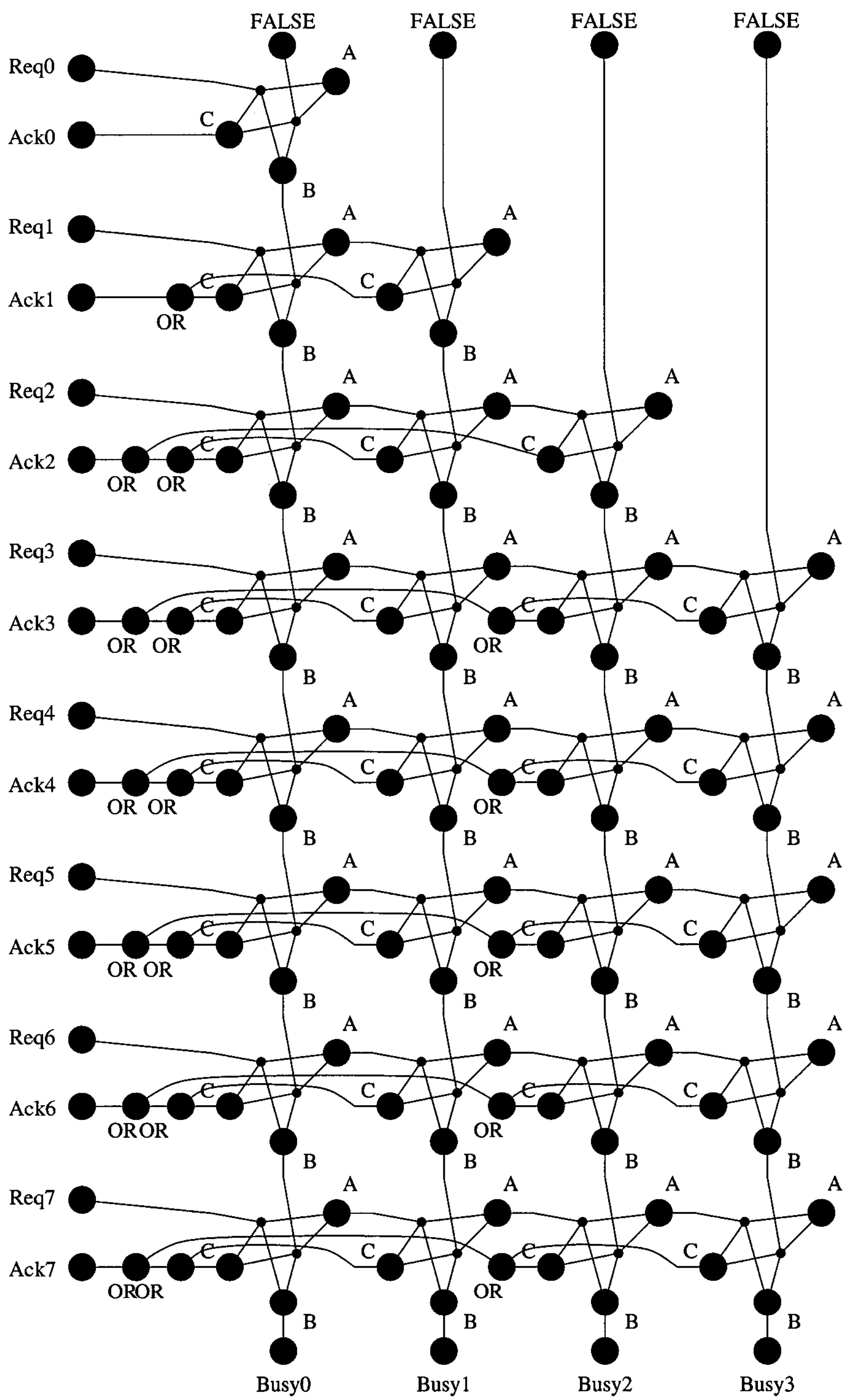


Figure 1.6: 8-to-4 Daisy Chain Concentrator in hypergraph representation.

mation process can be reversed. Big black dots represent nodes of the hypergraph, and lines represent edges that connect the nodes in which the lines terminate. The complete 8-to-4 Daisy Chain Concentrator in hypergraph representation is illustrated in figure 1.6.

Note, a *graph* is a hypergraph in which all edges have exactly two nodes. A thorough treatment of graphs and hypergraphs can be found in [Ber85] and [Ber89]. Berge [Ber85] defines the multiplicity of an edge in a graph as the number of edges connecting the same end points. Formally, the multiplicity of an edge is simply a positive integer associated with this edge. We define a grid as follows.

Definition 3 *Grid*

An $N_1 \times \cdots \times N_d$ grid $G(V, E)$ *with k parallel channels is a graph whose nodes are the points in the d-dimensional integer space* $\{0, \ldots, N_1 - 1\} \times \cdots \times \{0, \ldots, N_d - 1\}$, *and whose edges connect nodes that are separated by a euclidean distance of 1. The multiplicity* $m(e)$, *i.e., the number of parallel edges, of each edge e is k.*

$$
\begin{aligned}
V &= \{v : v \in \mathbb{Z}^d \text{ and } \forall i \in \{1, \ldots, d\} \quad 0 \leq v_i < N_i\}, \text{ and} \\
E &= \{\{u, v\} : u, v \in V \text{ and } |u - v| = 1\}, \text{ and} \\
m &: E \rightarrow \{k\}.
\end{aligned}
$$

This definition defines the number of parallel channels to be the multiplicity of the grid edges. If we label parallel grid edges with the numbers from 1 to k, then we say that the grid edge labelled i belongs to channel i, and all grid edges with label i together form channel i. This definition of a channel conforms with the terminology used in communications where channels are often allocated as frequency bands or wavelengths, for example. In the context of optical systems, e.g. [SH98], a channel is often understood to be the collection of like-labelled edges that connect nodes along one grid line, i.e., the nodes that differ only in one coordinate. In integrated circuit design, a channel is often understood as the collection of all edges along a grid line[BFRV92]. Throughout this dissertation we will use the communications understanding of a channel as described above.

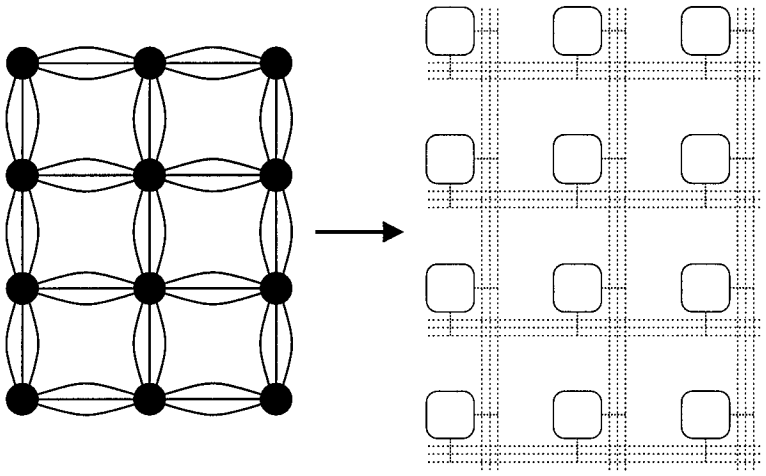


Figure 1.7: A 3×4 grid with 3 parallel channels drawn using a computer science convention (left) and an engineering convention.

In the computer science community, a graph such as a grid is normally drawn as dots interconnected by lines representing the edges as, for instance, in figure 1.7 (left). The electrical engineering community often prefers a representation in which the grid nodes are placed next to the vertical and horizontal edges as shown on the right of figure 1.7. The latter convention facilitates illustration of embeddings that include a placement and a routing. Figure 1.8 shows a two-dimensional 10 by 18 grid with three parallel channels using this electrical engineering convention. The grid nodes are labelled by their coordinates. This grid may be thought of as a programmable chip, or field-programmable gate array, cf. FPGA, section 2.4. Every grid node can be programmed to hold a number of logic gates of an electronic circuit. The dotted lines representing the grid edges can be programmed to form wires interconnecting the logic gates.

Now we can formulate all embedding and placement problems in terms of hypergraphs and grids. For this, we define an embedding as the conjunction of a placement and a routing. Among the many conceivable parameters describing an embedding are two useful parameters defined by Leighton [Lei92]: The *load* and the *congestion*. Here, the hypergraph is the guest graph, and the grid is the host graph. The load of an embedding is the maximum number of nodes in a guest graph that

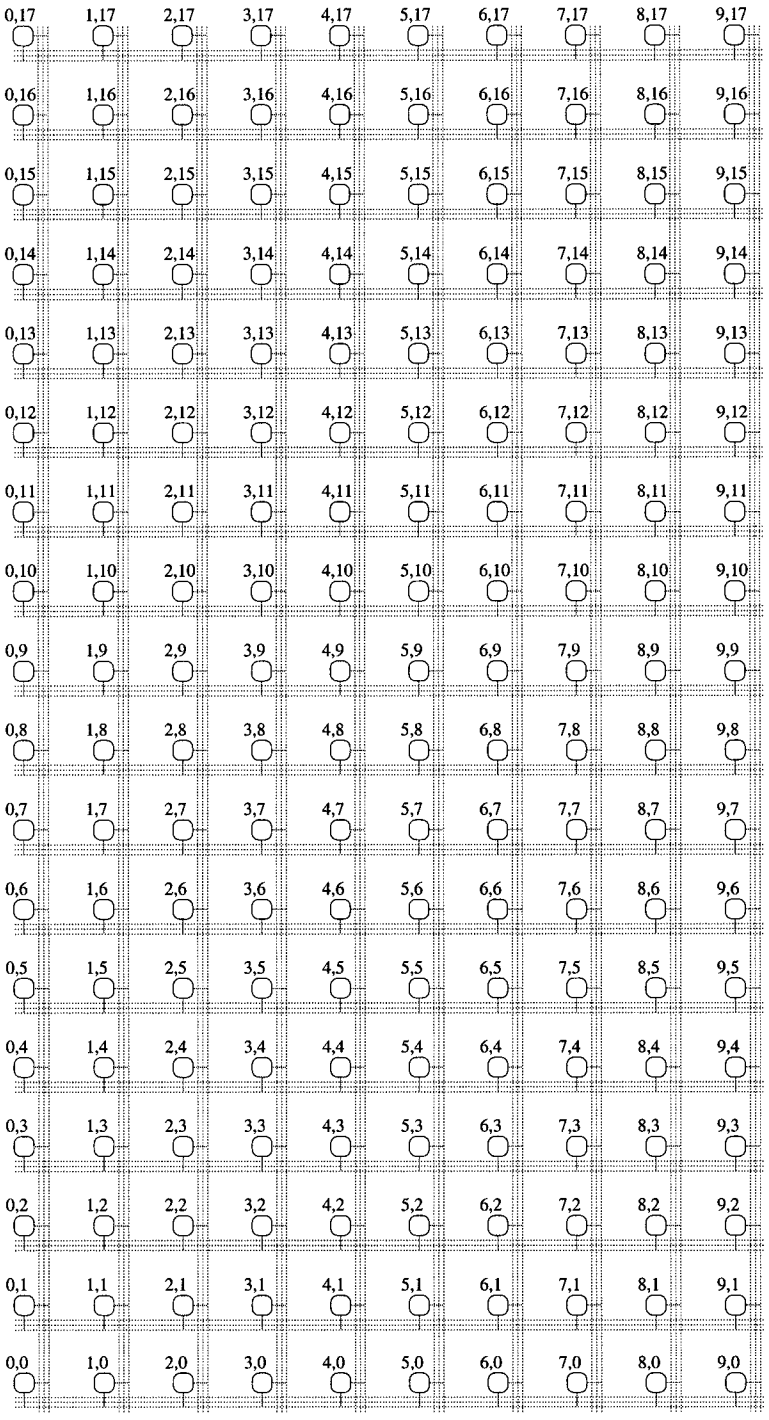


Figure 1.8: 10 × 18 grid with three parallel channels: The grid nodes are labelled by their coordinates, and the grid edges are shown in triplicate representing the multiplicity of each edge.

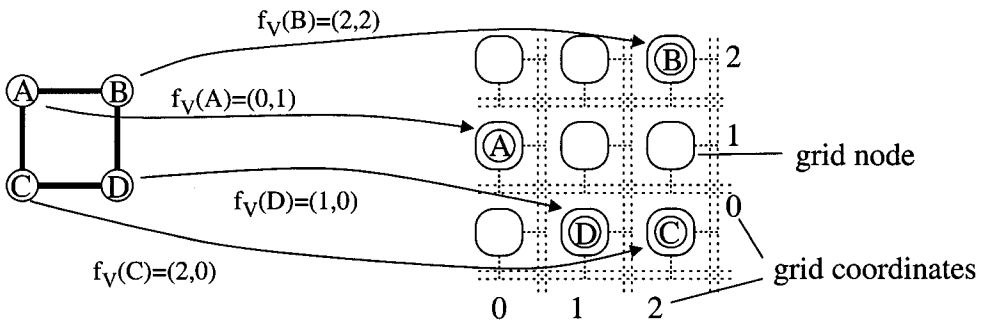


Figure 1.9: Placement of the nodes of a square in a 3×3 grid with 2 parallel channels.

are placed into the same node of a host graph. The congestion of an embedding is the maximum number of guest edges routed through a host edge. When we refer to the load of a particular host node v, we define the load as the number of guest nodes placed into node v, and similarly the congestion of an edge e is the number of guest edges routed through edge e. We refer to the load and congestion of an embedding (f_V, f_E) as load(f_V) and congestion(f_E), and to the load of a grid node v and congestion of a grid edge e as load(v) and congestion(e).

We define a hypergraph placement formally:

Definition 4 *Placement*

A placement f_V in n dimensions maps the nodes of a hypergraph $H(V,E)$ to nodes in a grid $G(V_G, E_G)$ with k parallel channels with a load of 1:

$$f_V : V \rightarrow V_G \text{ and,}$$
$$\max_{v_G \in V_G} |\{v : f(v) = v_G\}| = 1.$$

The left-hand side of the last equation formally defines load of a placement (= load of an embedding) as load$(f_V) \equiv \max_{v_G \in V_G} |\{v : f(v) = v_G\}|$.

In an effort to illustrate this definition of a placement, figure 1.9 shows an example of a placement of a square into two dimensions. In figure 1.10 we show a placement of the 8-to-4 Daisy Chain Concentrator into the 10×18 grid with 3 parallel channels. The nodes from the hypergraph of figure 1.6 are placed "inside" the grid nodes of the grid from figure 1.8.

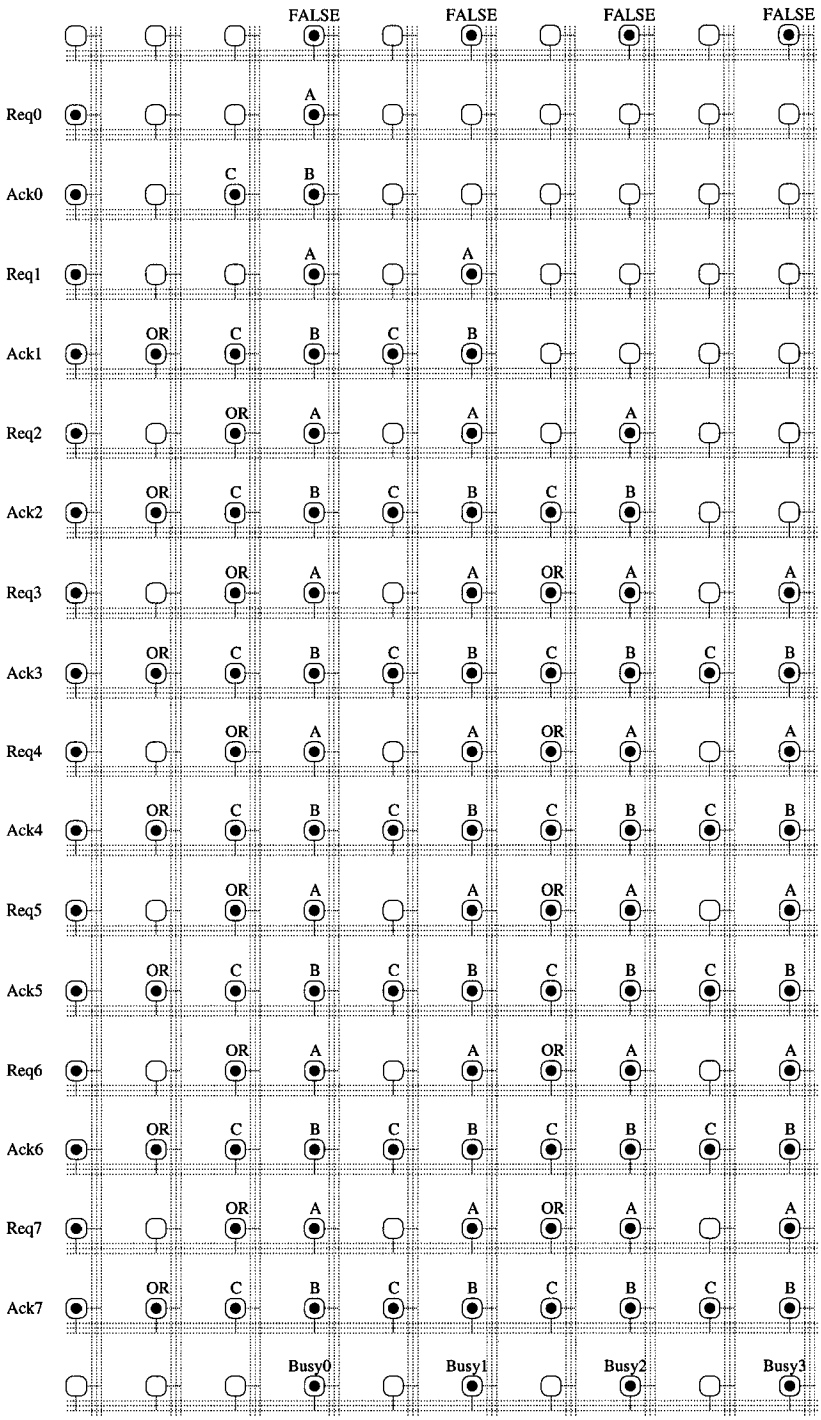


Figure 1.10: Placement of the 8-to-4 Daisy Chain Concentrator into the 10×18 grid with 3 parallel channels. The nodes from the hypergraph of figure 1.6 are placed "inside" the grid nodes of the grid from figure 1.8.

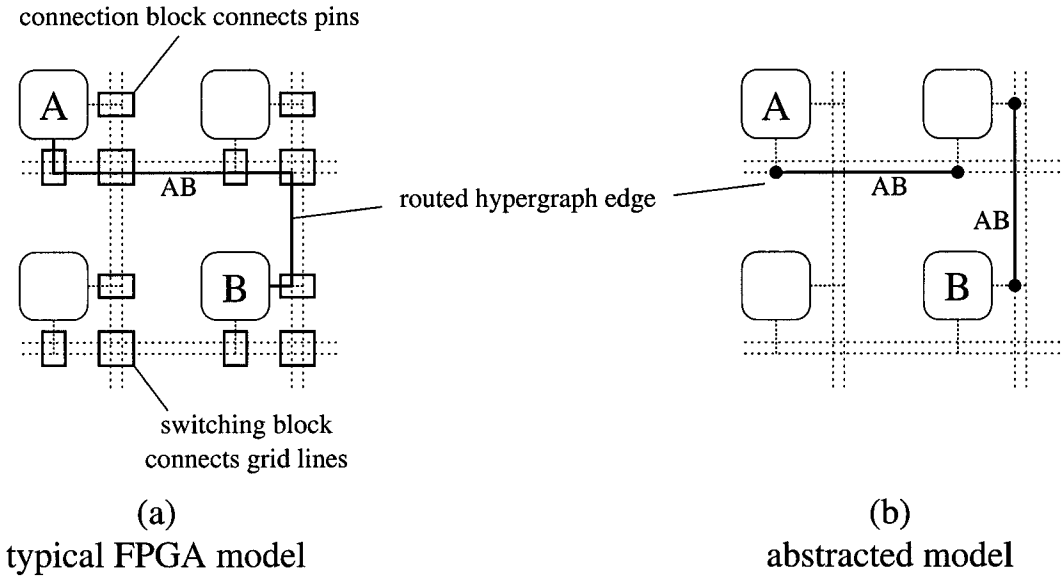


Figure 1.11: Routing diagram using a typical FPGA routing model [BFRV92] (a) and our abstracted routing model (b).

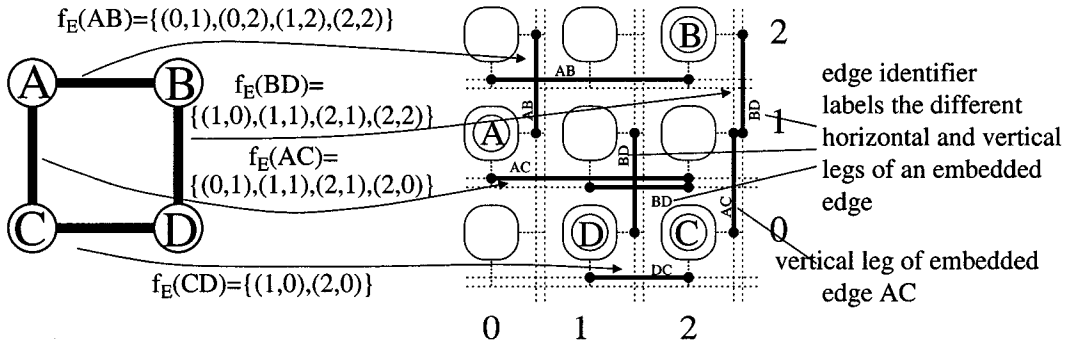


Figure 1.12: Routing of the edges of a square in a 3×3 grid with 2 parallel channels.

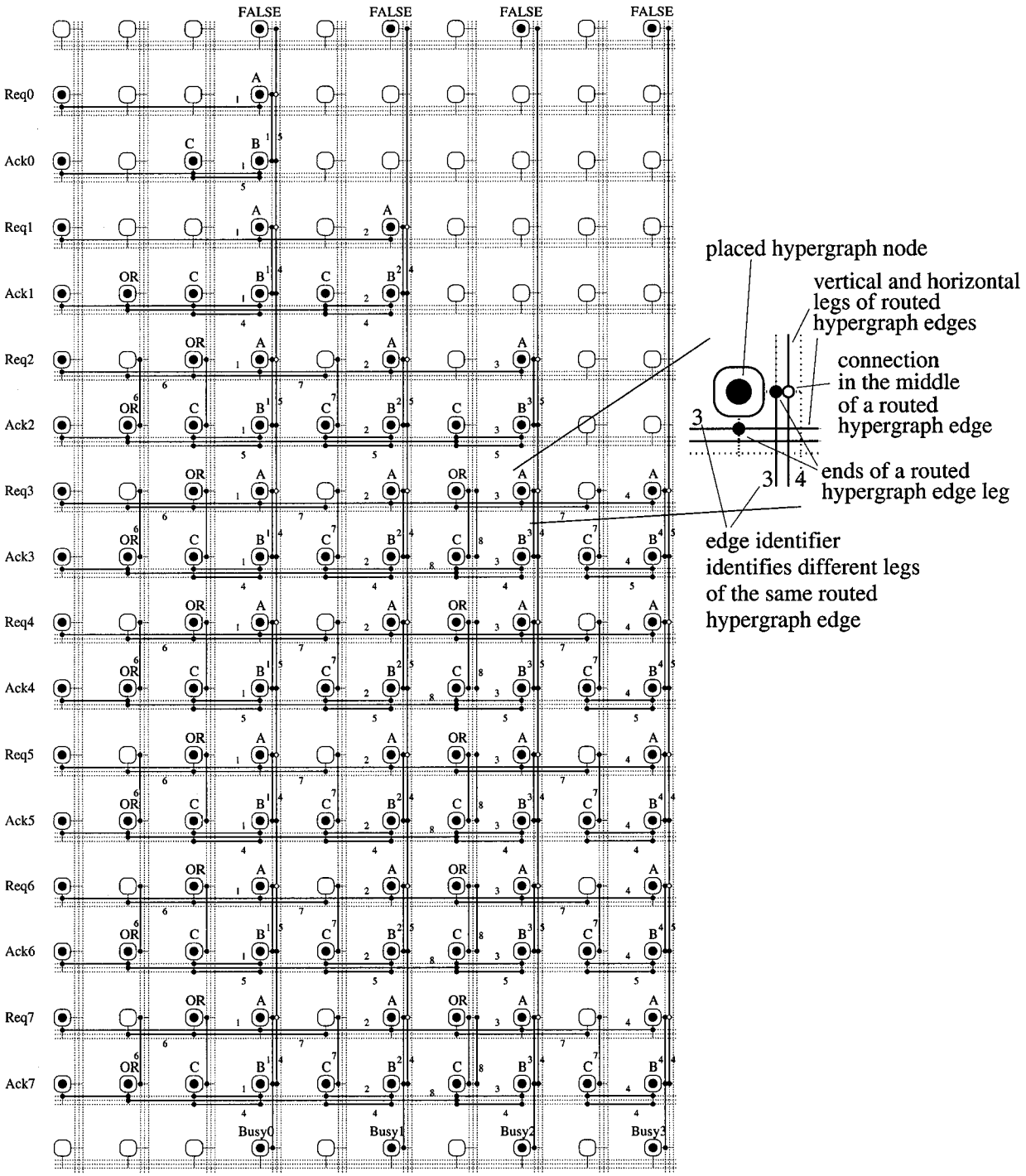


Figure 1.13: Routing of the 8-to-4 Daisy Chain Concentrator in the 10×18 grid with 3 parallel channels: The edges from the hypergraph of figure 1.6 are assigned to grid edges from figure 1.8 such that the original hypergraph connectivity is maintained.

Given that a the placement assigns hypergraph nodes to grid nodes, a routing assigns hypergraph edges to sequences of grid edges that connect the placed hypergraph nodes. Formally, we say that in a graph $G(V, E)$, two nodes u, v are *connected* if either $\{u, v\} \in E$, or a node connected to u is connected to v. Later in section 1.4.2 and chapter 2, we will see that there are many possible constraints put on finding a routing depending on which technology underlies it, such as timing constraints or segmentability of channels. However, the common denominator of all routing problems of this kind requires that a routing routes the hypergraph edges through the grid such that the congestion does not exceed the number of parallel channels in the grid.

As a notational aid, we say that if f is a function, and S is a subset of the domain of f, then $f(S)$ is the set of values mapped by applying f to the elements in S.

Definition 5 *Routing*

Given a hypergraph $H(V, E)$, a grid $G(V_G, E_G)$ with k parallel channels, and a placement f_V, then a routing f_E maps every edge $e \in E$ to a collection of grid edges that connects the members of e such that the congestion does not exceed k:

$$\forall e \in E \quad f_E(e) \subseteq E_G \text{ and,}$$
$$\forall e \in E \; \forall u, v \in e \quad f_V(u) \text{ and } f_V(v) \text{ are connected in } (V_G, f_E(e)) \text{ and,}$$
$$\max_{e_G \in E_G} |\{e : e_G \in f_E(e)\}| \leq k.$$

The left-hand side of the last inequality is a formal definition of the congestion of a routing (= congestion of an embedding), i.e., congestion$(f_E) \equiv \max_{e_G \in E_G} |\{e : e_G \in f_E(e)\}|$.

For illustration purposes, we use an abstracted diagram of a typical FPGA routing model [BFRV92]. Abstracted and original diagrams for a routed edge are compared in figure 1.11. The abstracted diagram exhibits no routing restrictions beyond those of definition 5 unlike a typical FPGA model, which is more restrictive. Figure 1.12 shows such an abstracted diagram of a routing of the edges of the square previously placed in figure 1.9. The complete routing of the 8-to-4 Daisy Chain Concentrator in the 10×18 grid with 3 parallel channels is illustrated in figure 1.13.

The edges from the hypergraph of figure 1.6 are assigned to grid edges from figure 1.8 such that the original hypergraph connectivity is maintained.

Optimization problems such as the ones related to embeddings are often phrased as decision problems that ask whether problems can be solved given a fixed amount of resources, e.g. channels, wire, etc., rather than asking for the minimum amount of resources required. Many of these problems are considered particularly difficult and classified as NP-complete.

1.4.1 NP-Completeness

A decision problem that is NP-complete is a difficult problem, whose solution can be easily verified if it can be found. In the context of NP-completeness, the adjectives "easy" and "efficient" mean "solvable in polynomial time, i.e., time proportional to a polynomial of the input size." Adjectives "hard" and "difficult" suggest that superpolynomial time is required.

NP stands for "non-deterministic, polynomial time." Intuitively, this means a problem in NP can be solved in polynomial time if you happen to guess the right answer. If a problem in NP is also NP-complete, it means that every other NP-complete problem can be transformed into it via a polynomial-time transformation. Thus if an efficient algorithm can be found for one NP-complete problem, an efficient algorithm is found for every NP-complete problem. Unfortunately, nobody has ever discovered a polynomial-time algorithm for any NP-complete problem. Hence, we require superpolynomial time to solve an NP-complete problem. Problems that are not necessarily NP-complete but proven to be at least as difficult to solve are called NP-hard. An in-depth treatment of NP-completeness and a comprehensive collection of known NP-complete problems can be found in [GJ79].

Knowing that a problem is NP-complete generally means that it is futile to look for an exact solution. Since many NP-complete problems are important, science has to give way to art, and heuristics have to be employed in order to find approximate solutions.

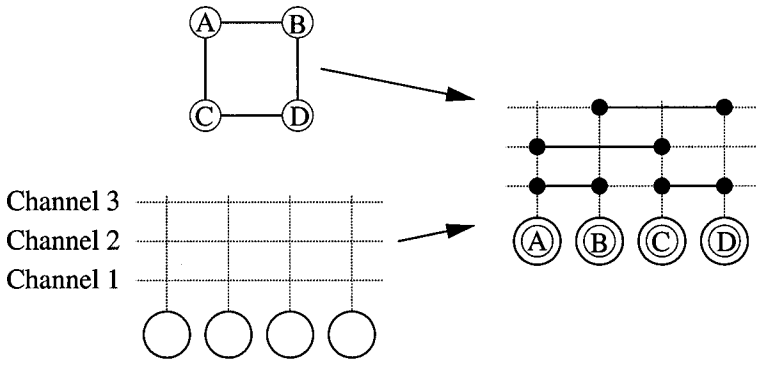


Figure 1.14: Channel assignment of a square into a 1-D grid.

1.4.2 Minimize Channels: Edge to Channel Assignment

Our definition of a routing, definition 5, purposely omitted the assignment of routed edges to specific *channels*. A channel is a logical entity which can correspond, for example, to a time slot in time division multiplexing (TDM) communication, a wavelength in wave division multiplexing (WDM), a frequency band in broadband communication, or space allotted for a wire track on a chip. Figure 1.14 shows a one-dimensional grid with three parallel channels hosting a square.

We note that channel 1 carries two edges, one between nodes A and B, and one between nodes C and D. This means the channel is segmentable, i.e., grid edges belonging to the same channel may host different hypergraph edges. In an environment where channels are not segmentable, every edge requires an entire channel for itself and the channel assignment problem becomes trivial. We will examine only segmentable channels.

Given a placement and a routing as defined above, we wish to minimize the number of channels used. Preferably, we only want to use a number of channels equal to the congestion, i.e., the maximum number of edges routed along a grid edge. The example in figure 1.14 uses three parallel channels even though at most two edges are routed in parallel through any given grid edge. We define this problem:

Problem 1 *Channel Minimization*

Instance: *A hypergraph $H(V, E)$, a grid with k parallel channels, a placement f_V, and a routing f_E.*

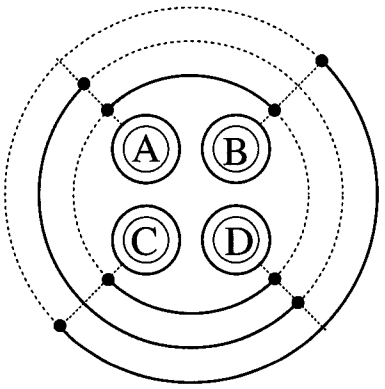


Figure 1.15: Circular one-dimensional grid with 3 parallel channels.

Question: *Does there exist a channel assignment* $f_C : E \rightarrow \{1, \ldots, k\}$ *such that no two edges utilize the same grid edge and channel, i.e.,*

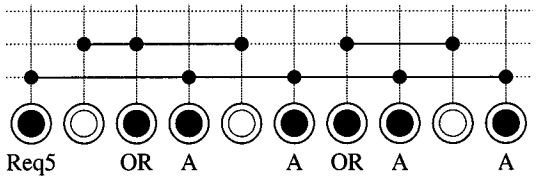
$$\forall e_1, e_2 \in E \;\; e_1 \neq e_2 \;\; and \; f_E(e_1) \cap f_E(e_2) \neq \emptyset \;\; \Rightarrow \;\; f_C(e_1) \neq f_C(e_2) \;\; ?$$

The simplest form of this problem occurs in an embedding of a hypergraph into a one-dimensional grid. This form of the problem can be solved in time proportional to the number of edges and nodes in the hypergraph by the *Left-Edge Algorithm* [HS71, KP87]. Edges are simply routed starting with the first channel with the leftmost edge. Then channel after channel are filled from left to right as tight as possible. The resulting embedding requires exactly as many channels as the maximum number of edges crossing a cut line between two nodes in the linear arrangement [Gav77].

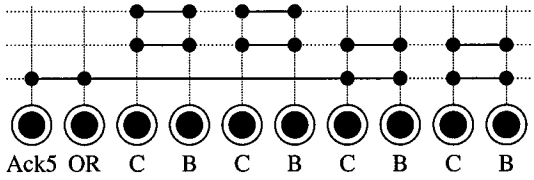
If, on the other hand there are additional, circular wrap-around edges that connect the right-most and left-most grid nodes, then the Channel Minimization Problem becomes NP-complete (*Arc Coloring*) [GJMP80]. Figure 1.15 shows an example of a one-dimensional circular grid hosting a square.

In such a circular grid, we have a choice for each edge to be routed. Due to the circular node arrangement, we can start routing a hypergraph edge at any one of its member nodes, and assign grid edges of a particular channel in, say, clockwise direction starting at this node and spanning all other member nodes of the edge. Thus, each routed hypergraph edge represents an arc of a circle. However, even if

Row of Req5 Input



Row of Ack5 Input



Column of Busy1 Output

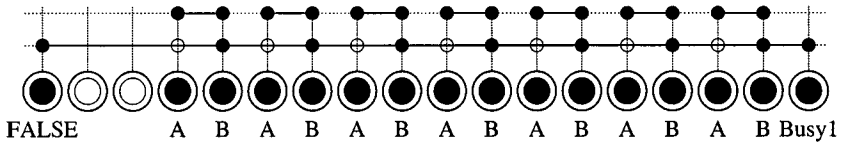


Figure 1.16: Channel assignment using the Left Edge Algorithm for selected rows and columns of the 8-to-4 Daisy Chain Concentrator in the 10×18 grid as embedded in figure 1.13.

we choose the shortest possible arc for each edge, the problem of assigning channels remains NP-complete (*Chord Coloring*) [GJMP80]. Tucker [Tuc75] showed that if we are given the orientation of each edge, i.e., clockwise or counter-clockwise, then it is possible to efficiently find a channel assignment which is within a factor of two of the minimum number of channels. If we are free to decide the edge orientation in the circular embedding, we can always find a channel assignment which is within a factor of two of the minimum number of channels: We simply ignore the wrap-around edges and apply the Left-Edge Algorithm.

For higher dimensional grids, the edge to channel assignment becomes a complex

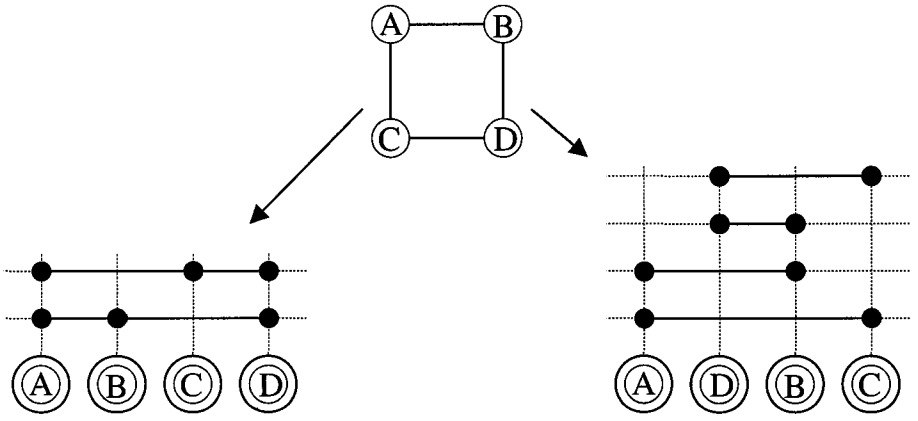


Figure 1.17: The square can be embedded using two channels (left) or four.

routing problem. See [BBG+97] for a brief overview of available techniques. However, often we can decompose the channel assignment problem into a collection of grid lines, i.e., one-dimensional grids representing the rows and columns of the original grid. In figure 1.16 we illustrate this decomposition of the channel assignment problem for our 8-to-4 Daisy Chain Concentrator. The channels for the individual rows and columns are assigned independent of one-another using the Left Edge Algorithm.

1.4.3 Maximize Bandwidth: Minimum-Cut Linear/Circular Arrangement

The available bandwidth in a network or circuit of a given topology can often be maximized by making an effort to use only a minimum number of channels along any grid edge, or across any horizontal or vertical cross-section. For example, consider an embedding of a square into a one-dimensional grid. Figure 1.17 shows how the square can be embedded using two channels or four channels. If the channels on this grid were based on time-division multiplexing, for example, then the channels' time slots could be twice as long in the two-channel embedding, thus doubling the bandwidth. In such cases we wish to minimize the maximum congestion for any grid edge, or the maximum cut width for any cross-section orthogonal to a particular grid

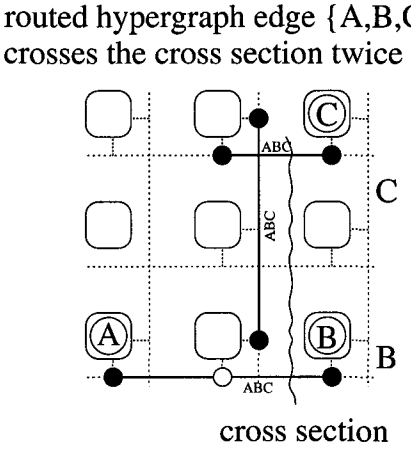


Figure 1.18: Cut width in a 3×3 grid with one channel per edge.

dimension. The cut width is the cumulative congestion through the cross-section, cf. figure 1.18.

Definition 6 *Cut Width C_{ij}*

The cut width $C_{ij}(f_E, E)$ of a routing f_E of edges E in a grid $G(V_G, E_G)$ is the number of times that routed edges cross a cross-section between coordinates points i and $i+1$ in dimension j.[1]

$$C_{ij}(f_E, E) = \sum_{\{\{u,v\}:\{u,v\}\in E_G \text{ and } u_j=i \text{ and } v_j=i+1\}} congestion(\{u,v\})$$

Thus, the problem definition comes in two flavours: one for the grid edge congestion, and one for the cut width across a grid cross section.

Problem 2 *Minimum-Cut Grid Arrangement*

Instance: *A hypergraph $H(V,E)$, a d-dimensional grid $G(V_G, E_G)$ with a number of parallel channels, and a cut width k.*

Question: *Is there an embedding (f_V, f_E) such that*

(a) *Minimum Congestion*

$$congestion(f_E) \leq k?$$

[1]Notation: u_j is the j-th coordinate of u

(b) *Minimum Cross-Section*

$$\forall\, i \in \{1, \ldots, d\}, j \in \{1, \ldots, N_i\} \qquad C_{ij}(E, f_E) \leq k?$$

In its simplest form, the one-dimensional grid, both Minimum Congestion and Minimum Cross-Section are identical. In the 1-D grid, an optimal routing f_E is implied by the placement f_V. This simpler problem is NP-complete and known for graphs as *Minimum-Cut Linear Arrangement* [GJ79] and thus Minimum-Cut Grid Arrangement is also NP-complete. For a linear grid arrangement, the maximum cut width is also the minimum number of channels required [Gav77].

Heuristics for solving *Minimum-Cut Linear Arrangements* for hypergraphs exist [CS87], and for special graphs, several polynomial-time algorithms have been found. Chen and Lee [CL92] provide an algorithm that recognizes graphs with maximum cut width 3 in linear time with respect to the number of nodes. Chao and Sha [CS92] provide an algorithm that finds an approximation within a constant factor of the optimum in linear time with respect to the number of edges for outerplanar graphs. A more general problem called the *Multicommodity Flow Problem* is investigated in [KPST94] and used to find approximations to *Minimum-Cut Linear Arrangement.* Klein et al. [KPST94] show that a node ordering that approximates *Minimum-Cut Linear Arrangement*, such that the maximum cut width is within a multiplicative factor of $O(\log N)$ of the optimal maximum cut, can be found in $O(|E|^2 \log |E|)$ expected time.

Let us examine how Minimum-Cut Grid Arrangement affects our running example. Let us assume our 10×18 grid has a capacity of two parallel channels. Can we still embed our 8-to-4 Daisy Chain Concentrator, i.e., is there an embedding (f_V, f_E) such that congestion$(f_E) \leq 2$? If we look at the embedding in figure 1.13 on page 14 we note that some grid edges carry three hypergraph edges. However, if we change the routing of the edges slightly using the same placement, we can reduce the maximum number of parallel channels to two. Figure 1.19 shows a routing using at most two parallel channels. Thus, the answer to the question is "yes, two parallel channels suffice." What is the answer if the grid has only one channel for each edge? Can we still find an embedding (f_V, f_E) such that congestion$(f_E) \leq 1$? Possibly. It is easy to see that the current placement does not permit such a routing.

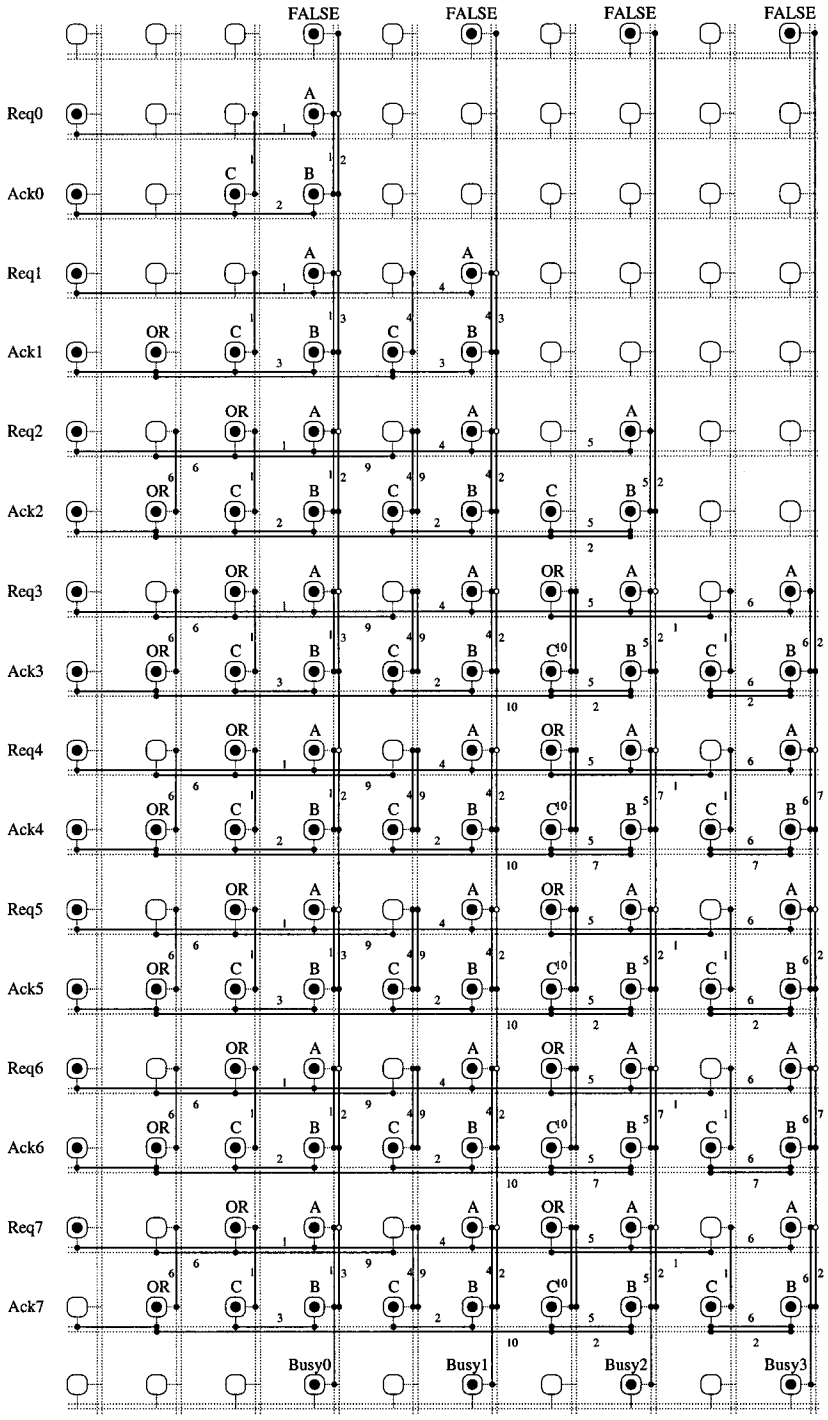


Figure 1.19: An embedding of the 8-to-4 Daisy Chain Concentrator into the 10×18 grid with two parallel channels.

To see this, we just need to recognize, for example, that the number of hypergraph edges spanning across the grid edges between columns 2 and 3 exceeds the number of rows. However, the total number of grid edges, exceeds the cumulative grid edge congestion: There are 332 grid edges and the routings shown in figures 1.13 and 1.19 have a cumulative grid edge congestion of $\sum_e \text{congestion}(e) = 309$ and 304, respectively. Thus, until proven otherwise, it may be possible that someone will find a placement and routing that does not require more than one channel per edge.

The Minimum-Cut Grid Arrangement problem is not to be confused with a similar important problem: *Minimum-Cut Partitioning*. *Minimum-Cut Partitioning* splits the nodes of a graph into two or more balanced subsets such that the number of edges connecting the subsets is minimized. Compared to *Minimum-Cut Linear Arrangement*, *Minimum-Cut Partitioning* only minimizes the centre cut whereas *Minimum-Cut Linear Arrangement* minimizes all cuts. Still, *Minimum-Cut Partitioning* is NP-complete, cf. *Max-Cut* [GJ79].

The first widely used fast algorithm to approximate *Minimum-Cut Partitioning* is due to Kernighan and Lin [KL70]. Currently, the partitioner that generally produces the the smallest cut widths among the fast, near-linear time, partitioners is *hMetis* [KAKS97]. *Minimum-Cut Partitioning* plays an important role in *partitioning placement*, a class of approximate solutions to *Shortest Total Distance*, cf. sections 1.4.4 and 3.3.

1.4.4 Minimize Latency: Shortest (Total) Distance

This problem deals with minimizing the propagation delay, or latency, of signals between nodes. The closer the nodes of an edge are placed in the grid, the shorter the propagation delay. Thus, we may want to find an embedding that minimizes the maximum distance between two neighbours. First, we define the *distance* between two nodes, and then the *diameter* of a graph.

Definition 7 *distance*

The distance between two nodes u and v in a set of edges E is the cardinality of the smallest subset of E in which u and v are connected.

The distance may be thought of as the number of "hops" over edges separating u and v, or the length of the shortest path between u and v as measured in edges. The *diameter* is the maximum of all distances:

Definition 8 *diameter*
The diameter(V, E) of a graph $G(V, E)$ is the maximum distance between any two nodes.

Using this definition, we can define the problem *Shortest Distance* as the search for an embedding with a bound on the diameter of the subgraphs induced by each routed hypergraph edge:

Problem 3 *Shortest Distance*
Instance: *A hypergraph $H(V, E)$, a grid G, and a distance k.*
Question: *Does there exist an embedding (f_V, f_E) such that*

$$\forall e \in E \;\; diameter(f_V(e), f_E(E)) \;\; \leq \;\; k?$$

In the absence of any constraints on routing f_E and given sufficient parallel channels, one can always use column routing [Lei92] where the distance between placed nodes is simply the Manhattan distance between the nodes. Thus the embedding problem reduces to a placement problem. In a one-dimensional grid, this is equivalent to the NP-complete problem *Bandwidth* [GJ79].

A related problem deals with the reduction of the average separation of nodes in the grid embedding. The problem has two flavours.

Problem 4 *Shortest Total Distance*
Instance: *A hypergraph $H(V, E)$, a grid G, and a distance k.*
Question: *Does there exist an embedding (f_V, f_E) such that*
(a) *Latency*

$$\sum_{e \in E} diameter(f_V(e), f_E(e)) \;\; \leq \;\; k?$$

(b) *Wire Length*

$$\sum_{e \in E} |f_E(e)| \;\; \leq \;\; k? \tag{1.1}$$

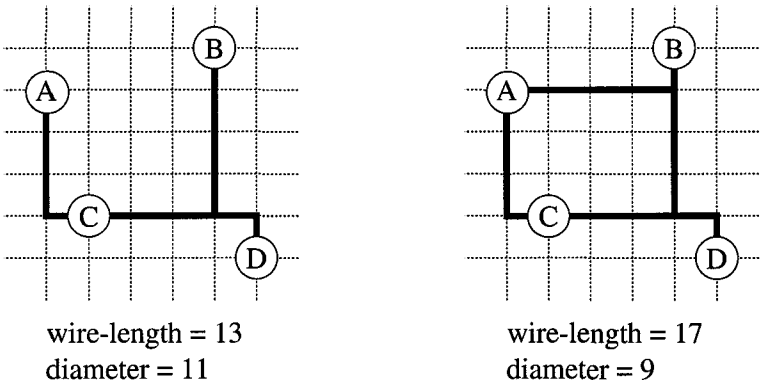


Figure 1.20: Difference between wire length and diameter for edge $\{A, B, C, D\}$.

Problem 4a asks to minimize the average delay, or *latency*, for each net assuming that signal delay is proportional to the distance travelled. Problem 4b, on the other hand, asks to reduce the average length of a net without specific regard to the maximum distance between any of the edges' nodes. Figure 1.20 illustrates this difference.

In one dimension, the routing follows from the placement and problems 4a and 4b become identical. This one-dimensional version is the NP-complete problem *Optimal Linear Arrangement* [GJ79] extended to hypergraphs. *Optimal Linear Arrangement* is a classic problem often used to reduce the total length of wire required to interconnect circuit components in a linear arrangement. Bhasker and Sahni [BS87] provide an overview.

In two or more dimensions, a routing to satisfy 4a is still easily derived from a placement by connecting all pairs of nodes in one hypergraph edge using column routing. Unfortunately, for 4b, the routing does not follow easily from the placement. To connect several points in a grid using a minimum of wire requires the computation of a *Rectilinear Steiner Tree* [Han66]. Finding a *Rectilinear Steiner Tree* is in itself an NP-complete problem, cf. *Geometric Steiner Tree* [GJ79]. Consequently, approximations of the wire length are used during the search for a placement.

Wire-length placement according to problem 4b is of great importance in chip manufacturing because wire length influences the size, and thus cost, of a chip. Further, a good solution to 4b is normally also a good solution to 4a, and the

combination of smaller chip size and shorter latency may allow a circuit to operate at a higher speed.

The study of problem 4b is often referred to as VLSI[2] placement or simply placement. Chapters 2.2-4 discuss VLSI placement in detail.

The routings for our running example, cf. figures 1.13 and 1.19 on pages 14 and 23, have a total wire length of $\sum |f_E| = 309$ and 304, respectively.

1.4.5 Minimize Routing: Straight Line on a Grid

For certain technologies, routing edges around corners in a grid may result in large latency penalties at each bend as signals may have to be routed from one grid dimension to another. In this case, it is desirable to embed all hypergraph neighbours on a straight line, if possible.

Problem 5 *Straight Line on a Grid*

Instance: *A hypergraph $H(V, E)$, and a grid $G(V_G, E_G)$ with a number of parallel channels.*

Question: *Is there a placement f_V such that all nodes of an edge differ only in one grid coordinate, i.e.,*[3]

$$\forall e \in E \;\; \exists i \;\; \forall u, v \in e \;\; \forall j \neq i \;\; f_V(u)_j = f_V(v)_j ?$$

This problem is known to be NP-complete for graph embeddings in a two-dimensional grid. This restricted version is called *Edge Embedding on a Grid* [GJ79]. This problem is a true decision problem and not a disguised minimization problem. For hypergraph topologies of a more-or-less random nature, such as an electronic circuit, a solution is unlikely to exist. However, for some popular topologies, such embeddings exist, cf. chapter 5, and *Edge Embedding on a Grid* can be solved for these instances. Given a placement that solves *Straight Line on a Grid*, we can always embed the edges in a straight line, provided there are sufficient channels. To determine the number of channels required, we solve *Channel Minimization* from section 1.4.2 for the subhypergraphs induced by each grid row or column.

[2]Very Large Scale Integration

[3]Notation: $f_V(u)_i$ indicates the dimension-i grid coordinate of $f_V(u)$.

1.5 Focus Areas

The focus of the remaining chapters lies in the study and proposal of fast solutions to problem 4b *Wire Length* and 5 *Straight Line on a Grid.*

In chapter 2, we examine the technologies that provided some of the motivation for researching the placement problem. Section 2.2 examines technologies for three-dimensional VLSI and sets up a model such that solutions to problem 4b, *Latency*, can be brought to bear. The remaining sections of chapter 2 describes how this model can be extended to deal with other, more traditional problems, namely two-dimensional VLSI placement and Field Programmable Gate Array (FPGA) layout, as well as examples of opto-electronic systems.

Existing algorithms for wire-length placement are the focus of chapter 3. Almost all deal with traditional two-dimensional VLSI as very little work has been done in the emerging field of three-dimensional VLSI.

In chapter 4 we introduce a new algorithm, *Gravity* that exhibits significant speed improvements over existing algorithms while producing competitive results. Gravity's results are compared to existing two-dimensional algorithms and to a generic partitioning placement method extended to three dimensions using a leading partitioner.

Finally, we turn our attention to placement problem 5, *Straight Line on a Grid.* We examine how some existing interconnection topologies can be placed into a grid without bends. Then we present a method for embedding the newer *star graph* topology efficiently without bends. Since its first publication in 1987 [AHK87], the star graph has received a lot of attention because it exhibits important topological parameters that are superior to established topologies such as the hypercube and the torus.

Chapter 2

Placement Model Motivation

In this chapter, we will take a closer look at the motivations leading to the placement model. First, we revisit the generic d-dimensional grid model, which was introduced in section 1.4. Then, we will examine several technologies from the fields of VLSI (3-D VLSI, 2-D VLSI, and FPGAs) and optics. For each technology, we explain how their physical implementations are modelled by the generic grid model.

2.1 d-Dimensional Grid Model

We recall definition 3 from page 8:

Definition 3 *Grid*

An $N_1 \times \cdots \times N_d$ grid $G(V, E)$ *with* k *parallel channels is a graph whose nodes are the points in the* d*-dimensional integer space* $\{0, \ldots, N_1 - 1\} \times \cdots \times \{0, \ldots, N_d - 1\}$*, and whose edges connect nodes that are separated by a euclidean distance of 1. The multiplicity* $m(e)$*, i.e., the number of parallel edges, of each edge* e *is* k*.*

$$
\begin{aligned}
V &= \{v : v \in \mathbb{Z}^d \text{ and } \forall i \in \{1, \ldots, d\} \quad 0 \leq v_i < N_i\}, \text{ and} \\
E &= \{\{u, v\} : u, v \in V \text{ and } |u - v| = 1\}, \text{ and} \\
m &: E \to \{k\}.
\end{aligned}
$$

This d-dimensional grid model captures important layout characteristics of the physical implementations of various technologies with respect to the placement problem. In the following sections we will explain in more detail how this grid model is derived from several technologies: 3-D and 2-D VLSI, FPGAs, and some optical systems. This grid model also allows an extension of the presented research to other technologies that favour grid-like structures.

All technologies modelled by this d-dimensional grid have three key features in common: array-like shape of the overall implementation, square-like shape of the individual elements within, and straight-line orthogonal orientation of interconnections.

All technologies favour a physical implementation in a rectangular area, volume, or space-time region. Conventional 2-D VLSI houses an electronic circuit on a rectangular die made from silicon. 3-D VLSI is often implemented as an interconnected stack of rectangular 2-D silicon dies, forming a rectangular box-like volume, and an optical system like a photonic backplane (cf. section 2.5) is implemented as a one-dimensional array, and can be conceptually extended into a two and three dimensional array structure. The one-, two- and three-dimensional array structures may be reconfigured in real-time yielding an implementation in a rectangular four-dimensional structure. In this dissertation we will primarily examine the two and three dimensional case but we will make some remark on higher dimensions are the opportunity arises.

Secondly, the elements interconnected in these rectangular structures are often rectangular and of similar size. The circuit elements on a silicon die are usually blocked together as rectangular cells. In programmable chips (FPGAs), all cells are identical and in conventional VLSI, cells are often standardized. All cells are taken from a library of cells and have the same height and vary in width only. In optical systems, the nodes are typically identical, and their spacing dictated by their size and required infrastructure. We will explore some of these implementations in more detail in the remaining sections of this chapter. In order to simplify research in the placement problem the individual elements are treated as being of identical unit square or cube size.

Finally, many implementations of these technologies favour rectangular interconnections along horizontal and vertical lines. In 2-D VLSI, horizontal and vertical wiring layers encourage rectangular interconnections. In 3-D VLSI, layers are typically connected by vertical metal or laser connections. FPGAs have horizontal and vertical routing channels. The optical backplane developed in part at McGill through a Network of Centres of Excellence program of the Government of Canada [SH98] has parallel optical channels along a one-dimensional grid.

Together, these three reasons justify the abstraction of the various physical implementations into a uniform grid as described in definition 3. Now, we will examine in detail how the grid model can be derived independently from several technologies.

2.2 3-D VLSI

One approach for dealing with the ever increasing complexity of integrated circuits is to extend conventional two-dimensional VLSI technology into the third dimension. Three dimensional chips are more compact and thus have shorter interconnection delays due to shorter connections. Such a reduction in interconnection length for 3-D VLSI has been predicted in [VMVC97] and [MM98], and in [OS99] we presented some experimental results that showed that larger benchmark circuits exhibited a reduction of interconnection length of over 50% in three dimensions as compared to a two-dimensional chip, cf. chapter 4.

Implementations of 3-D VLSI vary considerably from theoretical [LR86, Ohm98, RT86, TW95] to practical [CV98, DNVM+94, LMV+98]. However, these implementations have in common that several 2-D chip layers are stacked and interconnected vertically by either metal vias or by optics. One of the more advanced implementations of 3-D VLSI [LMV+98] comes very close to the homogeneous 3-D grid model (cf. definition 3) which we use in this dissertation, and which has been used in previous 3-D VLSI papers [LR86, Ohm98, SvC97].

Leeser et al.'s three dimensional chip [LMV+98] consists of a stack of two dimensional programmable logic dies (cf. section 2.4). These layers of dies are interconnected with metal micro vias in the vertical dimension. This architecture, which the authors call *Rothko*, consists of layers of two-dimensional field-programmable

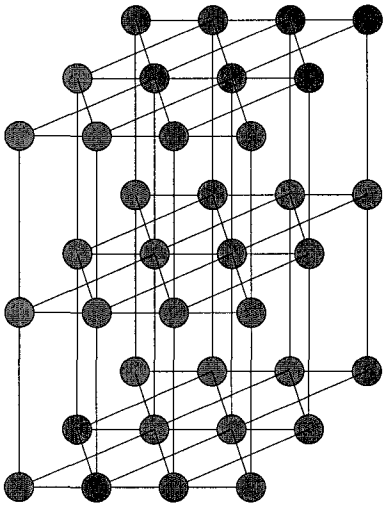


Figure 2.1: Routing-and-logic block interconnectivity in the 3-D Rothko [LMV+98] architecture (routing channels between columns not shown).

gate array (FPGA) layers. The elements of a layer are routing-and-logic blocks. These routing-and-logic blocks are arranged in a grid fashion and interconnected to routing channels to the left and right, to their neighbouring blocks to the left and right, their diagonal neighbours, as well as the neighbours in the layers above and below. Figure 2.1 shows the interconnectivity of a 3-D Rothko chip.

As mentioned above, the vertical inter-layer connections are implemented as metal vias. Such vias are metal connections through holes in the die. The diameter of these vias is 6μm. In comparison, Rothko has a 1.2μm feature size. This means a via is only about five times the size of a transistor. Consequently, inter-layer connections are possible for all routing blocks. Further, Leeser et al. [LMV+98] did not notice a difference in signal delay between diagonal intra-layer and vertical inter-layer connections.

More interesting than the detailed workings of the Rothko chip is that this technology demonstrates that dense vertical inter-layer connectivity is possible with current technology. This lends credence to the homogeneous grid model which has been used for 3-D VLSI as early as 1986 [LR86], and which we use in this dissertation, cf. definition 3 on page 8.

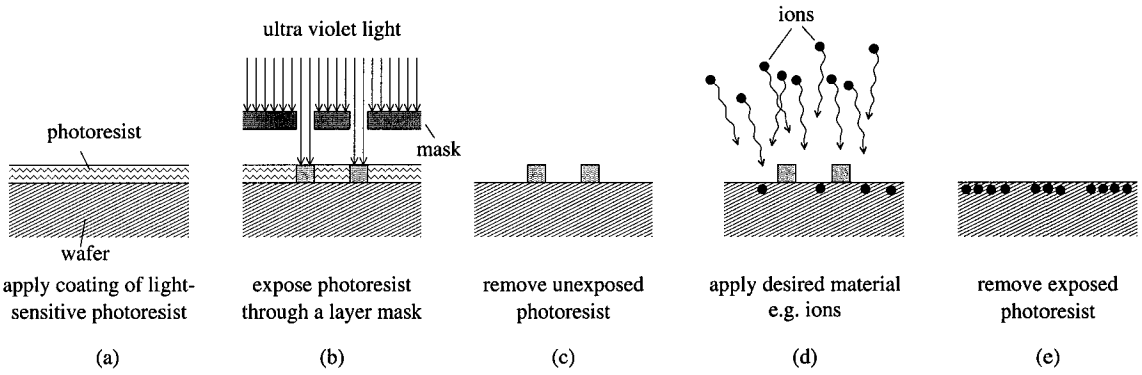


Figure 2.2: Simplified illustration of applying a layer of material to a silicon wafer: These steps are repeated many times for different materials.

2.3 2-D VLSI

Today, the vast majority of integrated circuits are made from, and on, rectangular slices of silicon. First, circular slices of purified silicon are cut from cylindrical silicon crystals. These slices, which are called wafers, are up to 30cm in diameter. Wafers are subdivided into dozens or hundreds of usually rectangular areas called dies. The integrated circuits are created on the dies in repeated photolithographic steps. The silicon is doped by depositing ions that alter the electrical properties of the silicon. Figure 2.2 illustrates this photolithographic doping step. The area to be doped is determined by the layer mask. In similar photolithographic steps, using chemical vapor deposition and metal sputtering, layers of insulators and metal interconnections are deposited. These insulator and metal layers complete the electronic circuit components, such as transistors, and their interconnections. After the circuits have been created, the wafer is cut into its individual dies, and the dies are packaged in ceramic or plastic packages with metal pins. During the fabrication process, flaws may be introduced into the circuit due to impurities in the silicon, or dust particles in the air, for instance. These damaged dies are discarded lowering the yield of the process. The yield is the number of good dies over the total number of manufactured dies. Yields vary from a few percent for new technologies to close to 100% for mature technologies. The finished product is often called a *chip*. Before a chip can be mass-produced, it has to undergo several cycles of functional testing

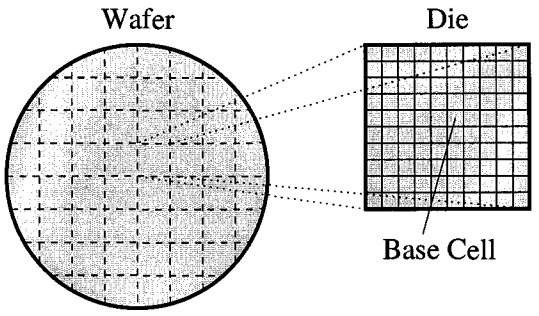


Figure 2.3: A grid of base cells on a silicon die for a channelless sea-of-gates array.

and improvement by the designer.

When chip designers design circuits from scratch, they could treat the silicon die as a blank piece of paper and create a circuit on it by manually defining all the necessary layers. This method would be the equivalent to a software designer writing a program in machine language. There may be a place for this approach, but usually it is faster and more reliable to take a more high-level approach.

Two popular high level design approaches are sea-of-gates and standard cell designs.

In channelless *sea-of-gates* chips, the chip die is pre-populated with a grid of generic base cells, cf. figure 2.3. The functionality of these base cells, as well as their interconnectivity, is defined by a few top metal layers. The chip designer can only customize these top layers. This design process reduces design time and manufacturing costs as wafers of dies with grids of base cells can be manufactured in advance and in bulk. The responsibility that remains with the chip designer is the placement of circuit components ("Which base cell should house which circuit element?") and the routing of the interconnections. In terms of the terminology used in this dissertation, sea-of-gates designers perform the following three steps. (1) Decide on a hypergraph, cf. definition 2 on page 4. The designer chooses a circuit and, by a similar transformation as described for the 8-to-4 Daisy Chain Concentrator circuit in section 1.4, a hypergraph is defined. (2) Choose a placement, cf. definition 4 on page 11, of hypergraph nodes into the grid, cf. definition 3. (3) Find a routing, cf. definition 5 on page 15, of edges in the grid.

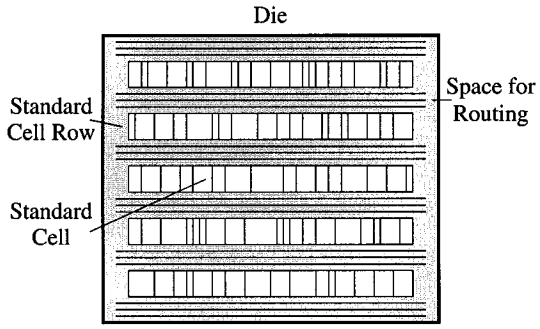


Figure 2.4: Standard cell placement on a die.

Another chip design method that uses standardized cells is *standard cell* design. Unlike a sea-of-gates dies, standard cells are not pre-manufactured. Standard cells exist in libraries for computer aided design tools. Standard cells are rectangular and have the same "height" on a die. However, cells in a standard cell library differ in functionality and "width". In the finished design, the standard cells are placed in rows onto the die. Between rows, space is left for horizontal wire tracks. Vertical connections cross standard cells at designated points called "feed-throughs." Figure 2.4 shows an example of a standard cell placement before routing of the metal interconnections. Even though the x-coordinate of a cell in its row is no longer an integer, the hope is that, with small modifications, a placement algorithm that is good for pure grid designs will also produce good solutions for standard cell designs.

In this section, we have overlooked another wide-spread design method, namely programmable chips. The functionality of a programmable chip is only defined after the chip has been manufactured. Programmable chips are described in the following section.

2.4 FPGA

Field-programmable gate arrays (FPGAs) form a distinct and important class of integrated circuits. The functionality of an FPGA is programmed by the customer in the "field". Thus, FPGAs are manufactured in bulk as standard parts.

FPGA technology varies, but some basic features are common to all FPGAs. The

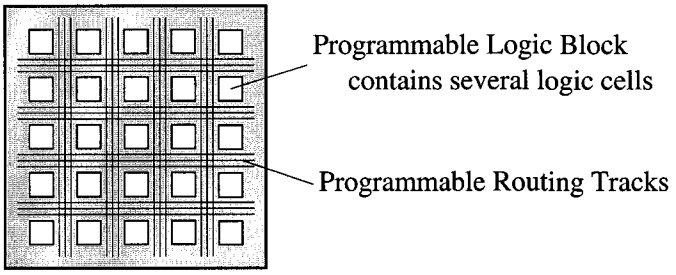


Figure 2.5: Typical FPGA topology.

die of an FPGA holds a grid of generic logic cells, the gate array, similar to base cells in the sea-of-gates architecture discussed in section 2.3. Further, horizontal and vertical tracks are set aside for interconnection routing. The customer can program the interconnections within the generic logic cells and on the routing tracks in order to define the functionality and interconnection of those logic cells. A simplified typical layout structure on an FPGA die is shown in figure 2.5.

There are three dominant kinds of technologies for these programmable interconnections. These technologies vary from permanent one-time programming, over erasable semi-permanent, to temporary programming.

The most permanent technology uses so called *anti-fuses* to make permanent connections. When a programming current is forced through an anti-fuse, the generated heat causes a chemical reaction in the anti-fuse, which establishes the desired connection.

A semi-permanent method for establishing FPGA (dis)connections is provided through electrically programmable read-only memory (EPROM) technology. With this technology, a connection through a programmable transistor can be severed by increasing the transistor's activation voltage with the help of a programming voltage. Such connections remain severed until the FPGAs programming is erased by ultra-violet light or by an erasing-voltage.

The most temporary FPGA programming technology uses static random-access memory (SRAM) technology to program connections. Each connection has a static state memory that remembers if a connection is open or closed. This state information must be downloaded into the FPGA at power-up. An SRAM based FPGA

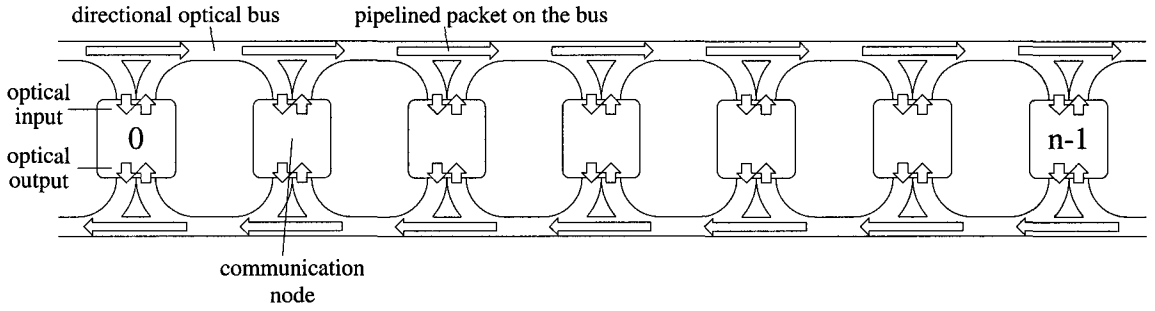


Figure 2.6: Linear array processors with pipelined buses.

maintains its programming only as long as it is supplied with power.

An in-depth treatment of FPGA technologies can be found in [Smi97].

Due to the additional space required for the programming infrastructure, and the generic nature of the cells, FPGAs contain less circuit elements then the custom made chips described in section 2.3. In volume, this makes FPGAs more expensive than conventional chips. Thus, FPGAs are used for prototyping, or when only small quantities of a particular circuit are needed, or when there is no time to wait for the fabrication of a custom-made chip.

As with the VLSI technologies described in the previous section, FPGA designers need to decide only on a circuit (=hypergraph), on the placement of the circuit elements in the FPGA grid, and on the routing of the interconnections (=edges).

2.5 Optical Systems

Besides integrated circuits, other technologies also benefit from good grid placement methodologies. A few newer technologies for which grid-like implementations are proposed are in the field of optics. In particular, [Dow92, GMH+91, SH98] propose multi-dimensional grid structures.

In [GMH+91] Guo et al. introduce *array processors with pipelined buses.* A one dimensional array of this kind features n processors in a row. All of these processors are connected to two directional optical buses of opposite direction. Optical signals carrying packets of bits originating from upstream processors can be received, and signals can be sent to downstream nodes. Figure 2.6 shows a schematic for n pro-

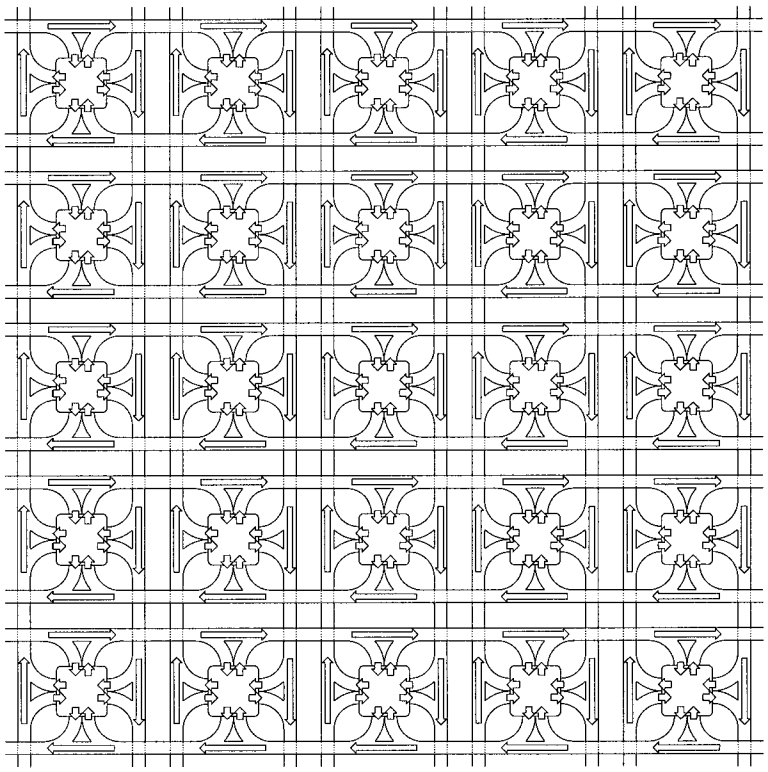


Figure 2.7: 5×5 array processors of pipelined buses.

cessors connected to up- and downstream buses. These buses are called "pipelined buses" because several optical signals from different processors can be on the bus at the same time in pipeline fashion. Guo et al. [GMH+91] call the maximum time required for a packet to travel from node 0 to node $n-1$ on the bus the *bus cycle.* The nodes on a bus are synchronized to simultaneously transmit packets at the beginning of each bus cycle. Thus, a bus cycle represents one hop of a packet from any node on the bus to any other node. Effectively, there exists one non-segmentable channel per node.

Despite some performance limitations of Guo et al.'s [GMH+91] proposed architecture, as noted in [Szy95], the concept of "bus cycle" has an interesting implication in higher dimensions. Guo et al. [GMH+91] propose an extension of their architecture to two dimensions by forming rows and columns of one-dimensional buses. In this case, every processor is connected to two uni-directional row buses and two

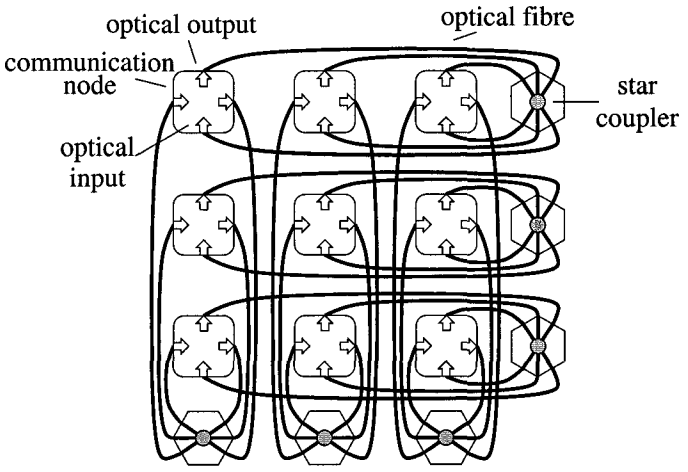


Figure 2.8: A 3×3 grid of communication nodes is connected by star couplers along every grid line.

uni-directional column buses. Figure 2.7 shows the schematic of four-by-four array processors on pipelined buses. In this two-dimensional arrangement, source and destination nodes do not necessarily share a common row or column bus. In this case, transmission of one packet requires at least two bus cycles since the packet has to traverse a row and a column bus. However, when source and destination nodes share a row or column, transmission requires only one bus cycle. With this consideration, Guo et al. [GMH+91] suggest that a bendless embedding of a network topology would be beneficial as synchronized transmission of packets between neighbours would require only one bus cycle as opposed to two or more in an embedding with bends. This problem is modelled by a grid in which the array processors form the grid nodes, a hypergraph which describes the topology to be embedded, and by the placement problem Straight Line on a Grid, cf. problem 5 on page 27. Guo et al. [GMH+91] examine bendless embeddings of binary trees and hypercubes. We will consider bendless embeddings in more detail in chapter 5. This architecture of arrays with pipelined buses is described and analysed in greater detail by Akl [Akl97] and Pavel and Akl [PA96], where bounds on time and area for certain applications are presented.

In [Dow92], Patrick Dowd introduces a different grid-like optical architecture

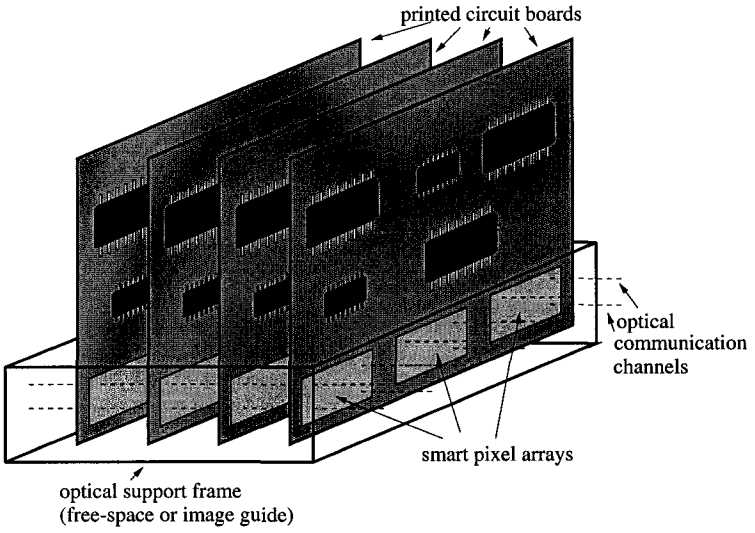


Figure 2.9: Schematic of an intelligent optical network as described in [SH98]

for two and three dimensions. In this architecture, all communication nodes on the same grid line are interconnected by optical fibres via a *star coupler.* A star coupler is a passive device that physically fuses n input fibres and broadcasts the input signals to n output fibres. Thus, a signal sent by any of the grid nodes is received simultaneously by all other nodes on the same grid line. Figure 2.8 shows a two-dimensional three-by-three grid with star couplers connecting all nodes along the same row or column. Multiple parallel channels are achieved for each grid line by using multiple wavelengths (WDM) and time division multiplexing (TDM).

When source and destination nodes are on the same grid line, i.e., connected by a star coupler, transmission of a packet requires one hop. If source and destination nodes do not share a common grid line, packet transmission requires at least two hops. Thus Dowd points out the advantages of finding bendless embeddings in which all neighbour to neighbour transmissions can be completed in one hop. For this purpose, Dowd examines some bendless hypercube embeddings.

In [SH98], Szymanski and Hinton describe the architecture of an optical network and discuss its extensions to two and three dimensions. The one-dimensional version of this network, prototypes of which have been built, features printed circuit boards aligned in an optical support structure. The printed circuit boards communicate

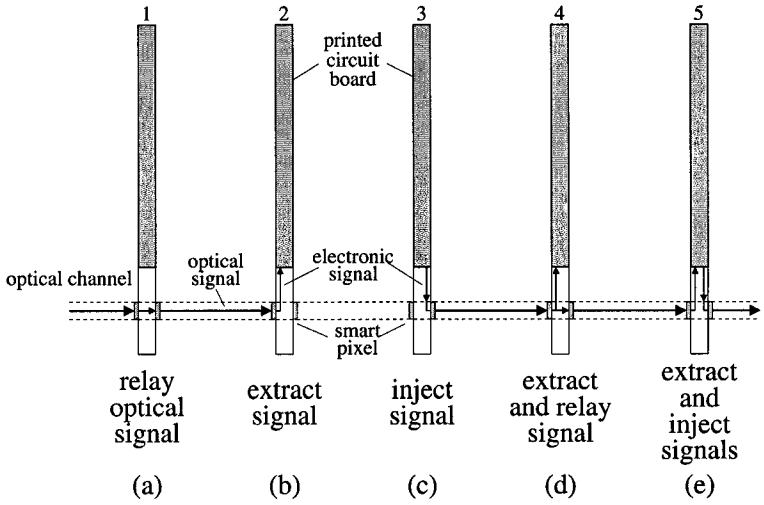


Figure 2.10: Operation of smart pixels.

with their neighbouring boards via thousands of parallel optical channels. Figure 2.9 shows a simplified schematic of the physical arrangement.

The thousands of parallel optical channels that interconnect the printed circuit boards are controlled by an array of *smart pixels* connected to each board. These smart pixels can relay an optical signal from one printed circuit board to the next, cf. figure 2.10a. Alternatively, the smart pixels can extract an optical signal and convert it into an electronic signal, cf. 2.10b, or they can convert an electronic signal from the printed circuit board into an optical signal and inject it into the optical channel in the backplane, cf. 2.10c. Combinations of relaying and extraction, cf. 2.10d, or extraction of one signal and injection of another are also possible, cf. 2.10e.

Besides being able to implement hundreds or thousands of parallel channels, a further benefit of using smart pixel technology is that the optical channels are segmentable, i.e., different signals can use the same channel as long as they do not use overlapping segments between boards. For instance, in figure 2.10, the signal relayed by board 1 to board 2, and the signals originating from boards 3 and 5, respectively, all use the same optical channel over different segments. The resulting topology of this one-dimensional backplane thus resembles a one-dimensional grid with multiple channels, cf. definition 3, similar to the illustration in figure 1.14 on page 17. In contrast, the proposed architectures of Guo et al. [GMH+91], and

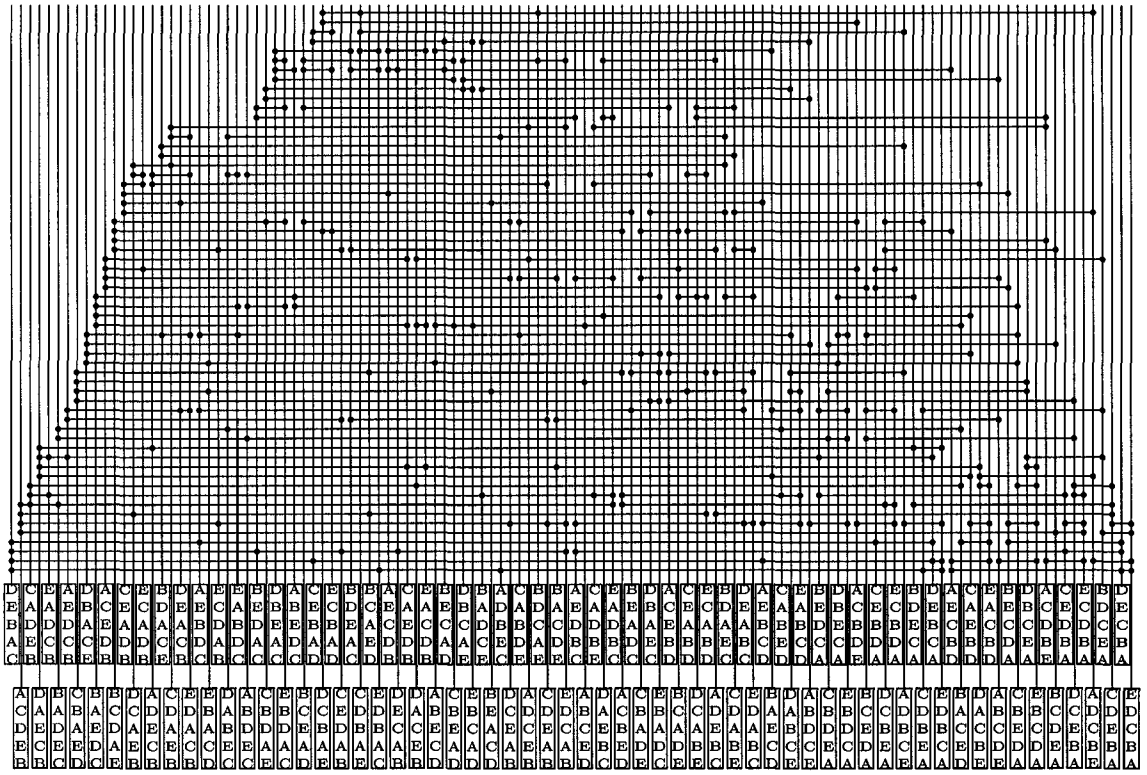


Figure 2.11: Best known minimum-cut embedding of a 120-node star graph.

Dowd [Dow92] do not allow for segmentable channels and thus impose additional routing constraints. For Szymanski and Hinton's architecture, the problems Channel Minimization, cf. problem 1 on page 17, and Minimum-Cut Grid Arrangement, cf. problem 2 on page 21, are of particular interest. The channel assignment that solves Channel Minimization can be computed in linear time using the Left Edge Algorithm, cf. section 1.4.2. The Minimum-Cut problem has been investigated for selected interconnection topologies in [Obe95, SH98]. Figure 2.11, for example, shows the best known embedding of a 120-node star graph. More general solutions were addressed in section 1.4.3.

Szymanski and Hinton extend this architecture to two and three dimensions by arranging rows, columns, and layers of printed circuit boards, and connecting them to one dimensional optical backplanes in each dimension. Despite the differences with respect to Guo et al.'s [GMH+91] and Dowd's [Dow92] architectures, the optical backplane also benefits from bendless embeddings. In a bendless embedding, the

routing is simplified since no provisions have to be made for routing a signal from one one-dimensional backplane to another. In addition, any possible latency penalties resulting from these backplane transitions are avoided with a bendless embedding.

In this chapter, we have reviewed several technologies from the fields of VLSI and optics. We have shown how all these technologies are modelled by our grid and embedding definitions from chapter 1. In the remaining chapters, we will examine some methodologies for solving the Wire-Length and Straight-Line problems (cf. problems 4b and 5). Chapter 3 takes a look at previous placement algorithms for the Wire-Length problem. In chapter 4, we introduce a new placement algorithm that finds competitive solutions to the Wire-Length problem in a fraction of the time of previous algorithms. Finally, in chapter 5, we look at some embeddings of popular interconnection topologies without bends, and we show how the star graph can be embedded efficiently without bends into grids.

Chapter 3

Existing Wire-Length Placement Approaches

This chapter provides an overview of the extensive research dedicated to the wire-length placement problem, cf. problem 4b. As briefly mentioned in section 1.2, the wire-length minimization problem is an important problem because it translates directly into manufacturing cost savings in VLSI. Shorter interconnections require less space, and have less capacitance and inductance. Reducing the length of interconnections thus shrinks expensive chip dies and speeds up circuitry, cf. [SM91].

Before we look at various placement methodologies for arbitrary circuits (=hypergraphs), we will explore in section 3.1 what some of the analytical bounds for minimum wire-length placements are. Afterward, we turn our focus to the research in competing placement methodologies. In particular, we will study four classes of approaches: placement using *simulated annealing*, *partitioning placement*, *force-based placement*, and *quadratic placement*. Before we close this chapter with a summary, we include two short sections on genetic placement and a look at placement algorithms in three dimensions.

3.1 Analytical Estimates

In light of the cost savings and performance gains associated with a reduced total connection length, it would be desirable to have a reasonably tight upper bound

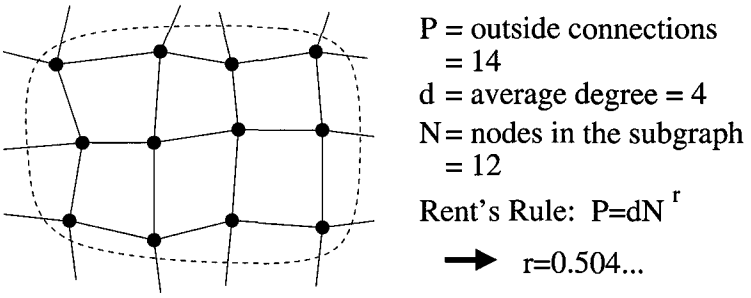


Figure 3.1: Rent's Rule

on the minimum wire length required to place-and-route a circuit. If such a bound could be computed in a reasonable amount of time for a given circuit, we could then compare how well our placement algorithm performed and perhaps try harder to improve the placement.

The only useful bound for general VLSI placement is due to Donath [Don79]. Donath derived an upper bound on the minimum total wire length. This upper bound is of closed form and based on an empirical rule called *Rent's Rule* [LR71]. In a graph used to represent an electronic circuit, Rent's Rule relates the number of edges leaving a partition of the graph to the number of nodes in a partition. This rule is not universally applicable to all graphs and should be treated as a rule of thumb for electronic circuits.

Definition 9 *Rent's Rule (empirical rule for electronic circuits)*

$$P = dN^r$$

where P is the number of edges leaving a subgraph of a circuit graph, d is the average degree of a node in the graph, N is the number of nodes in the subgraph, and r is the empirical Rent exponent between 0 and 1.

Figure 3.1 illustrates Rent's Rule. Rent's Rule was significant because it was observed to hold recursively for hierarchical subdivisions of circuit graphs. However, Rent's Rule does not hold for many degenerate graphs, as, for example, the complete graph. For electronic circuits, the Rent exponent r usually ranges from 0.5 to 1.

Based on Rent's Rule, Donath developed an upper bound on the minimum wire length of a placement in a square grid assuming a recursive partitioning that obeys

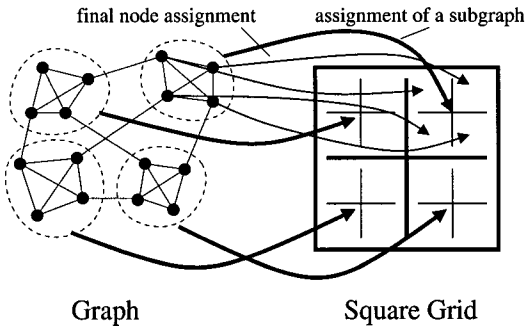


Figure 3.2: Virtual placement procedure employed by Donath. [Don79]

Rent's Rule. These partitions are then virtually assigned to corresponding partitions of a square grid which represents the chip die. Figure 3.2 illustrates this virtual placement procedure. This placement is virtual and not real because the actual circuit is not known. All that is known about the circuit is its size, and that it obeys Rent's Rule with a Rent exponent r. Based on this virtual placement and the Rent exponent, Donath derived the expected interconnection lengths and found an estimate for the average wire length R of an interconnection. For $r > 1/2$, the average interconnection length in a $\sqrt{N} \times \sqrt{N}$ grid is

$$R = \frac{2}{9}\left(7\frac{N^{r-\frac{1}{2}}-1}{4^{r-\frac{1}{2}}-1} - \frac{1-N^{r-\frac{3}{2}}}{1-4^{r-\frac{3}{2}}}\right)\frac{1-4^{r-1}}{1-N^{r-1}}. \tag{3.1}$$

While $1/2 < r < 1$ and as N becomes large, the terms $N^{r-3/2}$ and N^{r-1} become zero. Thus, asymptotically, this estimate approaches

$$R \approx \frac{14}{9}\cdot\frac{1-4^{r-1}}{4^{r-\frac{1}{2}}-1}N^{r-\frac{1}{2}} \tag{3.2}$$

$$\sim N^{r-\frac{1}{2}}. \tag{3.3}$$

This estimate can be used as an upper bound on the minimum wire length required to place-and-route a circuit. Unfortunately, this bound is exponentially sensitive to the empirical Rent exponent r. In order to measure r, the circuit has to be recursively partitioned. But such a measurement of r is essentially equivalent to performing a partitioning placement, cf. section 3.3. Donath acknowledged that an actual placement would normally produce a shorter wire length, and thus produce

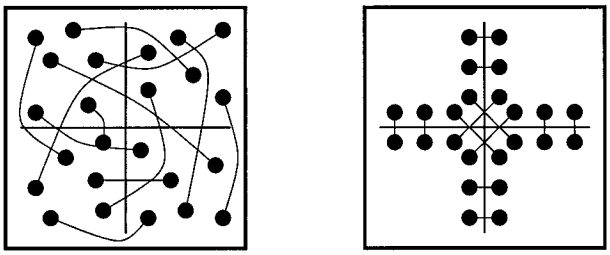


Figure 3.3: Virtual placement according to Donath (left) and Stroobandt and van Campenhout (right).

a tighter bound. Nevertheless, Donath's bound may serve as a rule-of-thumb if the Rent exponent is known, or can be estimated from experience.

Recently, Stroobandt and van Campenhout [SvC97] improved Donath's estimate. When computing average interconnection lengths between circuit elements of neighbouring virtual partitions, Donath assumed that these elements would be uniformly distributed over the square area assigned to them. Stroobandt and van Campenhout made the assumption that a good placer could always place these elements close to the border of their assigned areas. Figure 3.3 illustrates this difference. Although this assumption is not always justified, it lead Stroobandt and van Campenhout to arrive at an estimate that is closer to wire lengths that have been obtained experimentally using placement algorithms. Stroobandt and van Campenhout also extended their estimates to three dimensional VLSI. However, for lack of a 3-D placement algorithm, they where unable to compare their results to experimental values.

3.2 Simulated Annealing

One of the most powerful methods for VLSI placement is placement by *simulated annealing* [KSS98, SS93, SS95, SS97]. Simulated annealing and its application to VLSI was introduced by Kirkpatrick et al. in 1983 [KGV83]. This method mimics a physical annealing process in which a liquid slowly freezes and becomes a solid crystal. In this context, an unsolved optimization problem such as VLSI placement, initially in a random state, represents the liquid. An optimal solution to this

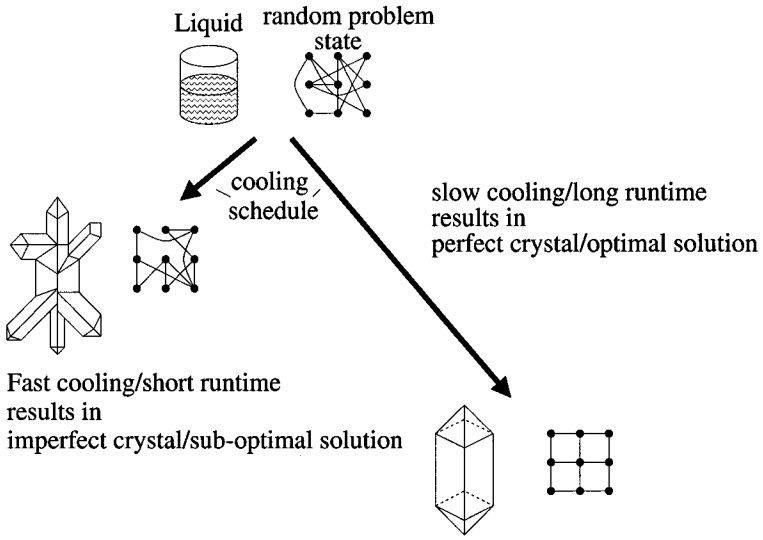


Figure 3.4: Simulated annealing mimics the formation of a crystal.

optimization problem corresponds to a perfect crystal. The simulated annealing algorithm performs a simulated cooling of the "liquid" into a "crystal". The slower the algorithm's cooling schedule, the more perfect the crystal, and thus the solution. Figure 3.4 illustrates this principle.

At the heart of any simulated annealing algorithm are three elements: a state generator, an energy function, and a cooling schedule. The state generator generates (usually in a randomized fashion) a new state. In the VLSI placement problem, each state represents a valid VLSI placement. Typically, but not necessarily, a new state is generated by permuting the current state. The energy function computes the "energy" of a state. The energy is a measure of the badness of a state. In VLSI placement, the energy function would usually compute an estimate of the wire length of the placement. A state that forms an optimal solution has the lowest possible energy. The cooling schedule determines how a number T which represents the "temperature" is monotonically reduced over the run time of the simulated annealing algorithm.

With these elements in place, a simulated annealing algorithm works as shown in figure 3.5. First, the state generator F generates a new state S', usually by permuting the current state S, cf. step 3. The change in energy ΔE is computed, cf.

Input: initial state S,
final temperature T_F.
Required subroutines:
cooling schedule $\{T_0, T_1, \dots\}$,
state generator $F()$,
energy function $E()$.

1: $i \leftarrow 0$
2: while $T_i > T_F$ do
3: $S' = F(S)$ *generate new state*
4: $\Delta E = E(S') - E(S)$ *compute change in energy*
5: if random($[0, 1)$) $< e^{-\Delta E/T}$
6: $S' = S$ *accept S' with probability $e^{-\Delta E/T}$*
7: $i \leftarrow i + 1$
8: output S

Figure 3.5: Generic simulated annealing algorithm.

step 4, and the new state is accepted if the new energy is lower, or with probability $e^{-\Delta E/T}$, cf. steps 5 and 6. The process is repeated until the temperature T_i at step i has dropped to the final temperature T_F.

In 1985, Mitra et al. [MRSV85] proved that simulated annealing would settle in an optimal solution state provided that the state transitions were well behaved and cooling was slow enough. By "well behaved" state transitions, we mean that after a start-up period, there must be a non-zero probability that any given state is the current state at every iteration. Further, the cooling schedule has to maintain the temperature roughly at $1/\log i$ at step i. We will look at Mitra et al.'s results in more detail later. First, we note that given any finite run time, simulated annealing cannot be guaranteed to produce an optimal result, even if we know that the algorithm will settle into an optimal state eventually. We know as a fact, that at time step i, there is always a possibility the simulated annealing algorithm is in a suboptimal state. This possibility results from the simulated annealing algorithm accepting a

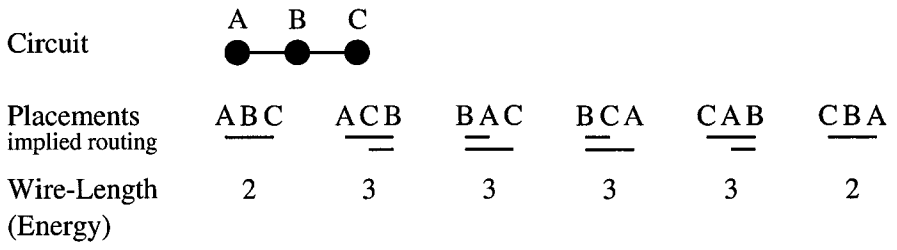


Figure 3.6: Possible linear placements of a simple circuit.

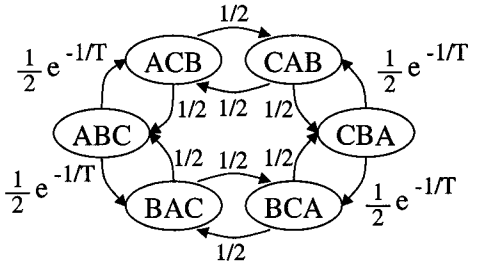


Figure 3.7: Markov chain of the possible placement with transition probabilities.

worse state with finite probability $e^{-\Delta E/T}$. However, it is comforting to know that simulated annealing becomes more and more likely to be in an optimal state as time progresses. This is a guarantee that the algorithm will not settle into a local minimum.

For purposes of illustrating VLSI placement by simulated annealing and Mitra et al.'s results, we consider a circuit with three elements A, B, and C. A is connected to B, and B is connected to C. A, B, and C are to be placed into a linear order such that the wire length is minimized. Figure 3.6 shows the possible placements and their energies, i.e., wire lengths. As our state generator, we choose a function that randomly exchanges the positions of two neighbouring elements. Figure 3.7 shows the resulting Markov Chain at temperature T. A Markov Chain lists all possible states, and their transition probabilities.

We observe two relevant parameters of the Markov Chain. The first parameter r is the radius, i.e., the maximum distance to reach any state s' from a state s that is not a maximum-energy state, and for which this maximum distance is a

minimum: $r = \min_{s \in S - S_{\max}} \max_{s' \in S} \text{distance}(s, s')$. In our example $r = 3$ is the maximum number of hops to reach any state from the two non-maxima ABC and CBA. The other parameter L is the maximum possible change in energy in a state transition. In our example, $L = 1$. Mitra et al. proved that if the cooling schedule sets temperature T_i at time step i at at least

$$T_i = \frac{\gamma}{\log(i+c)} \quad \text{for any constant } c > 1 \quad \text{and } \gamma \geq rL, \tag{3.4}$$

then the simulated annealing algorithm will settle in an optimal state. For our example, this results suggests a cooling schedule of

$$T_i = \frac{3}{\log(i+2)}. \tag{3.5}$$

After k iterations, Mitra et al. showed that the probability of being in a suboptimal state $P_{\text{suboptimal}}$ is bounded by

$$P_{\text{suboptimal}} = O(\frac{1}{k^{\min(a,b)}}) \tag{3.6}$$

where $a = \frac{w^r}{r^{rL\gamma}}$,

$w =$ minimum probability that any one neighbour state is generated,

$b = \frac{\delta}{\gamma}$, and

$\delta =$ (next to minimum energy) $-$ (minimum energy).

In our example, $\delta = 1$, $w = 1/2$, $r = 3$, $L = 1$, $\gamma = 3$, thus $b = 1/3$, and $a = 1/157,464$. Consequently, the probability that we are in a suboptimal state at step k is bounded by

$$P_{\text{suboptimal}} = O(\frac{1}{k^{1/157,464}}). \tag{3.7}$$

Probably the most successful simulated annealing placement algorithm, called Timberwolf, is due to Sun and Sechen [SS93, SS95, SS97]. Timberwolf is basically an implementation of the cell-swapping algorithm we employed in our example above

(cf. figure 3.7), adapted for standard cell placement. Recall that in standard cell placement, cells of varying widths are placed onto a chip into several rows. Timberwolf sets a maximum row length at the start of the algorithm. Empirically, they determined a length limit of 12% over the average row length (cumulative cell length/rows) works well. For the state generation (step 3 of figure 3.5) at each iteration of Timberwolf, a random cell A is selected, and a second random location is chosen. If cell A can be inserted at the new location without exceeding the new row's length limit, it will be inserted. Otherwise, cell A and the cell B at the new location are exchanged, provided neither row's length limit is violated. Then, the change in wire length is estimated, and the new state is accepted if it has an improved wire-length estimate, or with probability $e^{-\Delta E/T}$. Sun and Sechen further improve the runtime of Timberwolf by including hierarchical clustering in which netlists of the circuit are contracted to form clusters of cells. These clusters can then be used by Timberwolf to compute a coarse placement. After the coarse placement has been computed, the clusters are broken up and a fine placement is computed. In all, there are three levels of hierarchy processed in different stages: plain cells (stage 3), clusters of cells (stage 2), and clusters of clusters of cells (stage 1). The cooling schedule drops the temperature quickly during stage one to a constant level at which it is held throughout stage 2, and then drops off during stage 3 to its final level.

3.3 Recursive Partitioning

A generally much faster approach to VLSI placement is partitioning placement [Bre77, SK88, GLC93, YW96]. The most basic partitioning placement algorithm was pioneered by Breuer [Bre77]. In Breuer's algorithm, the circuit is recursively partitioned into equal halves until only one node remains. At the same time, the chip area is split in halves, alternately vertically and horizontally. When only one circuit node remains, it is assigned to the corresponding piece of the chip area. When partitioning the circuit, care is taken to cut only through a minimum number of interconnections. Figure 3.8 illustrates this placement method.

Although minimum-cut partitioning is as NP-complete a problem as minimum

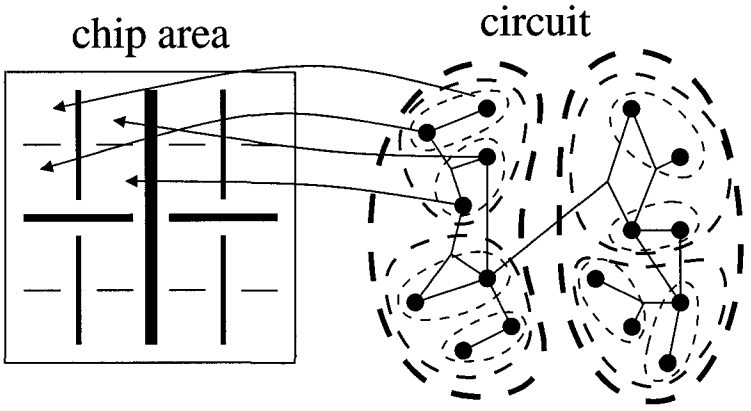


Figure 3.8: Partitioning placement (thicker lines indicate earlier cuts)

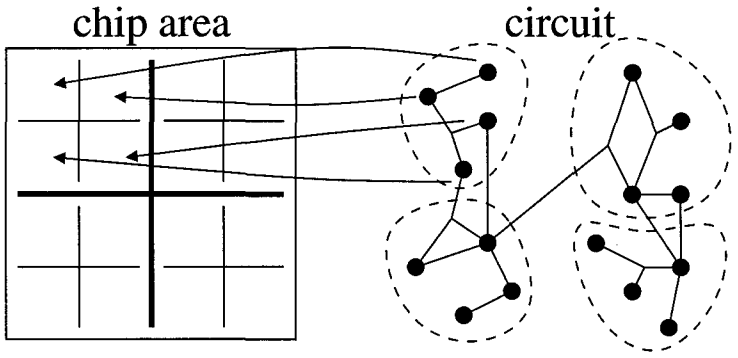


Figure 3.9: Quadrisection placement (thicker lines indicate earlier cuts)

wire-length placement, cf. [GJ79] and section 1.4.3, several good and fast approximation algorithms exit, e.g. [FM82, DD96, KAKS97]. The utilization of fast partitioners ensures that a placement can be computed in a relatively short period of time. Unlike simulated annealing where ultimately the optimal placement will be found, a good wire-length placement is only a wanted by-product of a partitioning placer. No direct effort is made to minimize the over-all wire length of the placement. However, by performing minimum-cut partitioning, the most heavily interconnected parts of the circuit are severed last, and those cells are placed most closely together.

Suaris and Kedem [SK88] suggested to recursively partition the circuit into four subcircuits and simultaneously divide the chip area into four quadrants, cf. figure 3.9.

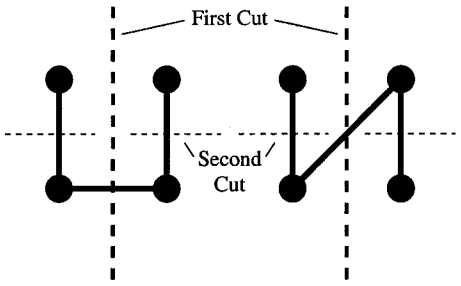


Figure 3.10: Non-optimal partition placement: the placement on the right has longer wire length, although both are results of perfect partitionings.

This placement method, called quadrisection, was expected to take advantage of the two-dimensional nature of the placement problem. Traditional partitioning placement can lead to non-optimal placements by ignoring the two dimensional nature of a chip die, e.g. figure 3.10. Quadrisection can avoid these problems. Whether quadrisection actually leads to better results is not clear. Suaris and Kedem did not offer a direct comparison to traditional partitioning placement, and our own tests have been inconclusive. However it is unlikely that quadrisection can perform worse than traditional partitioning placement. Leeser et al. [LMV+98] mentioned an extension of quadrisection to three dimensional VLSI where the chip volume was recursively subdivided into octants. However, no details were published.

Partitioning placement draws its strength from the quality of the underlying partitioner. Most algorithms use an implementation based on a method by Kernighan and Lin [KL70] which was later extended and refined by Fiduccia and Mattheyses [FM82]. Fiduccia and Mattheyses' algorithm consists of several passes that run in linear time, each. The algorithm starts with two random partitions. For each node a gain, i.e., the improvement in cut width that would result in moving the node to the other side, is computed. Then, the node with the highest gain and which does not violate the arbitrarily preset balance criterion is moved to the other side and locked in place. The gains of the other nodes are updated, and of the remaining nodes the one with the highest gain is moved. This process continues until all nodes are locked or until no more moves are possible because they would violate the balance criterion. The partitioning with the smallest cut width during the previous run is

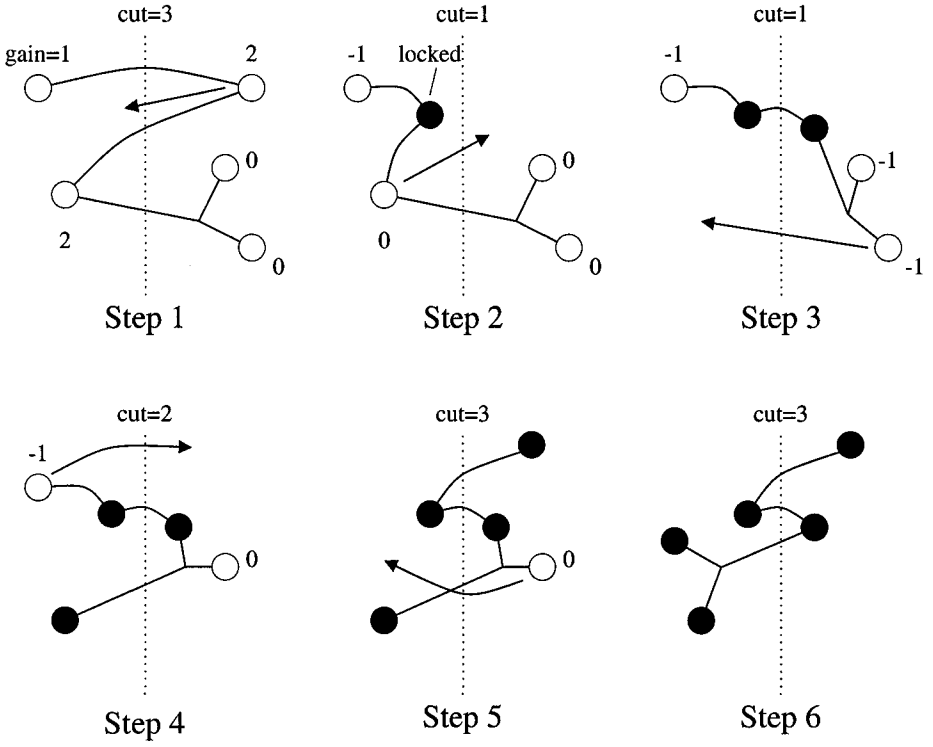


Figure 3.11: One run in the Fiduccia-Mattheyses algorithm.

used as the starting point for the next run where all nodes are unlocked again. The best partitioning is not likely to occur at the end of a run because, towards the end, the remaining gains are likely to be negative. The runs continue until no more improvement in cut width is achieved. Figure 3.11 illustrates one such run. In this example, the balance criterion is assumed to impose a 3:2 or 2:3 distribution. If there are ties during a step, they are arbitrarily broken, e.g. step 3, where two nodes in the right partition have gain -1. Both, steps 2 and 3, have a cut width of 1, and can be used as the starting point for the next run.

While the Fiduccia-Mattheyses algorithm is still the most well-known and used partitioning algorithm, a plethora of refinements and alternate algorithms have been published [AK93, AK94, AY95, CSZ94, CW91, CLS96, DD96, HK97, KAKS97, Kri84, KK89, MD97, Saa95, San89, SK96, SSB98, YCL94]. Currently the best and fastest efficient partitioning algorithm, called hMetis, was developed by Karypis et

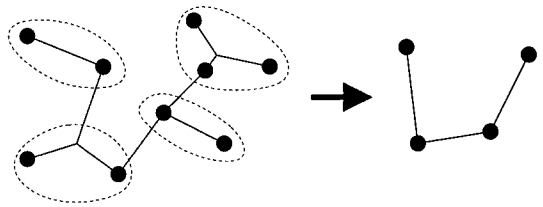


Figure 3.12: Clustering step in hMetis.

al. [KAKS97]. hMetis combines clustering with the Fiduccia-Mattheyses method. hMetis first generates a hierarchy of clusters. At each level of the hierarchy, edges (the nets in a circuit) are contracted such that each edge's member nodes form a new cluster node. Every node can only be part of one cluster node. Figure 3.12 shows an example of such a clustering step. This is repeated several times to form several levels of clustering until only a small hypergraph of clusters remains. This top-level hypergraph is randomly partitioned. Then, the top level clustering is undone, and the resulting second level hypergraph is partitioned using the Fiduccia-Mattheyses algorithm with the top level partition as a starting point. This procedure is continued until the base level is reached and partitioned. Details and variations of this implementation can be found in [KAKS97, KK98b, KK98a]. We will use hMetis in chapter 4 as the partitioner in a partitioning placer which we developed for comparison purposes.

In their 1991 placement survey, Shahookar and Mazumder [SM91] found partitioning placement to produce results that are in solution quality "second only to simulated annealing", but with substantially shorter run times. Since then, a class called force-directed placement has produced some good results which may even exceed partitioning placement in quality. Force-directed placements are the focus of the next section.

3.4 Force-Directed

Force-directed methods are inspired by attractive physical forces such as springs or gravitation acting on circuit elements. Their implementations vary substantially from analytical to iterative. In this section we will provide an overview of force-

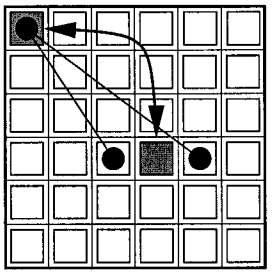


Figure 3.13: Node exchange in Goto's algorithm[Got81].

directed placement approaches.

Typically, a force such as one exerted by a spring attracts connected circuit cells toward each other. A set of repelling forces usually prevents all cells from occupying the same space. Implementation of the attractive and repulsive forces and the computation of a placement varies a great deal. Goto [Got81], for instance, places nodes on a grid as an initial solution and then iteratively exchanges nodes to place them closer to the median of its neighbours, cf. figure 3.13. Thus, in Goto's approach, the repulsive forces are implied by never allowing two cells to occupy the same grid position. An approach by Tia and Liu [TL93] has attractive forces acting on neighbours and, iteratively, nodes are moved proportionally to, and in direction of, the force acting on them. Unlike Goto's method, overlaps are not avoided by node-swapping, but rather by employing strong forces similar to Pauli's exclusion principle in physics, which forbids two particles to occupy the same space. No two nodes may inhabit the same grid positions. Consequently, as two nodes overlap, a repulsive force forces them apart.

An early analytical approach by Antreich et al. [AJK82] solves the placement problem by solving a system of equations describing the attractive and repulsive forces between nodes. Antreich leaves the final positions assignment to an interactive transformation from computed relative positions to grid positions on the chip.

Repulsive forces do not have to be intrinsic to each iteration. For instance, in an iterative approach, disguised as a neural net, by Chang and Hsiao [CH93] nodes are moved closer to their neighbours and then their positions are linearly scaled to map within the range of the chip area without regrad to local node densities.

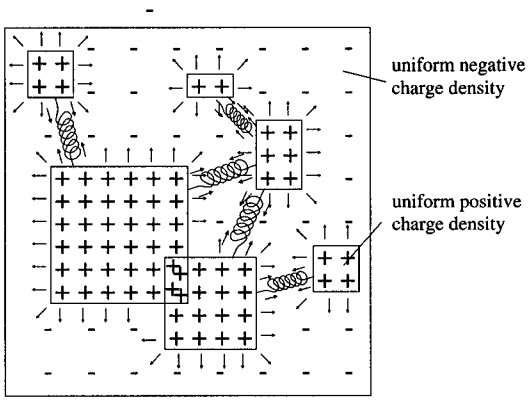
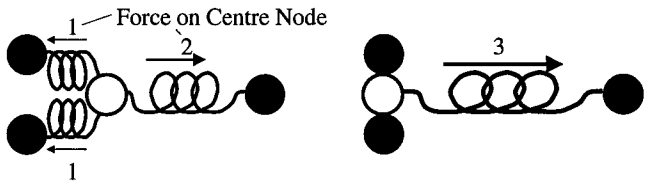


Figure 3.14: Spring-like and electrostatic field-like forces balance each other in the Eisenmann-Johannes[EJ98] method.

A two-phased approach recently patented by Koford [Kof98] has contracting forces iteratively drawing a node to the centre of its neighbours until a certain density is reached. This contraction phase is followed by an expansion phase in which cells repel each other. The contraction and expansion cycles are repeated until a termination criterion is met.

Unlike Koford's method, an algorithm by Eisenmann and Johannes [EJ98] combines contraction and expansion phases into one set of forces applied at the same time. This algorithm also combines both iterative and analytical approaches. The Eisenmann-Johannes algorithm claims the best results of any force-directed method to date. For this reason, we shall investigate their force model in a little more detail.

As for many force-directed methods, the attractive forces in the Eisenmann-Johannes method are spring like forces. Input/output pads, which are normally placed near the perimeter of a die, and other fixed connections as well as density-based repulsive forces counter the attractive forces. These repulsive forces mathematically mimic a static electric field. Consider the cells having a positive charge of equal charge density over all the cells' areas. Equivalently, imagine the empty chip surface having a constant negative charge density. The resulting electrostatic field determines the repulsive forces at any point on the chip. Figure 3.14 illustrates this set-up of attractive spring and repulsive electric-field-like forces. As a note to the physicist, we observe that the electrostatic-field-like forces acting on a cell do



Total Force on Centre Node:	0	3
Total Wire-Length:	4	3

Figure 3.15: Force on centre node is minimised, wire length is not.

not vary with size, or "charge" of the cell. The force depends solely on the cell's position, and thus the magnitude of the electrostatic **field**, and not the electrostatic **force**.

The Eisenmann-Johannes algorithm works iteratively. For each iteration, a system of equations is set-up and solved with the repulsive forces being held constant for the current iteration. Next, the repulsive forces are updated, and the next iteration is started. The iterations are stopped when the cells are sufficiently uniformly distributed. After the end of the iterations, some cells may still overlap. For this reason, and to force a standard cell layout, Eisenmann and Johannes employ a fine-grain post-placer called Domino [DJA94]. Domino uses a maximum-flow model to rearrange the cells on a local level to form standard cell rows. The published standard cell placement results of the Eisenmann-Johannes method rival or better the best published simulated annealing results.

Two problems are inherent to force-directed methods. Forces always act on pairs of nodes and thus, in the past, hypergraph edges usually were converted into cliques or similar constructs, thus altering the problem. Further, even an optimal solution to a force-directed placement problem may exhibit non-optimal wire length, cf. figure 3.15.

The time complexity of force-directed methods varies. Implementations that run in $\Theta(p)$ time, where p is the number of pins, are conceivable, e.g. iterative algorithms that use a constant number of iterations. Naturally, forcing linear run time may have an adverse affect on the solution quality. Numerical solutions that

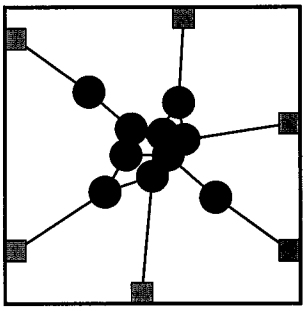


Figure 3.16: Example solution to quadratic placement equations: Nodes cluster in the centre.

solve systems of linear equations will require superlinear time, i.e., $\omega(N)$ where N is the number of nodes, since the problem matrices will be sparse but cannot expected to be band matrices.

Shahookar and Mazumder's survey [SM91] provides a good overview of force-directed methods. Force-directed methods are well respected in the graph drawing community[DBETT99] and a fundamental technique by Tutte [Tut60, Tut63] is also at the heart of a successful hybrid placement method called *quadratic placement.*

3.5 Quadratic Placement

Quadratic placement [KSJA91, PBS98, TK91, TKH88, Vyg97, WWM82] tries to combine the strengths of partitioning and force-based placement. A force-based method similar to the barycentric graph drawing method [DBETT99, Tut60, Tut63] is used to determine an initial relative placement of the nodes. These relative positions provide location information to create the initial partitions for a partitioner which splits the nodes. Then the process is repeated recursively on the two halves.

This method is called quadratic placement because the force-based method solves a system of linear equations that minimise the sum of the squares of the wire lengths between nodes $\sum_{\{a,b\}\in E}(x_a - x_b)^2$. Incidentally, this is same for spring-like forces, which are used in many force-based methods. Thus, force-directed methods could also lay claim to the name "quadratic placement", but they never did. As in the

barycentric graph drawing method, some nodes need to be fixed at the perimeter of the chip area to prevent the trivial solution where all nodes have the same position. In VLSI placement, the fixed nodes on the perimeter are called input/output pads. Solving the system of equations typically leads to a solution with many nodes clustered in the centre, cf. figure 3.16.

Using the squared wire-length objective allows for a linear system of equations to solve for the solution where the net force on all nodes is zero. These linear equations can be solved in $\omega(N)$ and $O(N^3)$ time. Partitioning in its simplest form requires $\Theta(N)$ time. Since we have $\log N$ levels of recursion, the overall time complexity of quadratic placement is $\omega(N \log N)$.

Even though quadratic placement is estimated to be utilised by many commercial tools, doubts have recently been raised about its efficiency. Alpert et al. [ACH+97] examined quadratic placement and compared the improvement in partitioning due to the numerical engine which solves the system of equations to a modern-generation partitioner. They found that modern partitioners do not benefit from the relative placement provided by a numerical engine. Since Alpert et al.'s paper was published, a still faster and better partitioner has been developed [KAKS97]. Thus, pure partitioning placers might outperform quadratic placement in the future.

3.6 Genetic Algorithms

A different class of placement algorithms deserves a brief note. Genetic placement algorithms are not as widely used as the other methodologies presented in this chapter. Standard benchmark results are also not easily available. Nevertheless, they form an interesting class of algorithms.

In genetic placement algorithms, a pool of random placements is generated. Probabilistically, placements with lower wire lengths are selected and paired to produce "offspring". These children are formed by crossing elements from both parents, and applying random mutations. To maintain the pool size, older and/or placements with long wire length are eliminated from the pool. After several generations, the best placement may serve as the solution. Genetic algorithms are similar to simulated annealing in that they work in a probabilistic way to evolve toward a better

solution. However genetic algorithms work on many solutions simultaneously, while simulated annealing constantly mutates only one solution. Thus, while genetic algorithms model biological evolution, simulated annealing behaves like an imperfect cloning factory. An overview of genetic VLSI design algorithms, including placement algorithms, is provided by Mazumder and Rudnick [MR99].

3.7 3-D Placement Approaches

Now that technology has made three-dimensional VLSI possible, cf. section 2.2, research into three-dimensional placement methods is emerging. Ohmura [Ohm98] has published a first three-dimensional placement algorithm. Given N circuit elements and M interconnections, Ohmura's algorithm runs in $O(N \cdot M)$ time. Iteratively, the algorithm interchanges pairs of nodes that promise the best wire-length improvement. Due to its high run-time complexity, it is unlikely Ohmura's algorithm will perform well for large circuits which are expected to best benefit from 3-D VLSI. The largest circuits the algorithm was tested on had 60 nodes, well short of the hundreds of thousands of nodes present in more complex modern circuits.

In a more practical approach, Leeser et al. suggested to use a three dimensional extension of quadrisection placement. As mentioned in section 3.3, this method splits a chip's volume recursively into eight subsections while simultaneously partitioning the circuit into eight subcircuits. However, no results were published.

Even though some theoretical work has been done on 3-D layouts [LR86] and routing [TW95], no practical placement algorithms have been published. In the following chapter, we provide two such algorithms, one newly developed force-directed placement algorithm for two and three dimensions, and a partition placement algorithm for 3-D VLSI, which we use to compare the results of our force-based methods. Some results of the latter algorithm were recently presented in [OS99].

3.8 Summary

Despite a wealth of published placement methods, it is difficult to pick a clear winner. Two main reasons are absence of comparable wire-length results and differences

in run-time complexity. Placement algorithms have been developed over many years with the earliest of the modern placement methods originating in 1977 [Bre77]. Over the years, electronic circuits that needed to be placed grew exponentially following Moore's Law and doubling in size approximately every 18 months. During the early 90s, a collection of benchmark circuits was made available under the ACM/SIGDA umbrella [Brg93]. The circuits in the suite range from a few hundred nodes to approximately 25,000 nodes. They used to be a valuable comparison platform through the mid-nineties. However, comparing modern placement algorithms using these smaller circuits is not effective for two reasons. Most modern algorithms yield the same or similar solutions to these small problems, and algorithms performing well on small circuits may have a large run-time or memory complexity that prevents them from yielding good results for larger circuits. In order to deal with modern circuit sizes, algorithms have to have a low run-time complexity which should not be significantly above linear-time. Thus, modern algorithms may perform poorly on smaller older circuits for which results have been published. However, the newer algorithms may be of much higher quality for larger circuits. Placement algorithms thus have to be compared as much qualitatively as quantitatively. It is clear, though, that a modern placement algorithm has to exhibit closer to linear-time behaviour than previous generation placers due to the ever increasing size of circuitry.

Chapter 4

Gravity

In this chapter, we present a new placement technique for minimum wire-length placements in two and three dimensions. Our placement method, which we named Gravity (**G**lobal **r**escaling and **a**ttraction between **v**ertices using **i**terative **t**ransformation **y**ields), is an iterative, force-directed technique which runs in linear time with respect to the circuit size. Gravity's low run-time complexity and efficient implementation make it a good placement algorithm for large circuits in three-dimensional and traditional two-dimensional VLSI.

The chapter is organized as follows. Section 4.1 explains the details of the placement algorithm in two dimensions. Section 4.2 provides a comparison of 2-D placement results measured against a generic partitioning placer using a set of standard benchmark circuits. We also include a comparison of standard cell placements for circuits for which standard cell placement results have been published. Section 4.3 explains how Gravity is extended to produce three-dimensional placements, and in section 4.4 we compare how our 3-D placement results compare against a generic 3-D partitioning placement algorithm.

4.1 Two Dimensional Algorithm

The Gravity placement algorithm has four simple steps. The first step is a random placement of nodes into the unit square. This is followed by a force-based iteration that moves neighbouring nodes closer together. After a number of these iterations,

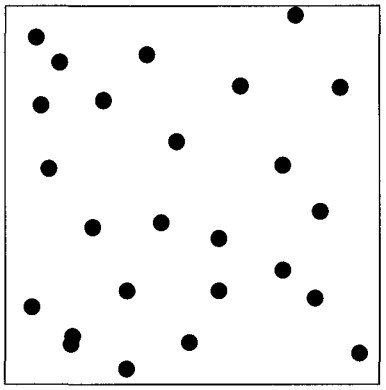


Figure 4.1: Step 1: Random initial placement of nodes.

node positions are rescaled in step three to re-achieve an approximately even node distribution. After a number of repetitions of steps two and three, step four determines the final placement through a linear-time recursive partitioning phase based on the nodes' computed coordinates.

In this implementation, we will only consider circuit elements of uniform area. We will point out how trivial modifications can be applied to accommodate arbitrary-area circuit nodes without loss of performance.

In this section, we will explain each step, and provide a complexity analysis.

4.1.1 Step 1: Initial Placement

Initially all circuit nodes are assigned a random initial position with a uniform distribution over the unit square. Edges, i.e., nets, connecting the nodes are ignored. Further, circuit elements may overlap. In fact, we will tolerate node overlap until the final placement step, cf. section 4.1.4. Figure 4.1 shows a random placement of nodes of a 25-element circuit.

4.1.2 Step 2: Force-Directed Step

A force-directed iteration computes a new node position for all nodes. Each node u's new position (x'_u, y'_u) is the weighted average of its own position, and the positions of its neighbours.

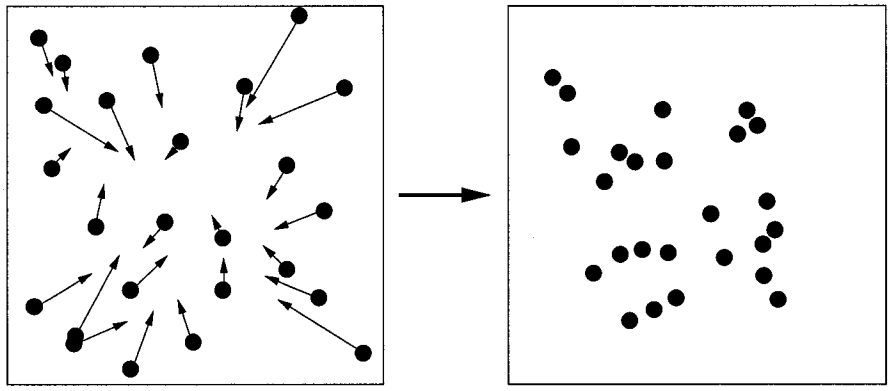


Figure 4.2: Step 2: Gravitate nodes.

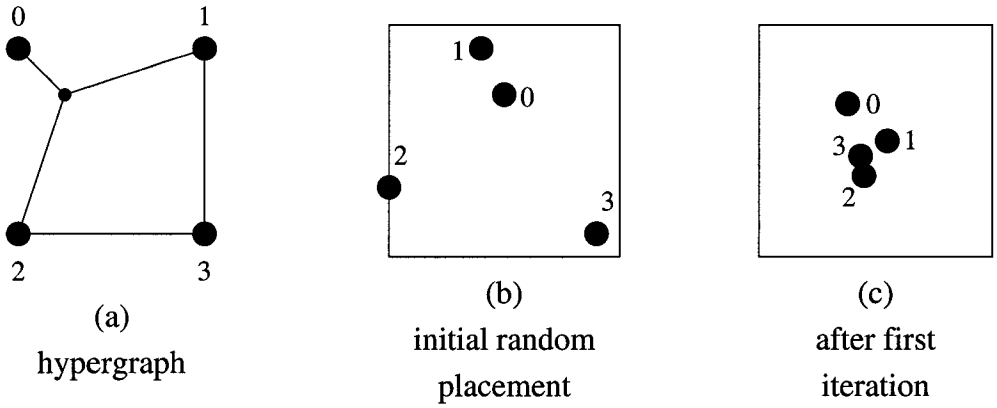
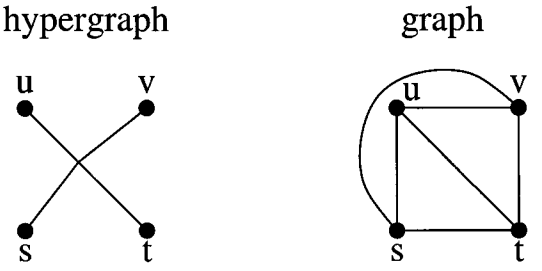


Figure 4.3: Example of a force step computation: The nodes of a hypergraph (a) are initially randomly placed (b). Then, node positions are updated according to equation (4.1) (c).



$$x'_u = \frac{x_u + (x_s + x_t + x_v)}{1+3} \qquad = \frac{x_u + x_s + x_t + x_v}{1+1+1+1}$$

hypergraph computation with $w_e = 1$ graph computation

Figure 4.4: Position computation according to equation (4.1) for a hypergraph and an equivalent graph in which the hypergraph edge has been replaced by a clique, and the weight w_e was fixed at 1.

$$x'_u = \frac{x_u + \sum_{e \in E_u} w_e \left[\left(\sum_{v \in e} x_v \right) - x_u \right]}{1 + \sum_{e \in E_u} w_e \left(|e| - 1 \right)} \tag{4.1}$$

where

$$\begin{aligned} E_u &= \text{set of edges incident to node } u, \\ |e| &= \text{cardinality of (or nodes in) edge } e, \\ w_e &= \frac{1}{\binom{|e|}{2}} = \frac{2}{|e|(|e|-1)} = \text{weight of edge } e. \end{aligned}$$

Equation (4.1) computes the average of the positions of node u, and all its neighbours as weighted by the edge weight w_e. In order to understand our arbitrary choice of the edge weight formula w_e, let us first assume we are placing a graph, and then discuss the implications for hypergraphs. If the hypergraph is a graph, i.e., for each edge e, $|e| = 2$, then all edge weights $w_e = 1$, and equation (4.1) reduces to

$$x'_u = \frac{x_u + \sum_{e \in E_u} \left[\left(\sum_{v \in e} x_v \right) - x_u \right]}{1 + \sum_{e \in E_u} (|e| - 1)}. \tag{4.2}$$

This is the average of all positions of node u and all its neighbours. In a hypergraph, an edge could have many nodes. In fact, if the weight was fixed at 1, then the position

computations for the hypergraph nodes would be equivalent to the computations of positions for a graph in which each hypergraph edge was replaced by a clique[1] of its member nodes, e.g. figure 4.4. In this case, every hypergraph edge e would be replaced by $\binom{|e|}{2}$ graph edges. A good placement algorithm for a graph where all hypergraph edges are replaced by cliques would try to minimize the projected overall wire length for all edges including all clique edges. However, in reality, a hypergraph edge only contributes to the overall wire length once, and not $\binom{|e|}{2}$ times. Thus, we scale the weight for each hypergraph edge by $1/\binom{|e|}{2}$. Ultimately, the choice of the edge weight is arbitrary, and can only be justified by the placement results it yields. In trial runs, the choice of $w_e = 1/\binom{|e|}{2}$ showed the best results, and choices of $w_e = 1/|e|$, or $w_e = 4/[|e|^2(|e|-1)]$, for instance, exhibited consistently worse placement results.

It should be observed that the cardinalities and weights of all edges have to be computed only once since they are constants. Further, the position sums $\sum_{v \in e} x_v$ have to be computed only once for each edge e at each iteration. Thus, this iteration step's execution time is linear in the number of pins $p = \sum_e |e|$.

Let us consider an example of one such force-directed computation. Let the random initial positions of the nodes in the hypergraph of figure 4.3a be $x_0 = (0.50, 0.70)$, $x_1 = (0.40, 0.90)$, $x_2 = (0.00, 0.30)$, and $x_3 = (0.90, 0.10)$, cf. figure 4.3b. The position update equations given by equation (4.1) simplify to

$$x_0' = \frac{x_0 + \frac{1}{3}(x_1 + x_2)}{1\frac{2}{3}} \tag{4.3}$$

$$= (0.38, 0.66),$$

$$x_1' = \frac{x_1 + \frac{1}{3}(x_0 + x_2) + x_3}{2\frac{2}{3}} \tag{4.4}$$

$$= (0.55, 0.50),$$

$$x_2' = \frac{x_2 + \frac{1}{3}(x_0 + x_1) + x_3}{2\frac{2}{3}} \tag{4.5}$$

$$= (0.45, 0.35), \text{ and}$$

$$x_3' = \frac{x_3 + x_2 + x_1}{3} \tag{4.6}$$

$$= (0.4\overline{3}, 0.4\overline{3}). \tag{4.7}$$

[1] A clique is a graph or subgraph in which every node is connected to every other node.

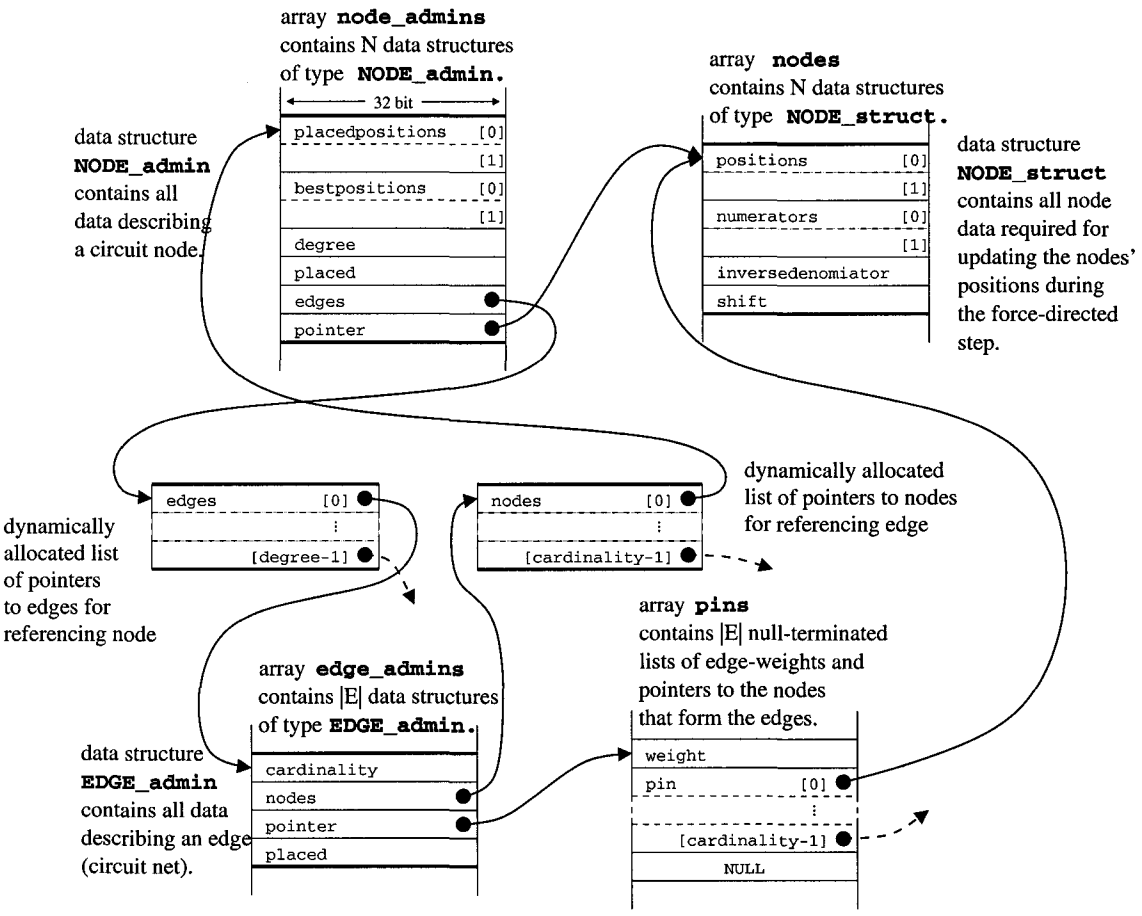


Figure 4.5: Diagram of the data structures representing the edges and nodes of a circuit hypergraph for 2-D placement

The updated positions are shown in figure 4.3c.

Gravity is implemented in C. In order to ensure linear time behaviour and efficient execution of the force-directed position computation, the data structures, computation loops, and arithmetic have to be designed carefully. We first examine the selection of data structures. Then, we discuss the construction of the computation loops, and finally we explain the implementation of the arithmetic.

4.1.2.1 Data Structures

The data structures of interest are those that describe the circuit hypergraph. These data structures have been optimized for speed of execution of the placement algorithm rather than aesthetics. Figure 4.5 shows a diagram of these data structures describing edges and nodes. Both nodes and edges of the hypergraph are represented by two data structures each. Nodes are described by `NODE_admin` and `NODE_struct` data structures. The `NODE_struct` data structure, cf. figure 4.5 top right, contains only the data required to compute a node's position according to equation (4.1). The `NODE_admin` data structure, cf. figure 4.5 top left, contains all other node data such as the position whose iteration yielded the best estimated wire length, a final placement position, the node's degree, a list of pointers to edges that are incident to the node, cf. figure 4.5 centre left, and a pointer to the corresponding `NODE_struct` data structure. Assuming there are N nodes in the hypergraph, there will be two arrays, `nodes` and `node_admins`, each containing N data structures of type `NODE_struct` and `NODE_admins`, respectively. Similarly, for each of the $|E|$ edges, there are two data structures, one of type `EDGE_admin`, and one null-terminated list. The null-terminated list, cf. figure 4.5 bottom right, contains all relevant data for the force-directed computation step. Stored at the head of each such list is the weight of the edge followed by a list of pointers to the edge's member nodes (pins). All other data is stored in the `EDGE_admin` data structure, cf. figure 4.5 bottom left, such as the edge's cardinality, a list of pointers to the member nodes' `NODE_admin` data structures, cf. figure 4.5 centre left, and a pointer to the null-terminated list of pins. The `EDGE_admin` data structures are contained in an $|E|$-element array called `edge_admins` and the $|E|$ pin lists are stored head-to-tail in an array called `pins`. By reducing the size of the data structures required for the force-step computation, more nodes can be kept in the computer's cache memory, thus speeding up execution. We will revisit caching optimization in section 4.1.2.3 where we discuss the initialization of these data structures. This choice of data structures yielded approximately 20% faster execution times than an earlier implementation in which all node and all edge data were stored in one data structure per node or edge.

For a circuit with N nodes (cells), $|E|$ edges (nets), and P pins, the total memory

1: for each edge e
2: position_sum $\leftarrow 0$
3: for each node u in e
4: position_sum $\leftarrow$ position_sum + position_u
5: for each node u in e
6: $\text{numerator}_u \leftarrow \text{numerator}_u + \text{weight}_e \cdot (\text{position\_sum} - \text{position}_u)$
7: for each node u
8: $\text{numerator}_u \leftarrow \text{numerator}_u + \text{position}_u$
9: $\text{position}_u \leftarrow \text{numerator}_u/\text{denominator}_u$
10: $\text{numerator}_u \leftarrow 0$

Figure 4.6: Pseudo code for the inner loops which update the node positions at each iteration.

requirement M for these data structures in the two-dimensional version of Gravity in a 32 bit architecture is

$$M = (14N + 6|E| + 3P) \cdot 32\text{bit} \quad (4.8)$$
$$= O(P). \quad (4.9)$$

4.1.2.2 Computation Loops

Equation (4.1) must be computed at every iteration of Gravity for every node. In order to avoid duplication of effort and guarantee efficient execution, the computation loops have to be selected with care. We first show how the selection of the loops yields linear run time. Then we show how the initialization and selection of the data structures serves in increasing the efficiency of the execution.

First, we note that the denominator in equation (4.1) is constant for each node and does not need to be re-computed for each iteration. Secondly, we note that the inner sum in the numerator, $\sum_{v \in e} x_v$, needs to be computed only once per iteration for every edge. With these observations in mind, we implement the computation of equation (4.1) in two loops as shown in figure 4.6. The first loop, which we refer to as the *edge loop* (steps 1-6), loops through each edge. For each edge, this loop

computes the inner sum $\sum_{v \in e} x_v$ (steps 2-4), and then computes one summand of the outer sum $\sum_{e \in E_u} w_e \left[\left(\sum_{v \in e} x_v \right) - x_u \right]$ for each of the edge's nodes and adds it to the corresponding node's numerator (steps 5-6). The time-complexity of this first loop is the number of pins (cumulative cardinality of all edges) as each pin is visited once in each of the two loops of steps 3-4 and 5-6. After the edge loop has been completed, the numerator for each node has the value of the outer sum $\sum_{e \in E_u} w_e \left[\left(\sum_{v \in e} x_v \right) - x_u \right]$. To complete the numerator, the position of each node x_u remains to be added. Following the edge loop is the *node loop* (steps 7-10). In the node loop, the numerator is completed by adding the node position (step 8), and the new node position is computed by dividing the numerator by the denominator, which has been precomputed during initialization, (step 9). Finally, the numerators are reset to 0 for the next iteration. The loop executes three constant-time operations for each node. Thus, its time complexity is linear with the number of nodes.

4.1.2.3 Depth First Search

In order to execute these loops efficiently, it is desirable to keep as much of the required data in the CPU registers and level-one cache as possible. A CPU's level-one cache keeps recently used data close to the execution units and can be accessed considerably faster than higher level caches or main memory. In the case of a Pentium II processor, which we used for comparing performance results, the level-one cache holds 16kB of data, and can be accessed with a latency of three clock-cycles and a throughput of one operand access per cycle [Int97].

We need to look at the actual implementation of the inner loops described by pseudo code in figure 4.6 to help us understand how we can improve the cache hit ratio. Figure 4.7 shows the unedited C code used in Gravity. In parenthesis next to the line numbers are the corresponding line numbers in the pseudo code. The added length in the C code is due to syntactical constructs and additional loops for the calculations for each dimension. The only data structures accessed in the inner loops that cannot be kept in registers are members of the `pins` array and the `nodes` array, cf. figure 4.5.

Improving the cache hit ratio can be achieved by frequently re-accessing recently used data, or by accessing data in consecutive memory locations. Accessing data

```
      /* compute edge position sums and update numerators */
1(1): for(pin=0,edge=0;edge<nonzero_nedges;edge++,pin++){
2:      unsigned long weight,positions_sum[DIMENSIONS];
3:      int first_pin;
4:      weight=(unsigned long)pins[pin];
5:      for(d=0;d<DIMENSIONS;d++)
6(2):     positions_sum[d]=0U;
7:      pin++;
8:      first_pin=pin;
9(3):   for(;pins[pin];pin++)
10:       for(d=0;d<DIMENSIONS;d++)
11(4):      positions_sum[d]+= ((NODE_struct *)(pins[pin]))->positions[d];
12(5):  for(pin=first_pin;pins[pin];pin++)
13:       for(d=0;d<DIMENSIONS;d++)
14(6):      ((NODE_struct *)(pins[pin]))->numerators[d]+=
15:           (weight*(positions_sum[d]
16:                    -((NODE_struct *)(pins[pin]))->positions[d]))>>BYTE;
17:   }

      /* compute new node positions */
18(7):for(i=0;i<nnodes;i++){
19:     NODE_struct *node=&nodes[i];
20:     for(d=0;d<DIMENSIONS;d++){
21(8):    node->numerators[d]+=node->positions[d];
22(9):    node->positions[d]=
23:         (((long long unsigned) (node->numerators[d]))
24:          *node->inversedenominator)>>(WORD+node->shift);
25(10):   node->numerators[d]=0U;
26:     }
27:   }
```

Figure 4.7: Unedited C source code for the inner loops that update the node positions at each iteration. In parenthesis next to the line numbers are the corresponding line numbers of the pseudo code of figure 4.6.

```
1: DFS(node u)
2:     if u is not placed
3:         place u into nodes array
4:         for each edge e connected to u
5:             if e is not placed
6:                 place e into pins array
7:                 for each node v in e
8:                     DFS(v)
9:                     set pointer to v in pins array
```

Figure 4.8: Depth first search algorithm for assigning elements in the **pins** and **nodes** arrays.

in consecutive memory locations improves the cache hit ratio because memory is copied to the cache in chunks, called cache lines. In case of the Pentium II, these chunks are of 32 byte length. In the edge loop (figure 4.6, steps 1-6, and figure 4.7, steps 1-17) the elements of the **pins** array are accessed consecutively. Similarly the node loop (figure 4.6, steps 7-10, and figure 4.7, steps 18-27) accesses the elements of the **nodes** array consecutively. These consecutive accesses ensure that the majority of these memory accesses are to level-one cache.

The remaining memory accesses are due to pointers in the **pins** array to elements in the **nodes** array, cf. figure 4.5, from within the edge loop (figure 4.7, lines 11 and 16). The order in which these nodes are accessed from the pins array is less predictable. To improve the cache hit-ratio, we would like to arrange the elements in the **nodes** and **pins** arrays such that consecutive references from the **pins** array to the **nodes** array reference mostly consecutive or recently used elements. One way of approaching this goal is to perform a depth first search on the hypergraph and to store nodes and edges in the **pins** and **nodes** array in the order in which they are encountered. The algorithm in figure 4.8 shows the procedure DFS used in Gravity for traversing the hypergraph. This procedure initializes the **pins** and **nodes** arrays. Figure 4.9 shows a simple example hypergraph and how DFS places the edges and nodes in the **pins** and **nodes** arrays. Using this depth-first search initialization we

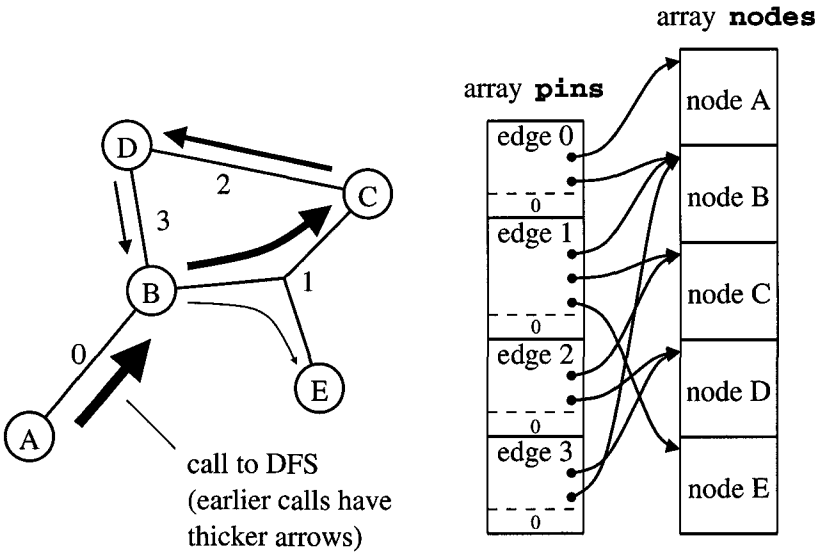


Figure 4.9: Depth first search of a hypergraph and resulting order of elements in the `pins` and `nodes` array.

noticed an approximate 2% speed improvement of the force-directed step loops for the largest circuits.

4.1.2.4 Arithmetic

Having addressed time-complexity, and caching issues, we can still take advantage of the superscalar, pipelined nature of current CPUs. The Pentium II, for example, has two integer pipelines, but only one floating point pipeline. Further, integer operations complete in fewer clock cycles. For this reason, Gravity uses fixed point arithmetic by way of integer operations. Different CPU architectures may have faster floating point units or more floating point pipelines such that floating point arithmetic may be more efficient. By using integer arithmetic, we observed an approximate 40% speed improvement for the node position computation loops on the Pentium II. Figure 4.10 provides an overview of the integer arithmetic and the involved bit-accounting. Node position coordinates fall within a $[0, 1)$ range. Gravity stores these coordinates with a 24-bit resolution inside a 32-bit integer. The low order bit of the 32-bit integer thus represents a value of 2^{-24}. When the

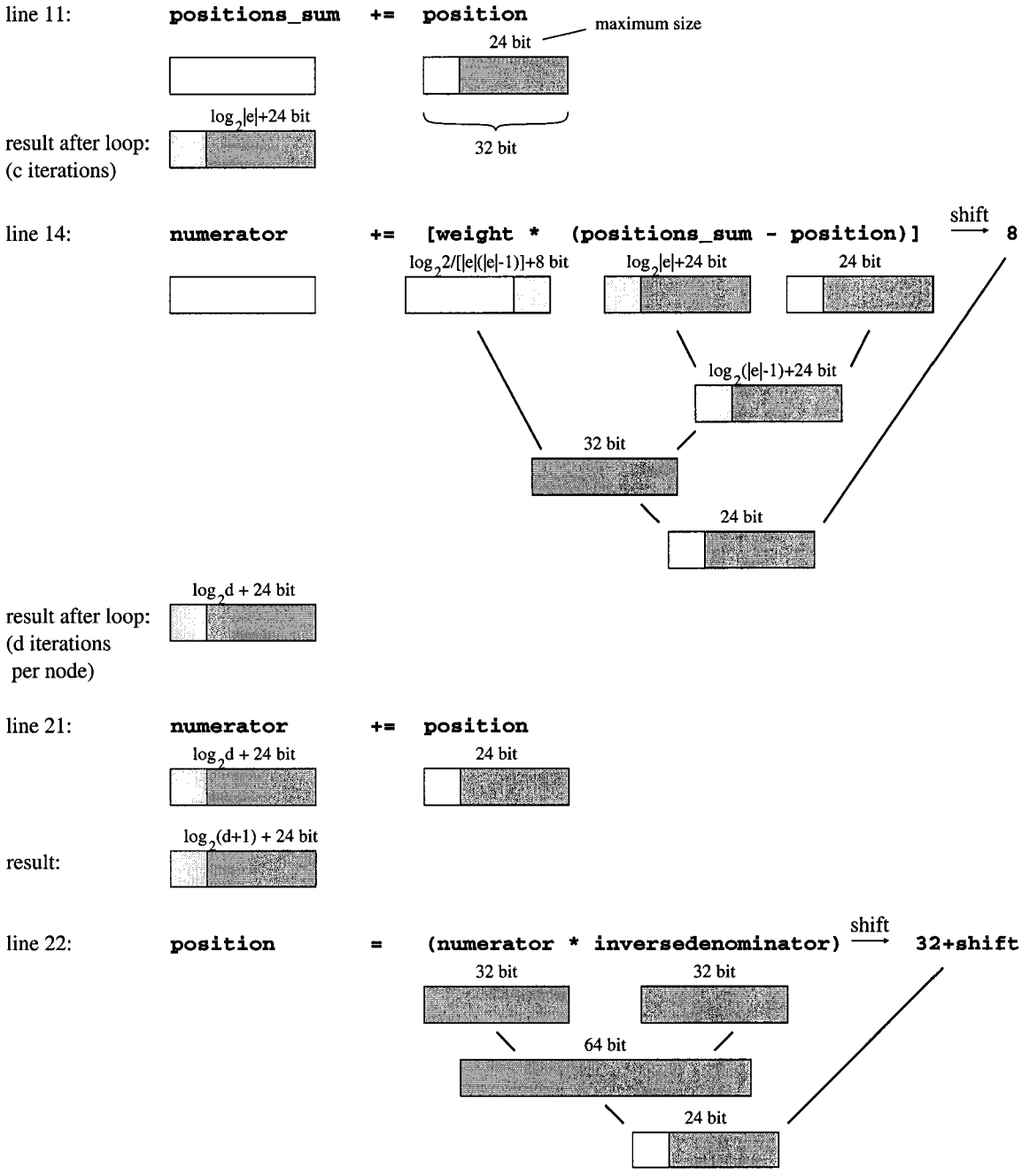


Figure 4.10: Fixed point arithmetic for solving equation (4.1) using integer operations. The corresponding line numbers of figure 4.7 are provided.

sum $\sum_{v \in e} x_v$ of all the c coordinates of an edge's c member nodes are computed in steps 9-11 of figure 4.7, the result has a size of up to $\log_2 |e| + 24$ bits. At line 14, a summand of the outer sum $\sum_{e \in E_u} w_e \left[\left(\sum_{v \in e} x_v \right) - x_u \right]$ is computed. First, a 24-bit position coordinate is subtracted from the previously computed inner sum, yielding a result of at most $\log_2(|e| - 1) + 24$ bits. This is multiplied by a weight of $8 + \log_2 \frac{2}{|e|(|e|-1)}$ bits. The size of the weight is arbitrarily set to 8 bits with the low order bit representing 2^{-8}. A higher resolution has not shown to result in improved results. The reason for this is that the weight is mainly a heuristic value. As long as the 8-bit weight is used consistently, slight deviations from the desired value of $\frac{2}{|e|(|e|-1)}$ are of no consequence. The value of the resulting product has at most 32 bits with a low order bit value of 2^{-32}. We right-shift this product by 8 bits. This 24-bit value is then added to the numerator which has a low order bit value of 2^{-24}. After adding the d summands of the outer sum for a node with degree d, the node's numerator has a value of at most $\log_2 d + 24$ bits. This value might exceed the alloted 32 bits for the numerator if the node had a degree of over 256. However, such nodes are extremely rare and comparison runs have shown that errors due to these nodes can safely be ignored. After adding the node's own position in step 21, the numerator's final value has at most $\log_2(d + 1) + 24$ bits which is by design no more than 32 bits. Finally, the node's new position is computed in step 22 by dividing the node's numerator by its denominator. The denominator is fixed for each node and can be precomputed during initialization. The inverse of this precomputed denominator is stored with a 32-bit resolution. The low-order bit of the inverse denominator represents a value of $2^{-\texttt{shift}}$. To complete the computation this 64-bit product has to be right shifted by 32+`shift` bits.

We call this a force based method because an attractive force between neighbours exerts a pull on each node, thus reducing the total wire length needed to embed the circuit. The overall effect is that all nodes are slowly pulled to the centre of the chip area, cf. figures 4.2 and 4.3. In contrast to most previous methods, we allow nodes to move freely, even if nodes overlap and occupy the same area. In the next step, we counter the pull toward the centre through rescaling.

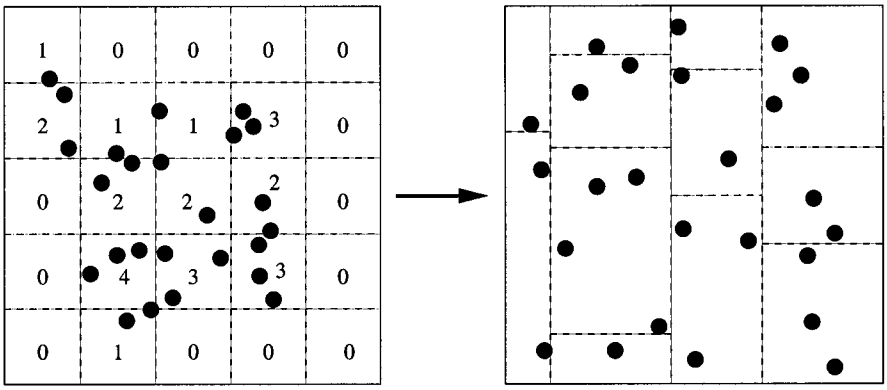


Figure 4.11: Step 3: Rescaling of nodes.

4.1.3 Step 3: Bucket-Rescaling

After a few iterations, it becomes necessary to counter the nodes' tendency to cluster near the centre. If the nodes are allowed to contract further, arithmetic precision will begin to introduce unacceptable errors. Another issue is that the more nodes cluster in the centre, the more the optimization problem changes: we wish to minimize the total wire length of the final embedding which does not allow a clustering of nodes; the final embedding resembles an even distribution of nodes.

The rescaling step performs a bucket mapping and rescaling operation. The unit square area is sliced into a grid of m by m buckets. Each bucket has a height and width of $1/m$. The nodes in each bucket are counted. Then, the number of nodes n_i in each column i of buckets determines the width w_i of the scaled column, $w_i = n_i/N$. Next, the number of nodes $n_{i,j}$ in each bucket i, j determines the height $h_{i,j}$ of each bucket, $h_{i,j} = n_{i,j}/n_i$. If nodes had different individual areas, the total areas in each bucket could be counted and bucket sizes could be adjusted to reflect their total contained area. The positions of all nodes are consequently adjusted to reflect the new heights and widths of their buckets. Figure 4.11 illustrates such a rescaling using a grid of 5 by 5 buckets. The number of nodes in each bucket are indicated.

This rescaling method may introduce discontinuities in the neighbourhoods of some nodes. However, empirical results have shown that smoother, computationally

more expensive, rescaling methods do not improve the overall performance of the algorithm. In practice, circuits placed by Gravity have at least several hundreds of nodes, and thus discontinuities at bucket borders are affect relatively fewer nodes than in smaller circuits.

This rescaling step is repeated until a certain uniformity criterion is met. We found a good threshold to be if the smallest bucket count is within 20% of the mean. That is, after a rescaling step, the unit square is divided into the same number of buckets as before the rescaling, and the number of nodes in each bucket are counted. Ideally, they should be the same for every bucket, however we will be satisfied if all bucket counts are within 20% of the mean and refrain from further rescaling of the buckets. The threshold value of 20% is an arbitrary value chosen because we believe it to yield a node distribution over the unit square that is relatively even but that does not disturb node neighbourhoods too much through repeated rescaling steps. Experiments with different values, e.g. 10% or 40%, have yielded either excessive rescaling iterations, and/or worse results.

Usually, not more than one repetition is required. At this point, the buckets are not resized and Gravity proceeds with further force-based iterations. In the case where significant node overlap occurs, sometimes multiple repetitions are ineffective. In these cases, after more than a dozen iterations, the number of buckets is reduced to from $m \times m$ to $(m-1) \times (m-1)$, and the node positions are jarred by a small random amount between $\frac{-1}{2\sqrt{N}}$ and $\frac{+1}{2\sqrt{N}}$. Thus a node's position may fall anywhere within a square of size $1/\sqrt{N} \times 1/\sqrt{N}$ about its original position. We chose this value because it corresponds to the spacing of N nodes in a $\sqrt{N} \times \sqrt{N}$ grid placing in the unit square. Since this kind of jarring of nodes is rare, the exact size of this random node movement is unimportant as long as it is relatively small but still separates nodes with identical positions. Such a reduction in the number of buckets happens typically once or twice for some circuits and never for most circuits. We found it best to perform a bucket-mapping for every ten force-based iterations.

4.1.3.1 Number of Buckets

We want to select the number of buckets such that node neighbourhoods at bucket borders are disturbed as little as possible while at the same time insuring that all

buckets are approximately equally filled after rescaling. Before a the rescaling, the node density will be higher in the centre and lower at the borders since nodes are drawn toward the centre by each force-directed position computation. Increasing the number of buckets will ensure that neighbouring buckets vary less in node density, thus disturbing node neighbourhoods less. On the other hand, having more buckets increases the probability that some buckets will fall outside the allowable deviation from the mean bucket size. If this is that case, another rescaling is required.

For a truly uniform distribution, the number of nodes in each of the $M = m \times m$ buckets is governed by a binomial distribution. The probability P_k of having k nodes out of a total of N nodes in a particular bucket is

$$P_k = \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}. \tag{4.10}$$

If we approximate the binomial distribution using a normal distribution, we can approximate the probability that all buckets are within a factor of ε of the mean number of nodes in a bucket N/m^2:

$$\begin{aligned} P_\varepsilon &= [P(|X - N/M| \leq \varepsilon N/M)]^M \\ &= \left[\sum_{i=(1-\varepsilon)N/M}^{(1+\varepsilon)N/M} \binom{N}{i} \left(\frac{1}{M}\right)^i \left(1 - \frac{1}{M}\right)^{N-i}\right]^M \qquad (4.11) \\ &\approx \left[\operatorname{erf}\left(\frac{\varepsilon N/M + 1/2}{\sqrt{2}\sqrt{N/M(1-1/M)}}\right)\right]^M. \qquad (4.12) \end{aligned}$$

For small values of N/M using a normal distribution to approximate a binomial distribution is not valid. However, the smallest value of N/M which we are interested in is 833/25, and the approximation is sufficiently close. For example, using $N = 833$, $M = 25$, and[2] $\varepsilon = 0.37$, equation (4.11) yields $P_{0.37} \approx 0.50$, and the approximation using a normal distribution in equation (4.12) yields $P_{0.37} \approx 0.55$.

Since Gravity is expected to create an approximately symmetrical node distribution peaked at the centre, we will consider only odd numbers of bucket columns and rows. For even numbers, the centre rows and columns would become virtually one double sized row or column. Further, on the average, we want to do no more

[2]This is $\varepsilon = 0.2$ multiplied by a fudge factor of 1.85 as discussed in the following paragraphs.

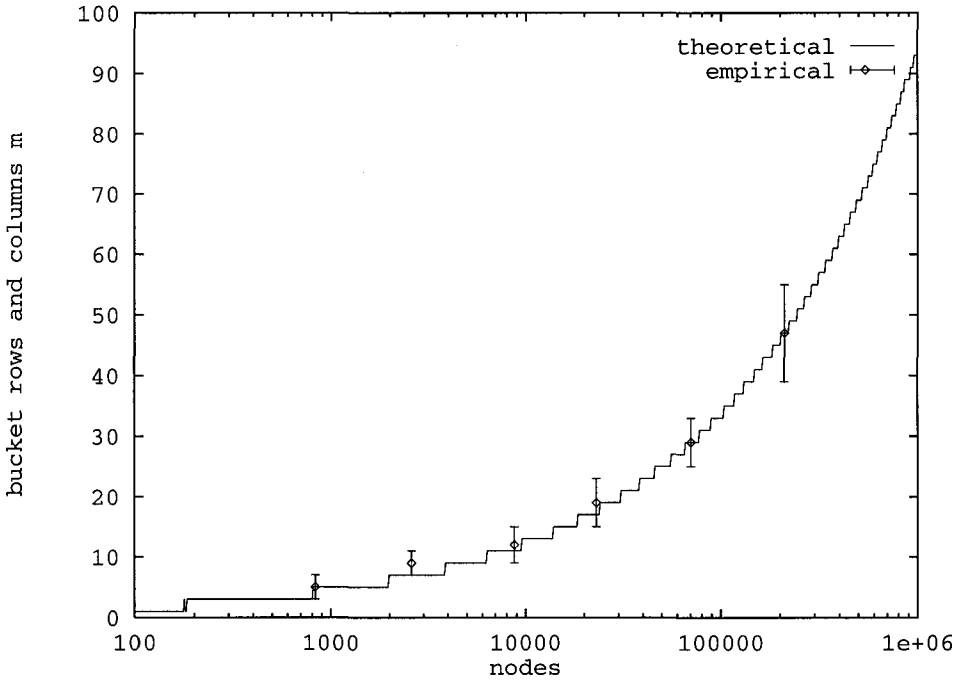


Figure 4.12: Computed and observed best number of buckets $m \times m$ for $\alpha = 1.85$.

than 2 rescaling steps to meet our uniformity constraint. With more consecutive rescalings, the algorithm would slow down, and more node neighbourhoods would be disrupted. Thus, we select the largest odd number of bucket rows and columns that guarantees that $P_\varepsilon \geq 0.5$:

$$m = \max\{m' : m' \text{ odd and } P_\varepsilon \geq 0.5\}. \tag{4.13}$$

Once the uniformity constraint ε is met, our rescaling step will have produced a node distribution that is more evenly distributed than a uniform distribution. By adjusting ε in the formula for P_ε by an empirically determined fudge factor α, we can take this evenness of the node distribution into account. We performed a number of runs for varying bucket numbers and circuit sizes. By selecting the number of buckets that contributes the shortest wire lengths while maintaining acceptable run times, we determined α to be approximately 1.85. Figure 4.12 shows that $\alpha = 1.85$ is a good match for the observed best number of buckets. Using $\varepsilon = 0.2$, the applied

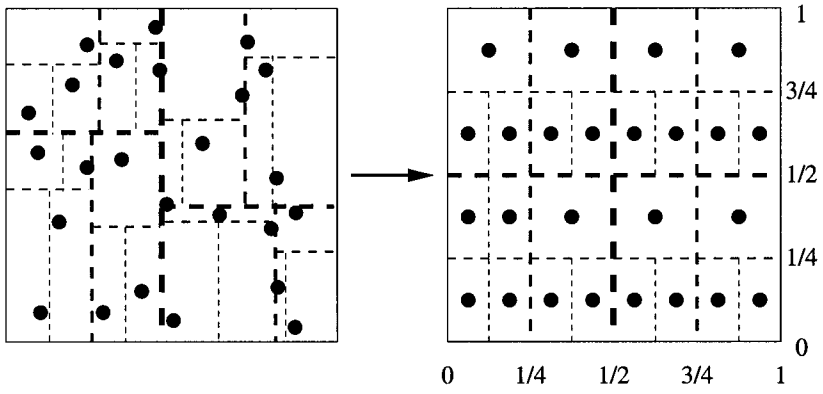


Figure 4.13: Step 4: Recursive partitioning (left) leads to final placement of nodes (right).

formula for the number of buckets $m \times m$ becomes:

$$m = \max\left\{ m' : m' \text{ odd and } M = m'^2 \text{ and } \left[\text{erf}\left(\frac{1.85\varepsilon N/M + 1/2}{\sqrt{2}\sqrt{N/M(1-1/M)}} \right) \right]^M \geq 0.5 \right\}. \quad (4.14)$$

Best results were observed if a rescaling step was performed after every 10 force-directed steps, and when the orientation of the bucket columns, alternates between the horizontal and vertical dimension. Then, after a preset number of iterations of steps two and three, typically several hundred to a few thousand, Gravity proceeds to the final placement step in which all node positions are adjusted to remove overlap.

4.1.4 Step 4: Final Placement

After a number of iterations, some nodes are expected to overlap partially or completely. This final placement step removes such an overlap. All nodes are recursively bipartitioned based on their positions. This is a simple and fast technique for arriving at a final node placement which can easily be compared to a generic partitioning placement algorithm. However, a more refined "fine-grain" placement method for computing a final placement such as DOMINO [DJA94] should be of benefit, par-

ticularly for standard cell placements, cf. sections 2.3 and 4.2.4. Unfortunately, DOMINO was not made available to us.

First, we sort all nodes by their x- and y-positions and store these two sorted node lists. At this point, the coordinates determined through numerous iterations of steps two and three, have served their purpose and can be forgotten. Then, we split the list of x-sorted nodes in half. The first half will be assigned coordinates in the left half of the chip area, and the second half gets coordinates in the right half. If the nodes were of different areas, the split could divide the nodes into two halves weighted by area. The list of y-sorted nodes has to be parsed completely and split according to the split in the x-sorted list. Then for each half, we perform a vertical split on the y-sorted list. We continue recursively until all nodes are placed. The recursion will reach a maximum depth of $\lceil \log_2 N \rceil$. Since one of the two x- and y-sorted lists have to be parsed at each step, the total execution time of this step is $O(N \log N)$. Figure 4.13 demonstrates this placement process for our 25-node circuit.

For any number of nodes other than $N = 2^{2i}$ for any integer i, there will be an uneven node distribution with different areas assigned to nodes, cf. figure 4.13(right), where the nodes on the top row occupy twice as much area as the nodes on the bottom row. It is certainly possible to use a final placement scheme that does not produce final placements with such uneven node distributions, cf. the grid splitting technique in section 4.3.2. However, this final placement method lets us easily compare our algorithm to placement methods build on recursive partitioning alone.

Typically, we perform a final placement step 25 times during a run, i.e., after every $k/25$ of k force-step iterations. This lets us sample the solution over the maximum number of iterations in an effort to eliminate noise and to help us avoid potential local maxima. The number of 25 final placement steps is another arbitrary parameter. With more final placements we observed an increase in the run time without achieving noticeably better placements. Less final placements did not reduce the run time significantly while placement quality started to suffer.

4.1.5 Complexity

The overall time-complexity of Gravity is $\Theta(p + N \log N)$, and the space complexity is the size of the input, i.e., $\Theta(p)$.

Time complexity is the sum of step one, random placement, i.e., $\Theta(N)$, a constant number k_1 of iterations of steps two and three, force based gravitation and bucket rescaling, i.e., $\Theta\left(k_1 \cdot (p + N)\right) = \Theta(p)$, and a constant number k_2 of final placement computations, i.e., $\Theta(k_2 N \log N)$. Due to the large but constant number of iterations, effectively, we observe a run-time complexity of $\Theta(p)$, as long as $N < 2^{k_1+k_2}$. The reader may question if the number of iterations k_1 is indeed a constant. The answer is "yes." For one, k_1 is preselected, and the algorithm will terminate after k_1 iteration no matter how well the placement job has been accomplished. Secondly, and more importantly, all test runs have shown that circuits converge to their best results after about 2000 iterations, independent of their size, provided the circuits are not trivial. Constant k_2 is the number of intermittent placements that are computed. This number is preset independently of the number of iterations chosen.

The run-time constant is dominated by the number of iterations k, and by how fast each iteration completes. In sections 4.1.2.1-4.1.2.4, we showed how the run time of Gravity is reduced by simplifying the data structures, simplifying the arithmetic, and improving cache hit ratios. The data structures accessed in the force step are stripped to the minimum information necessary. Thus, more nodes and edges can be kept in cache and fewer pointer dereferences have to be performed. Secondly, we use integer operations to simulate fixed point arithmetic. Finally, we arrange the node and edge data arrays according to the order of a depth first search before we start the iterations. This helps keeping more nodes and edges in level-one cache.

4.2 2-D Results

The two-dimensional version of Gravity is compared with a generic partitioning placement algorithm. Section 4.2.1 outlines the implementation of the partitioning placement algorithm. Section 4.2.2 provides an overview of the benchmark circuits used for comparing Gravity and the partitioning placer. The results of both algo-

rithms are compared in section 4.2.3. In section 4.2.4, we add a comparison with previously published standard cell placement results.

4.2.1 Recursive Partitioning Placement

Partitioning placement algorithms work almost exactly like our final placement step described in section 4.1.4. At every recursive iteration, the remaining chip area is split into two halves, and nodes are bipartitioned and assigned to one of the halves. Then we iterate recursively on each half until only one node remains. If we wanted to take node areas into account we would split the nodes based on equal cumulative node areas rather than equal node counts. The difference between our final placement and this partitioning placement algorithm is that during our final placement we bipartition on the basis of the computed node positions whereas with partitioning placement nodes are partitioned without prior knowledge. This lack of knowledge makes the computation of the partitioning considerably more complicated than the simple split-partitioning of Gravity's final placement step.

Most partitioners use a "min-cut" strategy that aims to reduce the number of edges that cross between the partitions. Even though min-cut is not directly related to placement wire length, the resulting placements obtained from a min-cut partitioning placements are quite good, cf. [Har86, SM91]. Good results are achieved because typically short wires mean most edges are contained in one partition, and vice versa. Recursive partitioning placement was discussed in more detail in section 3.3.

For comparison with Gravity, we use a hypergraph partitioning algorithm called hMetis 1.5[KAKS97, KK98a]: hMetis is on the average faster and achieves smaller cut widths than previous published algorithms. To eliminate any artificial performance disadvantage that hMetis suffers from partitioning very small subcircuits, we partition subcircuits with four or less nodes optimally without calling hMetis. These subcircuits contribute about 3/4 of the partitioning problems.

Circuit	Nodes	Nets	Pins	$\frac{Pins}{Node}$	$\frac{Pins}{Net}$	Circuit	Nodes	Nets	Pins	$\frac{Pins}{Node}$	$\frac{Pins}{Net}$
19ks	2844	3282	10547	3.71	3.21	s15850P	10470	10383	24712	2.36	2.38
avq.large	25178	25384	82751	3.29	3.26	s35932	18148	17828	48145	2.65	2.70
avq.small	21918	22124	76231	3.48	3.45	s38417	23949	23843	57613	2.41	2.42
baluP	801	735	2697	3.37	3.67	s38584	20995	20717	55203	2.63	2.66
biomedP	6514	5742	21040	3.23	3.66	s9234P	5866	5844	14065	2.40	2.41
golem3	103048	144949	338419	3.28	2.33	structP	1952	1920	5471	2.80	2.85
industry2	12637	13419	48158	3.81	3.59	t2	1663	1720	6134	3.69	3.57
industry3	15406	21923	65791	4.27	3.00	t3	1607	1618	5807	3.61	3.59
p1	833	902	2908	3.49	3.22	t4	1515	1658	5975	3.94	3.60
p2	3014	3029	11219	3.72	3.70	t5	2595	2750	10076	3.88	3.66
s13207P	8772	8651	20606	2.35	2.38	t6	1752	1641	6638	3.79	4.05

Table 4.1: The 1993 ACM/SIGDA benchmark circuits.

Circuit	Nodes	Nets	Pins	$\frac{Pins}{Node}$	$\frac{Pins}{Net}$	Circuit	Nodes	Nets	Pins	$\frac{Pins}{Node}$	$\frac{Pins}{Net}$
ibm01	12752	14111	50566	3.97	3.58	ibm10	69429	75196	297567	4.29	3.96
ibm02	19601	19584	81199	4.14	4.15	ibm11	70558	81454	280786	3.98	3.45
ibm03	23136	27401	93573	4.04	3.41	ibm12	71076	77240	317760	4.47	4.11
ibm04	27507	31970	105859	3.85	3.31	ibm13	84199	99666	357075	4.24	3.58
ibm05	29347	28446	126308	4.30	4.44	ibm14	147605	152772	546816	3.70	3.58
ibm06	32498	34826	128182	3.94	3.68	ibm15	161570	186608	715823	4.43	3.84
ibm07	45926	48117	175639	3.82	3.65	ibm16	183484	190048	778823	4.24	4.10
ibm08	51309	50513	204890	3.99	4.06	ibm17	185495	189581	860036	4.64	4.54
ibm09	53395	60902	222088	4.16	3.65	ibm18	210613	201920	819697	3.89	4.06

Table 4.2: The 1998 ISPD benchmark circuits.

4.2.2 Benchmark Circuits

Placement methods are commonly compared with the aid of benchmark circuits. We compare the performance of Gravity using two benchmark circuit suites, the ACM/SIGDA suite [Brg93] and the ISPD98 suite [Alp98].

The ACM/SIGDA suite is well established. Circuits in this suite have been used since the late 80s to compare placement and partitioning algorithms. This suite's main advantage is that standard cell placement results have been published. The biggest disadvantage is that this suite is relatively old, and its circuits are small in comparison with today's circuits. Even though its largest circuit has approximately 100,000 nodes, the remaining circuits only cover the dynamic range from 800 to 25,000 nodes. Table 4.1 shows the basic circuit characteristics of the ACM/SIGDA suite.

The newer ISPD98 circuit suite consists of circuits covering the dynamic range from 12,000 to 210,000 nodes. The circuits are typical integrated circuits designed at IBM representing "bus arbitrators, bus bridge chips, memory and PCI bus interfaces, communication adaptors, memory controls, processors, and graphics adaptors" [Alp98]. The circuit data provided with this suite are the cells, the nets, and a scalar number to indicate cell area. All other electrical information as well as cell dimensions have been stripped. Even though this suite is not as established as the ACM/SIGDA suite, its circuits are needed to measure the placement performance on larger modern circuits. Table 4.2 lists the characteristics of the ISPD98 circuits.

The ACM/SIGDA and ISPD98 are the only published and commonly used circuits suites of this kind. Although the ISPD98 suite is more often used for partitioning comparisons, it is important that we include placement results for this suite because of its larger sizes and circuit characteristics. The larger sized circuits are essential as placement algorithms have to cope with ever increasing circuit sizes as observed in Moore's Law. Further, the circuit characteristics should reflect modern circuit design. Among the characteristics that can be objectively measured are the distributions of net lengths. The number of pins per net, and pins per node, cf. tables 4.1 and 4.2, are relatively consistent throughout the circuits of both suites. However, despite the variety of circuit types, the circuits of the newer ISPD98 suite have a much more even distribution of nets. Figure 4.14 shows the distribution

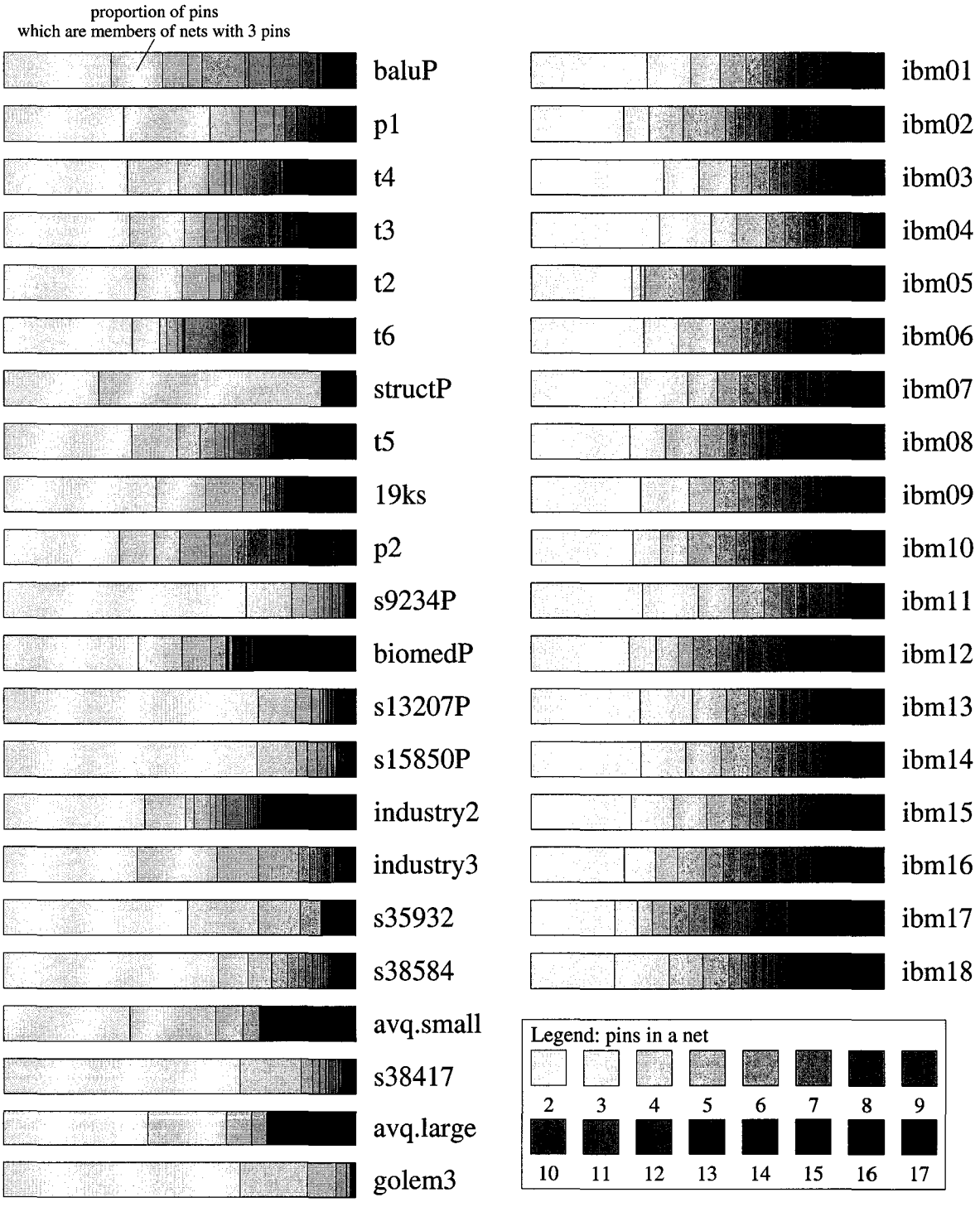


Figure 4.14: Distribution of pins sorted by net lengths for the ACM/SIGDA circuits (left) and the ISPD98 circuits (right). Smaller circuits are at the top, larger circuits at the bottom.

of pins sorted by net-lengths. While the number of pins belonging to nets of different lengths is approximately equally distributed for all circuits of the ISPD98 suite, the distributions of pins in the ACM/SIGDA suite vary considerably between circuits. This may be an indication that modern circuit designs uses a more consistent design methodologies, i.e., methodologies may have evolved to the point where large variances are suppressed. Alternatively, it may indicate that the circuits in the ACM/SIGDA suite do not mirror the bulk of circuits designed nowadays, perhaps because modern integrated circuits are composed of many of the older smaller circuits, thus averaging out variances of smaller circuits.

4.2.3 Result Comparison

To evaluate the performance of Gravity, we compared it against recursive partitioning using todays fastest and best bipartitioner hMetis [KAKS97]. Since both placement algorithms, Gravity and hMetis recursive partitioning placement, are randomized algorithms, we performed 10 runs for each circuit. The wire lengths were estimated using semi-perimeter bounding box approximations of the lengths of rectilinear Steiner trees [Han66]. This approximation, which simply adds the horizontal and vertical spread of nodes in a net, is easy to compute and commonly used. It gives exact results for nets with two or three pins which form the large majority of all nets.

For every circuit, we ran the hMetis-based algorithm 10 times. Similarly, we ran Gravity 10 times for each circuit. We repeated these 10 runs for Gravity using 500, 1000, and 2000 iterations of step 2, cf. section 4.1.2. In every case, we performed the rescaling step 3, cf. section 4.1.3, after every ten iteration of step 2. For the rescaling step, we used buckets as determined by equation 4.14 until all bucket counts were within 20% of the average bucket count.

As can be seen in tables 4.3 and 4.4, Gravity achieves competitive results even with only 500 iterations. On the ACM/SIGDA suite, Gravity performs 4% better than hMetis, and on the ISPD98 suite Gravity even outperforms hMetis on the average by more than 8%. Using 2000 iterations, the wire-length advantage of Gravity increases to over 12%, on the average.

During our experiments, we found that wire length decreases steadily up to

Circuit	hMetis	500 iterations			1000 iterations			2000 iterations		
	length	length	% change	speed-up	length	% change	speed-up	length	% change	speed-up
19ks	450	355	-21.17	21.04	350	-22.30	12.26	343	-23.68	6.61
avq.large	1089	1176	8.00	15.42	1002	-7.99	8.43	915	-15.94	4.38
avq.small	1054	1071	1.63	16.13	955	-9.39	8.85	869	-17.54	4.57
baluP	167	145	-13.14	20.87	143	-14.47	11.35	140	-16.16	5.74
biomedP	514	455	-11.55	16.36	450	-12.52	8.99	441	-14.27	4.68
golem3	4720	6944	47.11	12.90	6306	33.60	7.60	5888	24.74	4.17
industry2	1278	1297	1.48	14.51	1288	0.74	8.51	1286	0.63	4.85
industry3	2389	2088	-12.61	16.30	2030	-15.05	9.05	2004	-16.13	4.66
p1	214	168	-21.33	17.35	166	-22.33	9.41	166	-22.51	5.31
p2	599	497	-17.01	17.77	483	-19.25	10.28	492	-17.86	5.83
s13207P	401	381	-4.97	15.37	360	-10.17	8.49	351	-12.39	4.47
s15850P	443	434	-2.13	15.21	405	-8.74	8.18	394	-11.06	4.37
s35932	579	746	28.87	15.52	628	8.51	8.67	562	-3.00	4.47
s38417	684	872	27.56	13.29	755	10.36	6.79	709	3.72	3.38
s38584	757	889	17.45	13.20	815	7.61	7.10	790	4.35	3.77
s9234P	350	327	-6.60	17.00	310	-11.33	9.42	305	-12.93	5.05
structP	228	187	-18.03	20.75	181	-20.81	12.03	172	-24.69	6.69
t2	333	255	-23.49	19.05	253	-24.03	10.79	251	-24.48	5.80
t3	304	249	-18.06	18.75	237	-22.06	10.52	238	-21.87	5.66
t4	292	239	-18.23	18.14	238	-18.39	10.16	236	-19.15	5.64
t5	390	328	-15.74	17.33	328	-15.97	10.35	331	-15.17	5.48
t6	304	254	-16.56	17.65	258	-15.40	10.85	254	-16.42	5.76
Average			-4.02	16.81		-9.52	9.46		-12.36	5.06

Table 4.3: Wire-length comparison of Gravity vs. hMetis for ACM/SIGDA benchmarks. Average over 10 runs.

Circuit	hMetis	500 iterations			1000 iterations			2000 iterations		
	length	length	% change	speed-up	length	% change	speed-up	length	% change	speed-up
ibm01	1545	1331	-13.83	18.51	1305	-15.51	10.29	1293	-16.29	5.39
ibm02	3176	2847	-10.34	19.74	2771	-12.73	11.22	2717	-14.43	5.98
ibm03	3557	3178	-10.64	17.48	3105	-12.71	9.72	3083	-13.31	5.17
ibm04	4020	3436	-14.54	15.34	3291	-18.14	8.55	3279	-18.45	4.58
ibm05	5786	4719	-18.45	19.37	4656	-19.53	11.12	4675	-19.21	5.97
ibm06	4224	3472	-17.80	17.01	3355	-20.55	9.56	3306	-21.73	5.05
ibm07	5230	4506	-13.86	17.03	4320	-17.40	9.50	4305	-17.69	5.02
ibm08	5533	4887	-11.69	16.87	4796	-13.32	9.91	4688	-15.27	5.58
ibm09	5046	4859	-3.70	13.31	4712	-6.61	7.39	4609	-8.67	3.92
ibm10	7292	6546	-10.23	17.67	6282	-13.85	9.77	6180	-15.25	5.23
ibm11	6727	6288	-6.53	15.28	5968	-11.29	8.35	5863	-12.84	4.42
ibm12	9168	8236	-10.17	17.21	8064	-12.04	10.35	8026	-12.46	5.76
ibm13	7464	6929	-7.17	16.66	6708	-10.14	9.12	6668	-10.67	4.82
ibm14	10784	10524	-2.41	17.62	9800	-9.12	9.85	9640	-10.60	5.18
ibm15	13095	13212	0.89	17.73	12494	-4.59	9.76	12295	-6.11	5.13
ibm16	13557	13369	-1.39	18.68	13034	-3.86	10.44	12524	-7.62	5.43
ibm17	17033	16631	-2.36	19.95	15681	-7.94	11.27	15520	-8.88	6.14
ibm18	12283	13461	9.59	18.90	12809	4.28	10.53	12356	0.59	5.61
Average			-8.03	17.46		-11.39	9.82		-12.72	5.24

Table 4.4: Wire-length comparison of Gravity vs. hMetis for ISPD98 benchmarks. Average over 10 runs.

1000 iterations, but only marginally above 1000 iterations. This observation is reflected in the provided tables. We feel 1000-iteration Gravity produces the best cost-performance trade-off.

In appendix A, we offer detailed comparison tables between hMetis and Gravity. We note that the standard deviations of the individual runs are approximately equally small for both methods. The relative smallness of the standard deviations shows that both methods are very stable.

When we compare the run times of Gravity and the hMetis placer on a 300MHz Pentium II system with 100MHz memory bus, 512kB level-two cache and 16kB levek-one data cache, Gravity is considerably faster. For 500 iterations, Gravity

requires only about 1/17 of the time that the hMetis placer requires. With 2000 iterations, Gravity still requires less than 1/5 the time of the hMetis placer.

4.2.4 Standard Cell Placement Comparison

This section outlines the performance of a first implementation of standard cell placement using the Gravity placement algorithm. Unlike in the placements computed above where all nodes are of unit size, in standard cell placements nodes vary in size. We compare our algorithm, Gravity, to the best published algorithm for standard cell placement [EJ98]. The integrated circuit standard cells have variable sizes, and the total integrated circuit area depends on the cell area and the area of the wiring.

4.2.4.1 Standard Cell Placement

As presented in section 2.3, standard cell layout is a popular integrated circuit design method. Standard cell layouts consist of multiple horizontal rows of cells. Each cell has its own width, and all cells have the same height. Between rows of cells is space for horizontal interconnections, and vertical interconnections cross cells at predefined *feed-throughs*. These interconnections are called *nets*. Each net connects two or more cells. These connections are often called pins. Figure 2.4 on page 35 shows a standard cell placement.

In the past, placement papers used standard cell placements of benchmark circuits to compare their performance. Since placement algorithms do not perform routing, wire lengths are estimated using the semi-perimeter bounding box method [SM91], and routing space between rows, i.e., the row separation, is fixed at one row-height. We will compare the 2-D version of our algorithm versus the leading published placement algorithm by Eisenmann and Johannes [EJ98]. This Eisenmann-Johannes algorithm solves the "coarse-grain" placement problem which computes relative node positions which can be located anywhere on the placement area. This "coarse-grain" placement is comparable to Gravity before its final placement step. For the fine-grain standard cell layout, they use the existing DOMINO algorithm[DJA94]. Eisenmann and Johannes published their wire-length results for

Circuit	Nodes	Nets	Pins	$\frac{pins}{nodes}$	$\frac{pins}{nets}$	max. p such that less than 90% of nets have p or less pins, i.e., $\max\{p : \lvert\{e : \lvert e\rvert \leq p\}\rvert < 0.9\lvert E\rvert\}$
avq.large	25178	25384	82751	3.29	3.26	2
avq.small	21918	22124	76231	3.48	3.45	3
biomedP	6514	5742	21040	3.23	3.66	3
industry2	12637	13419	48158	3.81	3.59	4
industry3	15406	21923	65791	4.27	3.00	4
p1	833	902	2908	3.49	3.22	4
p2	3014	3029	11219	3.72	3.70	6
structP	1952	1920	5471	2.80	2.85	2

Table 4.5: 8 circuits from the ACM/SIGDA benchmark suite.

	Eisenmann-Johannes		Gravity				
Circuit	length (m)	Time (s)	Average Length (m)	% change	Minimum Length (m)	% change	Speedup
avq.large	5.38	1487.83	7.85	45.93	7.56	40.48	3.79
avq.small	4.91	1275.38	7.11	44.79	6.79	38.28	3.80
biomedP	1.78	208.05	2.05	15.26	1.97	10.74	2.53
industry2	14.60	939.87	15.12	3.56	14.45	-1.04	5.22
industry3	45.10	1175.76	45.59	1.09	44.46	-1.42	4.44
p1	0.87	27.10	0.81	-6.51	0.79	-9.07	2.96
p2	3.72	111.35	3.62	-2.65	3.52	-5.38	3.15
structP	0.34	29.30	0.35	2.84	0.34	0.13	1.77
Average				13.04		9.09	3.46
Average for $p > 3$ in Table 4.5				-1.13		-4.23	3.94

Table 4.6: Gravity standard cell placements vs. Eisenmann-Johannes.

8 circuits from the 1993 ACM/SIGDA benchmark circuit suite [Brg93]. Table 4.5 shows these circuit characteristics.

4.2.4.2 Results

The current implementation of Gravity for standard cell layouts does not yet include a fine grain final placement step. In order to compensate, we increased the run time of Gravity substantially by letting it run longer. Nevertheless, Gravity's run time is still approximately 3.5 times faster[3] than for Eisenmann-Johannes while the wire-length results are approximately 13% worse on the average. Table 4.6 shows a comparison of wire-length results and run times. However, Gravity outperforms Eisenmann-Johannes on the circuits which do not suffer a heavy short edge bias.

The last column in table 4.5 reflects the distribution of number of standard cells per wire (i.e., pins per net)[4]. For example, in the top three circuits, avq.large, avq.small, and biomed, approximately 90% of all wires interconnect 3 or fewer standard cells. Gravity does not perform very well for these circuits. In the next four circuits (industry2, industry3, p1, p2), the nets with at least four pins do not add up to 90% of the nets, which means that a larger number of pins are part of longer nets. Visually, this difference in pin distribution can be observed in figure 4.14 on page 89. For industry2, industry3, p1, and p2, the transitions of shading from light to dark is more gradual than for the circuits avq.large, avq.small, and biomed. Such a smoother transition of shading in figure 4.14 corresponds to a more balanced distribution of pins over nets of different sizes. With their smoother shading transition, circuits industry1, industry2, p1, and p2 are closer in their pin-distribution characteristics to the ISPD98 circuits, cf. figure 4.14 (right). Since Gravity produces competitive results for these circuits (cf. average 1.1% improvement on bottom row in table 4.6), it seems possible that Gravity may outperform Eisenmann-Johannes on the larger ISPD98 circuits. We expect a further improvement in Gravity's standard cell placements if a good "fine-grain" placer is employed. Unfortunately, no standard cell placement results have been published for the ISPD98 suite, and so the anticipated standard cell placement performance of Gravity remains conjecture.

[3]The Eisenmann-Johannes run times have been scaled from a DEC Alpha 250/4-266 to compare run times our Pentium II-300 reference platform.

[4]This is the maximum number p such that all nets with p or less pins combined do not add up to 90% of all nets E, i.e., $\max\{p : |\{e : e \in E \text{ and } |e| \leq p\}| < 0.9|E|\}$.

4.3 Three Dimensional Algorithm

As integrated circuits become more complex, utilization of the third dimension is becoming a more realistic implementation solution. In section 2.2 on page 31 we have shown how recent work has resulted in near uniform three dimensional field programmable gate arrays (FPGAs). However, cell placement for three dimensional integration is still in its infancy, cf. section 3.7 on page 63. If 3-D integration is to help the implementation of very large circuits, efficient placement and design tools are required. In this section, we will detail the differences between the three dimensional Gravity algorithm and the two-dimensional version presented in section 4.1.

The three dimensional version of Gravity varies from its two dimensional implementation by placement geometry, the size of the data structures, the selection of rescaling buckets, and by the final placement step.

The most obvious difference is the placement geometry. Where 2-D Gravity internally used a unit square to track node positions, 3-D Gravity uses a unit cube. Thus, the random initial placement assigns nodes to coordinates in the unit cube.

The data structures shown in Figure 4.5 are extended to hold the three-dimensional data by adding an additional `placedpositions[2]` field and `bestpositions[2]` field to the `NODE_admin` data structure, and an additional `positions[2]` and `numerator[2]` field to the `NODE_struct` data structure.

4.3.1 Bucket Rescaling

During the bucket rescaling step in the 2-D algorithm, cf. section 4.1.3, the unit square placement area would be sliced into $m \times m$ square buckets, with m being odd. In the 3-D case, our placement "area" becomes a unit cube. The cube is sliced into box-like buckets. Then, the thickness of slices of buckets is rescaled to reflect the number of nodes within, e.g. see figure 4.15. Within each slice the columns of buckets and the individual buckets in each column are rescaled as in the two-dimensional rescaling step, cf. section 4.1.3. The dimension along which the bucket slices are sliced rotates between the x, y, and z-axes for each rescaling step as not to interrupt node neighbourhoods in a systematic fashion. The natural extension of

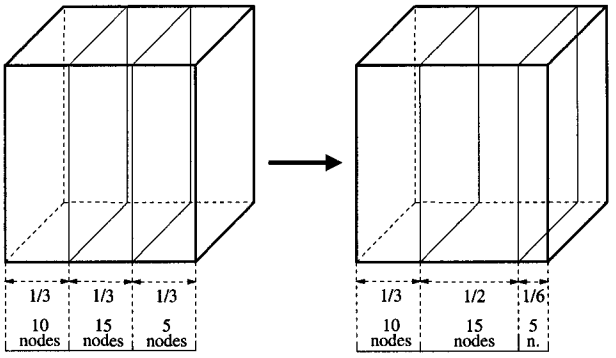


Figure 4.15: Rescaling of a slice of buckets in the 3-D rescaling step.

the 2-D method would suggest to carve the volume into $m \times m \times m$ cubic buckets. However, reducing m to the next lowest odd number $m-2$ changes the total number of buckets by $6m^2 - 12m + 8$. Experience has shown that this selection of buckets is too coarse. Hence it becomes difficult to achieve a good tradeoff between gaining an even node distribution by using a larger number of buckets, and not have to rescale multiple times by choosing less buckets, cf. section 4.1.3.1. For this reason, we allow the number of buckets in each dimension, m_1, m_2, and m_3, to vary from the others by at most 2. As in the 2-D case, given M buckets, the probability P_ε for a uniform distribution to have no bucket size varying by more than ε from the mean is given by equation (4.12). Since we expect the node distribution to be symmetrical, we choose an odd number of buckets m_i in each dimension i for a total number of $M = m_1 \times m_2 \times m_3$ buckets. Starting with a $3 \times 3 \times 3$ grid of buckets, we increase each m_i in turn by 2 as long as $P_\varepsilon \geq 50\%$. This ensures that the expected number of rescaling iterations is no more than 2. The actual node distribution after a rescaling step outlined in section 4.1.3 is not a true uniform distribution. In fact, the resulting distribution exhibits a smaller variance in bucket counts as is the aim of our rescaling technique. By empirical observation, we determined that ε should

be scaled by 1.85. Thus, the actual number of buckets used by Gravity, M' is

$$\begin{aligned} M' = \max\{M : \; & P_{1.85\varepsilon} \geq 0.5 \\ & \text{and } M = m_1 m_2 m_3 \\ & \text{and } m_1, m_2, m_3 \text{ odd} \\ & \text{and } \max\{m_1, m_2, m_3\} - \min\{m_1, m_2, m_3\} \leq 2\}. \end{aligned} \quad (4.15)$$

Observation has shown that choosing the initial number of buckets according to (4.15) leads to very good placement results for all benchmark circuits ranging from 833 nodes for `p1` to 210,613 nodes for `ibm18`. Occasionally, degenerate circuit features can still lead to excessive rescaling iterations. In the case where more than 12 rescaling iterations occur, we reduce m_1, m_2, or m_3 in turn by 2 and move the node positions by a random amount between $-\frac{1}{2\sqrt{N}}$ and $+\frac{1}{2\sqrt{N}}$.

4.3.2 Final Placement

In two dimensions, the final placement was determined by recursively splitting the nodes and the unit square in halves in order to facilitate comparison with a recursive partitioning placement algorithm, cf. sections 3.3 and 4.1.4.

Figure 4.13 showed how this could lead to non-homogeneous grids when the number of nodes was not a power of 4. In three dimensions, this problem becomes more acute as nodes would have to be a power of 8 to result in a homogeneous three dimensional grid. In order to avoid this problem, and to comply with the definition of the placement definition, cf. definition 4 on page 11, we use a grid-splitting scheme in order to generate placements into arbitrary 3-D grids.

This grid-splitting method recursively splits an arbitrarily chosen $n_1 \times n_2 \times n_3$ grid along the largest dimension. At the same time, the nodes are recursively split according to their position along the same dimension. This splitting process is continued until each node u is assigned to a unique position $f_V(u) \in \{1, \ldots, n_1\} \times \{1, \ldots, n_2\} \times \{1, \ldots, n_3\}$. For illustration purposes, figure 4.16 shows a 2-D version of this recursive grid-splitting procedure for a 25-node circuit and a 5×5 grid.

The size of the grid dimensions n_1, n_2, and n_3, is chosen such that the grid closely resembles a cube with approximately N nodes. The exact grid dimensions for an N

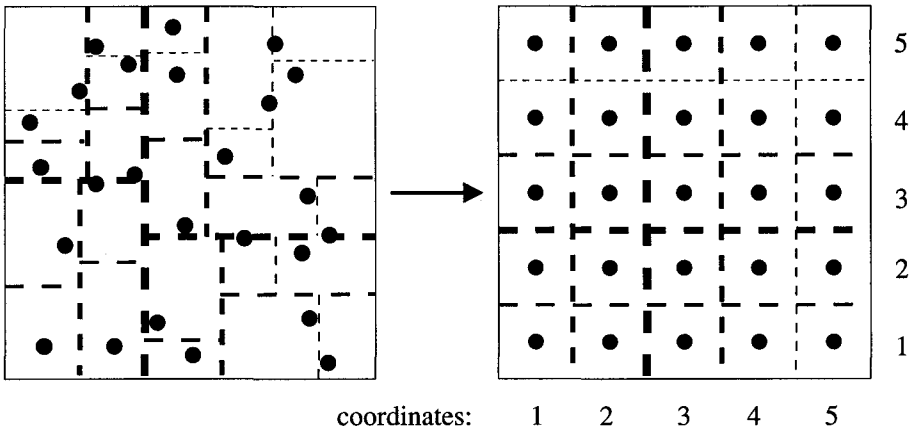


Figure 4.16: Step 4: Recursive grid splitting (left) leads to final placement of nodes (right).

node circuit are given by

$$n_1 = \left\lceil \sqrt[3]{N} \right\rceil \tag{4.16}$$

$$n_2 = \left\lceil \sqrt{N/n_1} \right\rceil \tag{4.17}$$

$$n_3 = \left\lceil \frac{N}{n_1 n_2} \right\rceil . \tag{4.18}$$

The quality of the three dimensional implementation of Gravity will be measured in the next section by comparing Gravity's wire-length results against a recursive partitioning placer that implements a similar grid-splitting strategy to the one described above.

4.4 3-D Results

Since no 3-D placement results have been published before, we needed to create a 3-D placement comparison basis on a fundamentally proven and strong technique. Partitioning placement is one of the basic and proven placement schemes in two dimensions, cf. section 3.3. Based on the simplicity of the partitioning placement method, and recent advances in partitioning algorithms, it is natural to extend

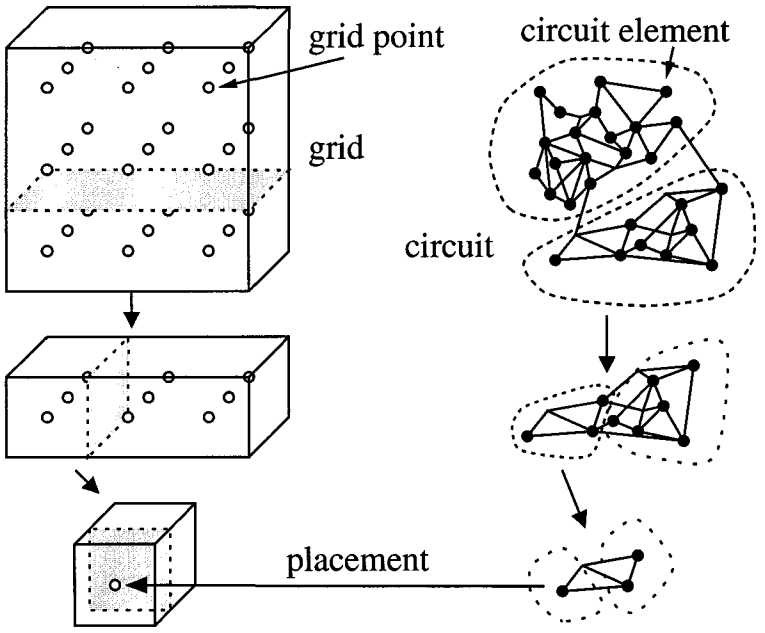


Figure 4.17: Partition placement process: simultaneous splitting of the grid and partitioning of the circuit.

partitioning placement to three dimensions for comparison purposes. In the following section we describe the 3-D partitioning placement algorithm which we used to present the first 3-D placement results for benchmark circuits at VLSI'99 [OS99]. In section 4.4.2 we compare the results of this 3-D partitioning placer with the 3-D results produced by 3-D Gravity. Finally, in section 4.4.3, we compare how three-dimensional placements compare to two-dimensional placements in terms of wire lengths for selected circuits.

4.4.1 Recursive Partitioning Placement using Grid Splitting

As outlined in section 3.3 on page 53, partitioning placement is one of the fundamental placement methodologies. For our 3-D recursive partitioning placer, we implement a grid splitting technique that mirrors 3-D Gravity's final placement step with the important difference that here the node partitioning has to be computed using a partitioner, whereas in Gravity the splitting of nodes is based on its com-

Variables and predicates:

$V = \{v_1, \ldots, v_{\|V\|}\}$	set of circuit nodes
$x[v]$	coordinates of gate for circuit node v
(a_1, a_2, a_3)	coordinates of lower left front corner
(b_1, b_2, b_3)	lengths of sides of gate array box
(n_1, n_2, n_3)	initial size of gate array box

Initial Call:

`place`$(V,(0,0,0),(n_1,n_2,n_3))$

`place`$(V,(a_1,a_2,a_3),(b_1,b_2,b_3))$

1: `if` $|V|$=1 `then`

2: $x[v_1] := (a_1, a_2, a_3)$

3: `else`

find largest side of box

4: $k := i$ such that $b_i = \max(b_1, b_2, b_3)$

split box b into two boxes b1 and b2

5: $(b1_1, b1_2, b1_3) := (b_1, b_2, b_3)$

6: $b1_k := \lfloor b_1/2 \rfloor$

7: $(b2_1, b2_2, b2_3) := (b_1, b_2, b_3)$

8: $b2_k := \lceil b_1/2 \rceil$

determine coordinates of lower left front corner of b1 and b2

9: $(a1_1, a1_2, a1_3) := (a_1, a_2, a_3)$

10: $(a2_1, a2_2, a2_3) := (a_1, a_2, a_3)$

11: $a2_k := a_k + b1_k$

partition V into subcircuits V_1 and V_2

of sizes no more than $b1_1 \cdot b1_2 \cdot b1_3$ and $b2_1 \cdot b2_2 \cdot b2_3$, respectively

12: $(V1, V2) :=$ partition$(V, b1_1 \cdot b1_2 \cdot b1_3, b2_1 \cdot b2_2 \cdot b2_3)$

invoke placement routine on subcircuits

13: `place`$(V1,(a1_1,a1_2,a1_3),(b1_1,b1_2,b1_3))$

14: `place`$(V2,(a2_1,a2_2,a2_3),(b2_1,b2_2,b2_3))$

Figure 4.18: Generic partitioning placement algorithm.

puted coordinates. Figure 4.17 illustrates how this algorithm proceeds, and figure 4.18 provides the pseudo code of this partition placement algorithm.

As briefly mentioned in section 3.7 on page 63, partitioning placement in three dimensions has been suggested before. Leeser et al. [LMV+98] used a partitioning placement method for placement in the Rothko architecture. Their partition placement method was based on a 2-D variation of partitioning placement, called *quadrisection* [SM91]. In quadrisection, the chip area is recursively split into four quadrants and circuits are recursively partitioned four ways. They extended this method into three dimensions by splitting the chip's volume into eight octants while concurrently partitioning the circuit eight ways. However, no placement results were published.

As our partitioner, we chose the hMetis hypergraph partitioner developed by Karypis et al. [KAKS97]. To our knowledge, this partitioner is currently the best of the published near-linear-run-time partitioners. Although we use recursive two-way partitioning, we could easily implement 3-D quadrisection with a few modifications to the hMetis library interface. Restrictions in the current hMetis library interface made it necessary to compute a recursive, balanced $(k+l)$-way partitioning to achieve a k:l split as is sometimes necessary in step 12 of the algorithm in figure 4.18 when an odd number of rows, columns, or layers needs to be split. While recursive multi-way partitioning increases run-time and memory requirement, it does not affect the quality of the cut [Kar99]. According to Karypis, the hMetis interface could easily be adapted to allow explicit k:l cuts.

In order to compensate for the excessive memory requirement for large $(k+l)$-way partitionings, we restricted the largest dimensions of the grid for the largest circuits to be of even length. Consequently, the largest cuts are balanced two-way cuts which require substantially less memory resources. Further, to obtain accurate estimates of the run time, assuming the hMetis interface was adapted to allow explicit k:l splits, we ran the algorithm while forcing balanced splits at all levels of recursion down to 8 or less nodes when no more partitioning intelligence is required. This is one recursion level less than for the 2-D partitioning placer, and thus, the three dimensional version is expected to have shorter run times.

Wire-length placement results for this generic partitioning placer have been pre-

Circuit	Nodes	3-D Gravity Grid	3-D hMetis Grid	Circuit	Nodes	3-D Gravity Grid	3-D hMetis Grid
19ks	2844	15 × 14 × 14	15 × 14 × 14	s15850P	10470	22 × 22 × 22	22 × 22 × 22
avq.large	25178	30 × 29 × 29	30 × 30 × 28	s35932	18148	27 × 26 × 26	27 × 26 × 26
avq.small	21918	28 × 28 × 28	28 × 28 × 28	s38417	23949	29 × 29 × 29	29 × 29 × 29
baluP	801	10 × 9 × 9	10 × 9 × 9	s38584	20995	28 × 28 × 27	28 × 28 × 27
biomedP	6514	19 × 19 × 19	19 × 19 × 19	s9234P	5866	19 × 18 × 18	19 × 18 × 18
golem3	103048	47 × 47 × 47	48 × 48 × 45	structP	1952	13 × 13 × 12	13 × 13 × 12
industry2	12637	24 × 23 × 23	24 × 23 × 23	t2	1663	12 × 12 × 12	12 × 12 × 12
industry3	15406	25 × 25 × 25	26 × 25 × 24	t3	1607	12 × 12 × 12	12 × 12 × 12
p1	833	10 × 10 × 9	10 × 10 × 9	t4	1515	12 × 12 × 11	12 × 12 × 11
p2	3014	15 × 15 × 14	15 × 15 × 14	t5	2595	14 × 14 × 14	14 × 14 × 14
s13207P	8772	21 × 21 × 20	21 × 21 × 20	t6	1752	13 × 12 × 12	13 × 12 × 12

Table 4.7: 3-D placement grids for the ACM/SIGDA suite

Circuit	Nodes	3-D Gravity Grid	3-D hMetis Grid	Circuit	Nodes	3-D Gravity Grid	3-D hMetis Grid
ibm01	12752	24 × 24 × 23	24 × 24 × 23	ibm10	69429	42 × 41 × 41	42 × 41 × 41
ibm02	19601	27 × 27 × 27	28 × 28 × 26	ibm11	70558	42 × 41 × 41	42 × 41 × 41
ibm03	23136	29 × 29 × 28	30 × 30 × 27	ibm12	71076	42 × 42 × 41	42 × 42 × 41
ibm04	27507	31 × 30 × 30	32 × 30 × 29	ibm13	84199	44 × 44 × 44	44 × 44 × 44
ibm05	29347	31 × 31 × 31	32 × 32 × 30	ibm14	147605	53 × 53 × 53	54 × 54 × 52
ibm06	32498	32 × 32 × 32	32 × 32 × 32	ibm15	161570	55 × 55 × 54	56 × 56 × 54
ibm07	45926	36 × 36 × 36	36 × 36 × 36	ibm16	183484	57 × 57 × 57	58 × 58 × 57
ibm08	51309	38 × 37 × 37	38 × 38 × 36	ibm17	185495	58 × 57 × 57	58 × 58 × 57
ibm09	53395	38 × 38 × 37	38 × 38 × 37	ibm18	210613	60 × 60 × 59	60 × 60 × 59

Table 4.8: 3-D placement grids for the ISPD98 suite

sented at VLSI'99 [OS99].

4.4.2 Result Comparison

We compare the placement results of our 3-D Gravity algorithm against the performance of the 3-D partitioning placer described above, cf. section 4.4.1. As a comparison basis we used the ACM/SIGDA and ISPD98 benchmark circuit suites, cf. section 4.2.2. For each benchmark circuit with N nodes, both algorithms computed

a placement into a homogeneous cube-like three-dimensional grid with a cube-edge length of approximately $\sqrt[3]{N}$ nodes. The exact 3-D grid dimensions are governed by equations (4.16) - (4.18), subject to the constraints described in section 4.4.1 above. Tables 4.7 and 4.8 show the exact grid dimensions. The cumulative wire lengths were estimated using an extension to three dimensions of the semi-perimeter bounding box method. This 3-D extension adds the height, width, and length of the volume spanned by the nodes in a net. This estimate is exact for nets with two or three nodes, which form the majority of all nets.

Tables 4.9 and 4.10 show wire-length and run-time comparisons on a Pentium II/300 for the ACM/SIGDA and ISPD98 circuit suites, for 250, 500, 1000, and 2000 force-step iterations. Gravity outperforms generic partitioning placement using the most powerful efficient partitioning algorithm currently available. On the more established ACM/SIGDA suite, Gravity with 250 force-step iterations runs on the average a factor of 12.5 faster while producing placements with approximately 12% less wire length. By increasing the number of iterations to 2000, Gravity can improve the wire-length advantage to over 15% while still requiring only half the time of the hMetis partitioning placer. For the newer ISPD98 circuit suite with the larger and more modern circuits, Gravity performs even better. With a 1/13 of the run time, 250-iteration Gravity produces results that are on the average almost 20% better than the partitioning placer. This advantage can be increased to 22.6% with a speed-up of 2.3 by using 2000 iterations. Detailed tables of the wire-lengths comparisons can be found in appendix A.

The target circuits for which Gravity is expected to compute placements in the future are expected to be large. For this reason it is encouraging to observe that Gravity performs even better and faster on the benchmark circuit suite with the larger circuits.

We observe, that in three dimensions, the wire-length improvement of Gravity over partitioning placement is roughly twice that in two dimensions, even though the run-time advantage is less due to the additional coordinate to be computed by Gravity, and the one-less level of recursion required by the partitioning placer.

Circuit	hMetis		250 iterations		500 iterations		1000 iterations		2000 iterations	
	length	time (s)	change (%)	speed-up	change (%)	speed-up	change (%)	speed-up	change (%)	speed-up
19ks	14,493.3	37.41	-19.87	13.87	-20.95	8.61	-21.58	4.69	-22.50	2.49
avq.large	104,104.1	302.81	-14.54	10.00	-20.54	5.83	-23.91	2.96	-25.68	1.51
avq.small	94,688.0	277.32	-15.68	10.52	-20.23	6.18	-23.70	3.17	-24.95	1.65
baluP	3,263.5	13.34	-16.05	14.79	-16.02	8.69	-16.70	4.49	-16.41	2.34
biomedP	25,239.2	106.47	-13.08	15.74	-14.64	9.65	-15.57	5.02	-16.01	2.66
golem3	687,104.9	1,519.41	-3.58	10.97	-12.25	7.24	-16.58	3.96	-18.89	2.07
industry2	78,997.7	201.72	-7.04	11.22	-7.04	6.92	-6.83	3.79	-6.15	2.00
industry3	152,962.3	300.69	-18.36	13.20	-19.13	7.75	-19.17	4.26	-19.21	2.27
p1	4,156.1	14.11	-17.70	15.93	-17.54	9.43	-18.09	5.07	-19.22	2.63
p2	18,562.5	43.16	-15.50	13.00	-16.42	7.70	-15.98	4.12	-15.69	2.13
s13207P	26,501.3	103.16	-13.32	12.17	-15.94	7.34	-17.31	3.76	-17.20	1.93
s15850P	30,950.7	121.61	-8.92	11.30	-12.16	6.59	-13.49	3.36	-12.93	1.84
s35932	57,926.1	218.19	-4.92	11.11	-10.42	6.46	-14.51	3.26	-15.67	1.70
s38417	73,282.6	252.72	2.44	8.55	-2.77	4.78	-4.10	2.45	-3.43	1.29
s38584	72,643.9	258.05	-2.10	9.78	-4.76	5.46	-5.43	2.92	-5.26	1.56
s9234P	17,670.1	85.74	-6.88	16.57	-8.46	9.99	-9.41	5.49	-9.19	2.82
structP	7,064.1	22.93	-13.33	13.31	-15.09	8.24	-15.99	4.46	-16.48	2.31
t2	8,501.9	22.21	-19.47	13.71	-20.78	8.24	-21.49	4.44	-21.08	2.23
t3	7,828.2	22.00	-14.65	13.09	-14.80	7.88	-14.54	4.21	-14.80	2.15
t4	7,375.9	21.88	-11.89	12.68	-12.20	7.61	-13.06	4.04	-13.49	2.06
t5	12,568.2	36.24	-12.35	11.99	-11.69	7.30	-11.20	3.99	-10.79	2.05
t6	7,968.1	22.52	-14.27	11.66	-14.20	6.71	-14.72	3.45	-14.52	1.79
Average			-11.87	12.51	-14.00	7.48	-15.15	3.97	-15.43	2.07

Table 4.9: Overview of 3-D placement results for the ACM/SIGDA suite

Circuit	hMetis		250 iterations		500 iterations		1000 iterations		2000 iterations	
	length	time (s)	change (%)	speed-up	change (%)	speed-up	change (%)	speed-up	change (%)	speed-up
ibm01	92,601.5	239.33	-15.16	13.87	-16.39	8.18	-16.63	4.38	-17.20	2.38
ibm02	202,121.7	391.13	-15.44	13.69	-15.78	8.23	-15.98	4.46	-16.45	2.31
ibm03	234,600.9	432.92	-16.24	12.17	-18.42	7.05	-19.00	3.81	-18.97	2.05
ibm04	301,324.1	497.46	-22.79	11.96	-23.47	7.36	-23.99	3.90	-24.45	2.05
ibm05	367,004.5	553.55	-23.79	13.79	-25.11	8.44	-25.11	4.58	-26.00	2.40
ibm06	333,985.0	655.11	-21.58	13.21	-22.93	8.01	-24.08	4.28	-24.37	2.17
ibm07	473,844.3	1,084.43	-20.14	15.04	-22.07	9.15	-23.39	4.92	-22.98	2.56
ibm08	531,860.7	1,191.15	-22.54	14.79	-23.64	9.39	-24.42	5.17	-24.73	2.73
ibm09	617,201.7	1,263.16	-23.44	13.13	-24.49	7.95	-25.24	4.43	-25.27	2.36
ibm10	832,125.1	1,761.46	-22.16	15.04	-24.57	9.12	-25.22	4.82	-26.18	2.50
ibm11	872,118.2	1,660.90	-26.25	13.72	-27.83	8.28	-29.17	4.40	-29.83	2.29
ibm12	991,783.9	1,863.66	-20.84	14.57	-21.12	9.56	-21.59	5.40	-22.10	2.89
ibm13	1,000,941.8	1,864.37	-19.58	12.49	-21.08	7.59	-21.80	4.03	-22.27	2.12
ibm14	1,657,408.1	3,064.37	-17.56	12.07	-20.12	6.84	-21.58	3.70	-21.89	1.98
ibm15	1,994,685.8	3,768.55	-10.54	12.19	-13.31	7.27	-14.07	3.89	-14.27	2.13
ibm16	2,222,138.0	4,029.56	-14.28	11.52	-17.02	7.13	-18.07	3.84	-18.65	2.04
ibm17	2,745,042.7	4,462.59	-18.23	12.66	-20.31	7.83	-21.32	4.21	-21.72	2.18
ibm18	2,639,356.6	4,284.66	-24.62	11.47	-27.01	7.07	-28.92	3.66	-29.61	1.94
Average			-19.73	13.19	-21.37	8.03	-22.20	4.33	-22.61	2.28

Table 4.10: Overview of 3-D placement results for the ISPD98 suite

4.4.3 From 2 to 3 Dimensions

It is expected that 3-D VLSI will have shorter average wire lengths than traditional 2-D VLSI. For instance, the average distance between neighbours in a N-node square grid is $\Theta(\sqrt{N})$, but only $\Theta(\sqrt[3]{N})$ in an N node cubic grid. How this expectation translates into wire-length improvement for actual circuits is not easy to estimate. For this reason, we have compiled in table 4.11 wire-length results for four circuits evenly covering the dynamic range from 800 to 210,000 nodes. For each circuit 3-D Gravity computed a placement in $d = 2$, $2\frac{1}{3}$, $2\frac{2}{3}$, and 3 dimensions. These non-integer dimensions mean to indicate that the base of the grid has an edge length of $\sqrt[d]{N}$. Thus a 2-D grid, is of size $\sqrt{N} \times \sqrt{N} \times 1$, and a 3-D grid of size $\sqrt[3]{N} \times \sqrt[3]{N} \times \sqrt[3]{N}$.

Circuit	Nodes	Grid			All Nets Counted		2- and 3-node Nets	
					Length	Change	Length	Change
p1	833	29	29	1	5027.6		2778.9	
		17	17	3	3751.3	-25.38%	2202.3	-20.74%
		12	12	6	3418.3	-32.00%	2032.3	-26.86%
		10	10	9	3385.9	-32.65%	2021.6	-27.25%
s9234P	5866	77	77	1	27055.6		18728.4	
		39	38	4	18749.9	-30.69%	13745.2	-26.60%
		26	26	9	16573.6	-38.74%	12570.3	-32.88%
		19	18	18	16041.9	-40.70%	12241.2	-34.63%
ibm06	32498	181	180	1	712282.8		233084.3	
		81	81	5	365606.2	-48.67%	126683.4	-45.64%
		49	48	14	277323.5	-61.06%	101553.6	-56.43%
		32	32	32	254276.8	-64.30%	95390.0	-59.07%
ibm18	210613	459	459	1	7128339.7		2179268.3	
		188	187	6	3140755.9	-55.93%	1011482.3	-53.58%
		98	98	22	2111055.3	-70.38%	728096.2	-66.58%
		60	60	59	1875854.6	-73.68%	655591.2	-69.91%

Table 4.11: Wire-length improvement moving from 2-D to 3-D.

In general, for dimension d and an N-node circuit, we selected a grid size close to $\sqrt[d]{N} \times \sqrt[d]{N} \times N/(\sqrt[d]{N})^2$.

Next to the usual total wire-length estimate, table 4.11 also shows the total wire lengths of nets with 2 and 3 nodes. For such nets, the semi-perimeter bounding box approximation of the wire length is exact.

We notice that a substantial wire-length advantage can be achieved for large circuits even when only a few layers in the third dimension are employed. While small circuits exhibit only a modest wire-length improvement in 3-D, larger circuits clearly benefit from the third dimension. Small circuits p1 and s9234P experience only a 30%-40% improvement in a 3-D placement, whereas ibm18 saves approximately 55% wire length in $2\frac{1}{3}$ dimensions and over 70% in three dimensions.

4.5 Summary

In this chapter, we have presented a new iterative force-directed placement algorithm which uses a bucket rescaling technique in lieu of repulsive forces. We have shown how this algorithm, which we called Gravity, runs in linear time with respect to circuit size, and how its data structures and implementation have been designed to take advantage of cache memory and superscalar features common to modern CPUs.

We compared the placement results to the proven technique of partitioning placement using the best efficient minimum-cut partitioner available to date. Our algorithm outperformed the partitioning placer in time and wire length in both two and three dimensions. Gravity bettered the wire-length results of the partitioning placer using an order of magnitude less run time. In three dimensions, Gravity's wire-length improvement was approximately 20% for the benchmark circuit suite with the larger, more modern circuits.

For completeness we also compared Gravity to standard cell placement results of the best published standard cell placer for smaller, older circuits. Lacking a refined final placement step for standard cell layouts, Gravity produced wire-length results that were on the average approximately 13% larger when using comparable run times. However, Gravity's strength is expected to lie with modern larger circuits for which we believe Gravity may perform better: When comparing standard cell placements only for circuits which have pin distributions closer to those typical for the larger circuit suite, Gravity managed to produce slightly better wire-lengths (approximately 1% improvement).

Lastly, we presented a wire-length comparison as the placement grid dimensions were varied from two to three dimensions. While smaller circuits showed some wire-length improvement, the larger circuits showed a remarkable reduction in wire length of approximately 50% for a small number of vertical layers, and as much as 70% for a true 3-D cubic grid.

Chapter 5

Bendless Embeddings

In section 2.5 on page 37, we encountered some optical grid-based architectures in which a potentially significant latency penalty was incurred whenever a signal had to be routed from one grid line to an intersecting grid line in order to reach its destination. In section 1.4.5 on page 27, we defined this problem for hypergraphs and n-dimensional grids as Straight Line on a Grid. Straight Line on a Grid is an extension of the 2-D problem Edge Embedding on a Grid for graphs formalized by Garey and Johnson [GJ79]. We also note that in the case of bendless embeddings, the routing of edges is implied to follow the grid lines that join neighbours. Thus, we use the terms "straight-line placement" and "bendless embedding" interchangeably.

In this chapter, we examine how some popular modern network topologies can be embedded into a grids without bending edges, thus facilitating speedy communications in technologies in which bends incur large latency penalties. First, we will describe some of the terminology used when describing graph embeddings, and how some optical architectures may benefit from a bendless embedding. In section 5.2, we present straight-line embeddings for the traditional network topologies torus, hypercube, and binary tree. Finally, in section 5.3, we introduce an efficient straight-line placement of a more modern topology, the star graph, which has been suggested as an alternative to the hypercube.

5.1 Embedding Model

In [Lei92], Leighton defines several terms to describe embeddings. When embedding one graph into another, we say we embed the *guest graph* into the *host graph.* We call the ratio of the number of host graph nodes over the number of guest graph nodes the *expansion* of the embedding. The *dilation* of an embedding is the maximum path length in the host graph between neighbours in the guest graph. The maximum number of guest graph nodes that are embedded into the same host graph node is called the *load* of an embedding.

Consider a grid-based computing architecture, with n^d electronic nodes arranged in a d dimensional array and with an optical medium interconnecting the n nodes in each row, column or dimension. The electronic nodes can represent printed circuit boards, multi-chip modules, or integrated circuits. Hence, each node may contain multiple processing elements. Due to the bandwidth advantage of optics, the optical medium within a row or column can support multiple channels; in a typical configuration, each electronic node may reserve its own contention-free broadcast channel along a row and a column. This optical *multi-channel mesh* architecture has been called a *hypermesh* [Szy95].

This optical model assumes that the passing of a packet between any nodes along one dimension requires one *logical hop.* For example, passing a packet from one node to another node along a contention-free optical channel in the same row takes one hop regardless if the nodes are spatially nearest neighbours or at opposite ends of the row. This model is valid whenever the transmission delay along a grid line is shorter than the switching delay from one grid line to another, e.g. Dowd's star coupler architecture (figure 2.8 on page 39), or when such dimensional "hops" are mandated by synchronization requirements, e.g. Guo et al's bus cycles (section 2.5 on page 37). When embedding a guest graph into this optical grid model, the number of dimensions traversed in an edge embedding is more relevant than the physical distance, since broadcasts along a row or column require one logical hop regardless of the destination's position within the row. In the architecture of Guo et al. [GMH+91], such a "hop" is referred to as a bus cycle, cf. section 2.5 on page 37. Equivalently, the number of bends of an embedding is more important than its dilation. If an embedding of a guest graph edge into the optical multi-channel grid

Topology	Nodes	Degree	Diameter
$n \times n$ torus	n^2	4	$\begin{cases} n & \text{if } n \text{ even} \\ n-1 & \text{if } n \text{ odd} \end{cases}$
height-n binary tree	$2^{n+1}-1$	3	$2n$
n-cube	2^n	n	n
n-star	$n!$	$n-1$	$\lfloor \frac{3}{2}(n-1) \rfloor$

Table 5.1: Characteristics of some topologies.

has k bends, then a packet requires $k+1$ hops to get from one guest node to its neighbour. In this model, the number of bends replaces the dilation as a distance measure of an embedding. As a result, the diameter of this optical grid-like network can be much smaller than that of the electrical grid [Szy95]. In the worst case, it requires d logical hops to transmit a packet between the furthest apart nodes in an d-dimensional optical multi-channel grid model. In the conventional grid model, where packets can only hop along host graph edges between physically nearest neighbours, it requires $d \cdot (n-1)$ logical hops to transmit a packet between the furthest apart nodes in the worst case.

Before we present a bendless embedding of the newer star graph topology in section 5.3, we first show how simple transformations can be used to find straight-line placements for the torus, hypercube, and tree.

5.2 Torus, Hypercube, Tree

Tori, hypercubes, and trees have long been studied and used as popular high-performance network topologies [Lei92]. Table 5.1 compares these topologies. All three topologies have an underlying orthogonal structure which makes them ideal candidates for bendless embeddings into grids. In this section, we will demonstrate such embeddings first for the torus, then for the hypercube, and finally for the tree.

A torus is a double-ring structure in three dimensions resembling a doughnut. Figure 5.1 (left) shows a 5×5 torus: five small rings of five nodes each, are inter-

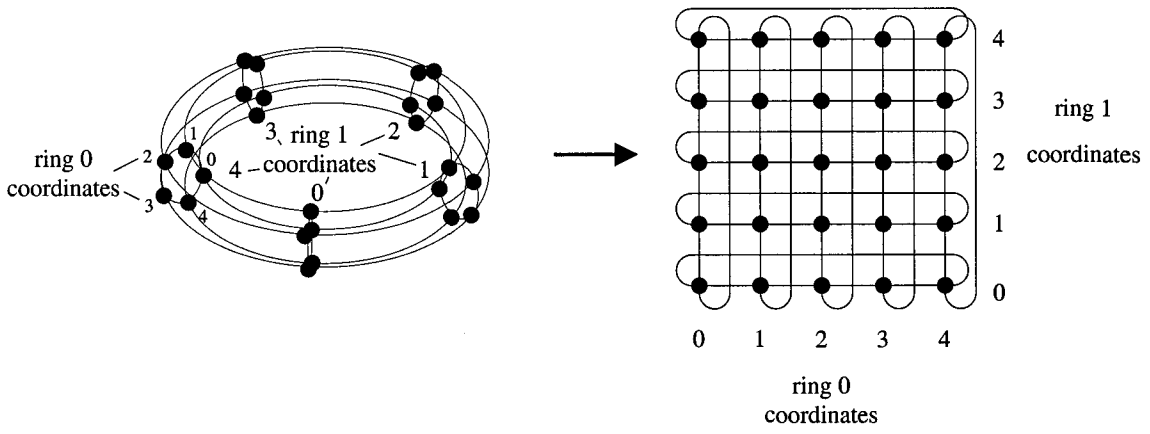


Figure 5.1: A 5×5 torus (left) is placed into a 5×5-grid without bending edges.

connected on a large doughnut-like ring. Such a 3-D torus can be flattened into a two-dimensional grid, where it represents a mesh with wrap-around edges, cf. figure 5.1 (right). In general, a $(d+1)$-dimensional torus, can be placed into a d-dimensional grid, by reusing the d torus ring coordinates as the d grid coordinates.

Bendless Torus-to-Grid Mapping Given a $(d+1)$-dimensional torus T whose nodes are labelled by d ring coordinates $t = (t_1, \ldots, t_d)$, and a d-dimensional grid G whose nodes have coordinates $x = (x_1, \ldots, x_d)$, the grid coordinates of each torus node t are

$$x = t. \tag{5.1}$$

For grids of less than d dimensions, the higher ring coordinates can be projected onto lower dimensional grid coordinates.

Mapping a tree into a grid is only slightly more involved. Consider a binary tree and a square. We place the root into the centre of the square. Then, we partition the square horizontally into two halves and place the children of the root at the centres of the subpartitions. This process continues recursively, alternating between horizontal and vertical partitionings. Such a tree is sometimes called an H-tree. Figure 5.2 shows how a 31-node binary tree can be placed into a 7×7 grid.

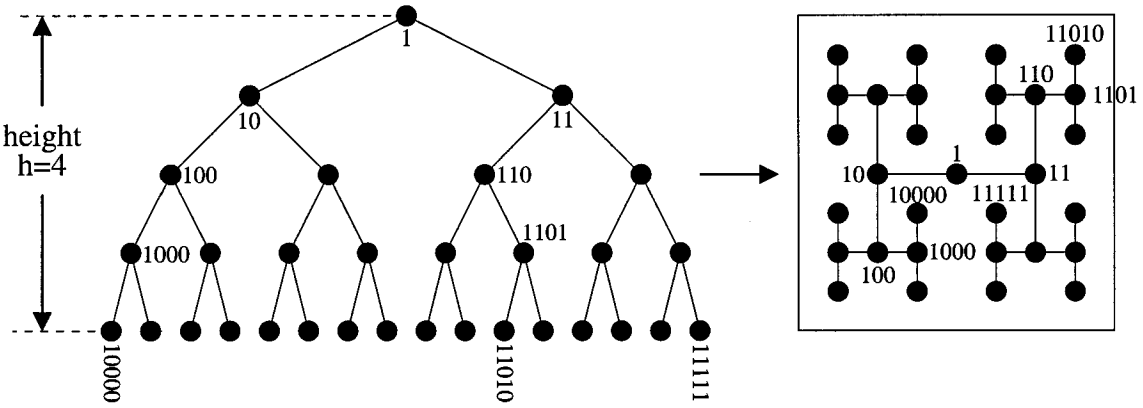


Figure 5.2: A 31-node binary tree (left) is placed into a 7×7-grid without bending edges.

Bendless Binary-Tree-to-Grid Mapping Given a complete binary tree of height h whose nodes are labelled by counting tree nodes in depth-first-search order, and a d-dimensional grid. Let $b_0, \ldots, b_{\lfloor \log_2 t \rfloor}$ the bits in the binary representation of node label t, and let $I = \{\hat{\imath}_1, \ldots, \hat{\imath}_d\}$ be the set of d coordinate base vectors. The grid position of tree node t is given by

$$x = 2^{\lceil h/d \rceil} \left[\sum_{i=1}^{d} \hat{\imath}_i + \sum_{i=1}^{\lfloor \log_2 t \rfloor} \hat{\imath}_{(i \bmod d)+1} 2^{-\lfloor \frac{i+1}{d} \rfloor} (2b_i - 1) \right]. \tag{5.2}$$

Incomplete binary trees are placed by omitting absent nodes, and k-ary trees can be placed by subdividing the square into k subpartitions at each tree level at the cost of increased edge congestion. Variations of the above embedding are possible as long as each child node and its parent share a grid line. Guo et al. [GMH+91] provide an alternative style for bendless binary tree embeddings.

A hypercube is the extension of a three dimensional cube to arbitrary dimensions. The n-dimensional hypercube is also a special case of an n-dimensional torus, where all n-rings have exactly two nodes. A one-dimensional hypercube is just a point. Given two n-dimensional hypercubes (or n-cubes) with the same node labels, a $(n+1)$-cube is constructed by connecting all pair of nodes with the same labels. Hypercube nodes carry a binary labelling. Each bit in the n-bit label of a node in an n-cube, represents its coordinate, 0 or 1, in one of the n dimensions. Labels of

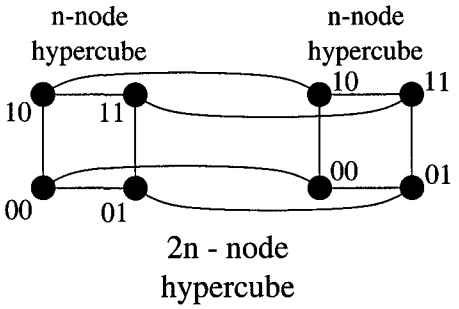


Figure 5.3: Two n-node bendless hypercube embeddings are placed side-by-side to form a bendless $2n$-node hypercube embedding.

neighbours in a hypercube differ in exactly one bit. Figure 5.3 shows how to 2-cubes (squares) are combined to form a 3-cube. Similarly, if a bendless embedding of an $(n-1)$-cube is known, an n-cube can be placed without bends by placing two $(n-1)$-nodes side-by-side such that all like-labelled nodes share the same row. Such a recursive placement definition was previously suggested by Guo et al. [GMH+91] and can be formalized as follows.

Bendless n-cube Grid Placement Given an n-cube, a d-dimensional grid, a dimension k. Let $u(i) = (u_1(i), \ldots, u_d(i))$ be the coordinates of node i in a bendless embedding of an $(n-1)$-cube. Let $w = \max_{0 \le i < 2^{n-1}} u_k(i)$ be the maximum coordinate in dimension k of any node. Then the coordinates $v(i) = (v_1(i), \ldots, v_d(i))$ of node i of an n-cube are given by

$$v_j(i) = \begin{cases} u_j(i(mod 2^{n-1})) & \text{if } j \neq k, \\ w + u_j(i(mod 2^{n-1})) & \text{if } j = k. \end{cases} \quad (5.3)$$

The aspect ratio (height/width) of the grid holding a hypercube varies depending on in which dimensions two lower level hypercubes have been placed side-by-side to form a higher level cube. By choosing alternate dimensions for each level, the aspect ratio can be kept at unity for even-n-dimensional hypercubes in a 2-D grid. Figure 5.4 shows a straight line placement of a 7-cube into a 4×32 grid.

These embeddings of the torus, tree, and hypercube, all have a load of one. The expansion is also one for the torus and hypercube, and $\frac{(2^h-1)^d}{2^{h+1}-1}$ for a binary tree when

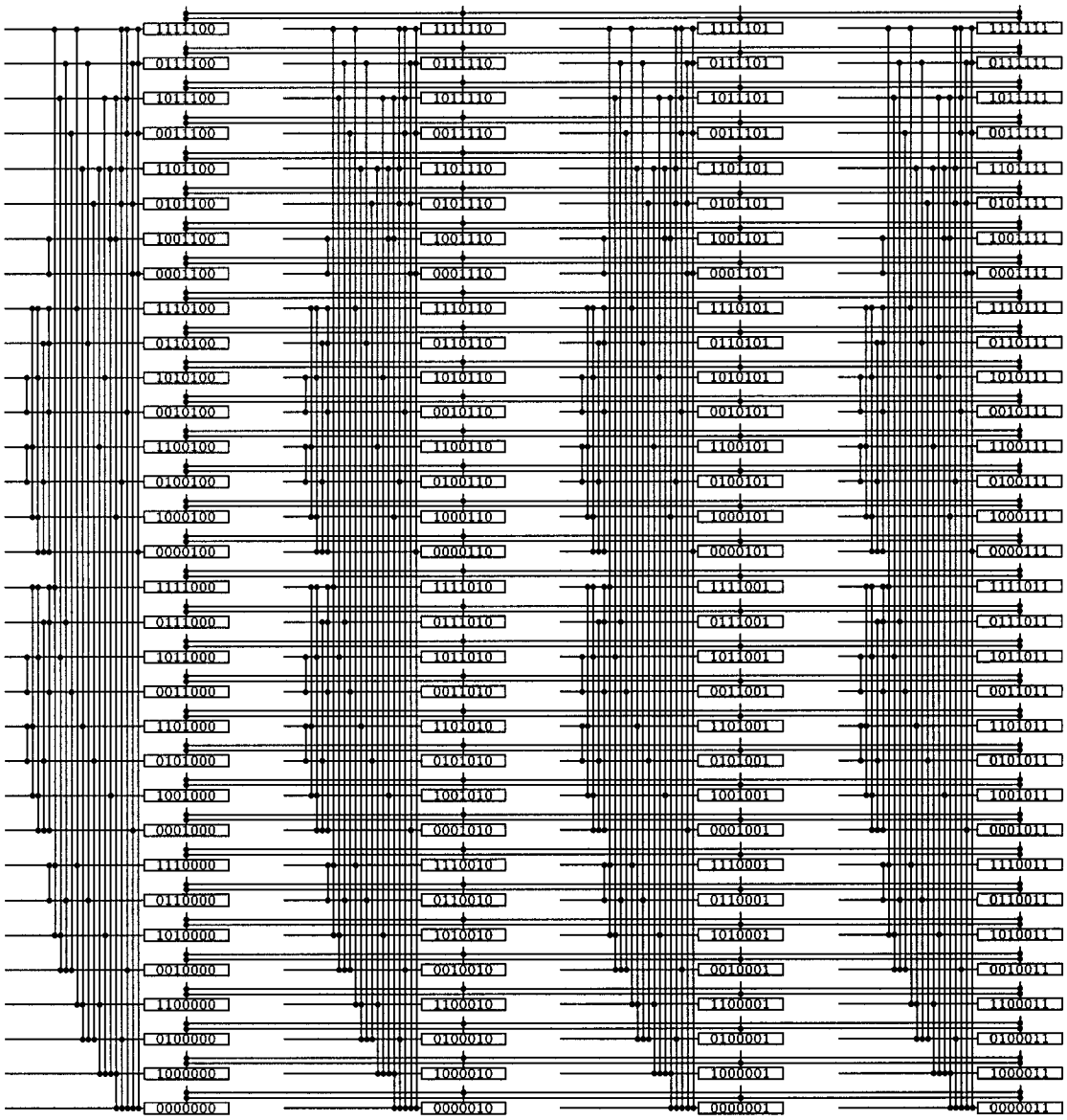


Figure 5.4: Bendless embedding of a 128-node hypercube into a 4×32-grid.

h is a multiple of d.

5.3 Star Graph

In this section, we describe an embedding of the star graph into a rectangular grid of d dimensions such that the embedding has no bends, that is, neighbours in the star graph always differ in exactly one coordinate in the grid. This embedding was based on an idea presented in [Obe95], and refined and optimized in [OS97]. In particular, [OS97] introduces an asymptotically tight bound on the expansion of the embedding in two dimensions, and an efficient contraction method for reducing the aspect ratio of the embedding to near unity.

To embed an n-star, the grid can have any number of dimensions d between 1 and $n-1$. The embedding has load 1 and an expansion of at most $n^{d-1}/d!$. The size of the grid will be at most $\underbrace{n \times \cdots \times n}_{d-1} \times (n!/d!)$. We optimize the size of the host grid using clique-partitioning to produce embeddings with expansions as low as unity. In two dimensions, for even n, the grid will be no larger than $n \times n(n-2)!$, and have an expansion of no more than $1\frac{1}{n-1}$. Further, we show how we can use a contraction method to efficiently embed the star graph into an optical grid with near-unity aspect ratios. Contraction on a two dimensional embedding will yield a grid of size no larger than $n \times n$ for even n with a load of $(n-2)!$.

5.3.1 Introduction

5.3.1.1 Motivation

Akers et al. [AHK87] introduced the star graph as an alternative to the hypercube. The n-star is an $n!$-node regular automorphic graph. Nodes are labelled by different permutations of n symbols. Nodes are neighbours if the label of one can be transformed into the label of the other by swapping the first symbol with one of the other symbols. Due to its small diameter ($\lfloor 3(n-1)/2 \rfloor$) and sublogarithmic degree ($n-1$), the star graph out-classes the hypercube in many aspects. See Day and Tripathi [DT94] for a comparative study. A more detailed analysis of star-graph properties was provided by Qiu et al. [QAM94]. Akl [Akl97] offers a study of

applications on the star graph.

A way of embedding grids into the star graph has been shown by Ranka et al. [RWY93]. Thus a star graph can simulate an n-dimensional grid efficiently. We are proposing the opposite, an embedding of the star graph into a grid.

Motivated by the need for embeddings of star graphs on two dimensional devices such as a printed circuit board, Hoelzeman and Bettayeb [HB94] investigated the genus of a star graph. The genus of a graph determines the number of "bridges" that have to be placed on a two dimensional surface to avoid edge crossings. They found that the star's genus is lower than the hypercube's, and concluded that the layout of a star graph should be more efficient than the layout of a hypercube of similar size.

In contrast to board layouts, edge crossings are not a problem in free-space optics, for example. However, unnecessary bends can be problematic. Up to this point, there has not been a convenient way of embedding a star graph into a common rectangular physical device. We propose a way of embedding star graphs into two, three, or more dimensions, such that the positions of neighbours differ in one coordinate only. Hence, in a two dimensional embedding, neighbours always share the same row or column.

5.3.1.2 Embedding Overview

Our unoptimized embedding of the star graph into the d dimensional grid will have a load of 1, an expansion of at most $n^{d-1}/d!$, and a dilation of at most $n!/d! - 1$. Most importantly, the embedding will have no bends. Finding a embedding of a graph into a grid without bending edges and having load and expansion one is a known NP-complete problem called *Edge Embedding on a Grid* [GJ79]. Our proposed embedding solves *Edge Embedding on a Grid* for selected even-n n-star embeddings in two dimensional grids. When the expansion-one condition is removed, our proposed embedding strategy solves *Straight Line on a Grid*, cf. 5 on page 27, for all n-stars in arbitrary dimensional grids.

We also present two optimization methods, called *group optimization* and *contraction*, which can reduce the expansion and dilation considerably. In two dimensional grid embeddings, group optimization guarantees an expansion below $1\frac{2}{n-1}$

while maintaining unity load. Contraction increases the load without increasing the degree or introducing internal edges within vertices in the contracted node. At the same time, contraction improves the aspect ratio of the host grid, and thus the dilation of the embedding. In two dimensions, the dilation is no more than $n+1$ while the aspect ratio is no greater than $n+1:n$. As well as improving the aspect ratio, the contraction operation we propose solves the known NP-complete problem *Graph Homomorphism* [GJ79] for n-stars embedded into selected d-dimensional grids.

In the remainder of this section, we will first describe the star graph introduced by Akers et al. [AHK87] and the d-dimensional grid. Then we will present and prove the embedding strategy. Finally, we show a few examples of embeddings.

5.3.2 Grid and Star Graph

Recalling definition 3b on page 8, a d-dimensional grid $G(V_G, E_G)$ of size $N = n_1 \times \cdots \times n_d$ has N nodes and extends n_i nodes into dimension i. The N grid nodes in V_G are labelled by d coordinates, that is

$$\begin{aligned} v &\in V_G && (5.4)\\ \Leftrightarrow v &= (v_1, v_2, \ldots, v_d) && (5.5)\\ &\quad \text{where } 0 \le v_i \le n_i - 1 \text{ and } 1 \le i \le d. \end{aligned}$$

Two nodes $u, v \in V_G$ are neighbours, i.e., $\{u, v\} \in E_G$, if their labels only differ by one in exactly one coordinate, that is

$$\begin{aligned} &\{u, v\} \in E_G && (5.6)\\ \Leftrightarrow &|u - v| = 1. && (5.7) \end{aligned}$$

An n symbol star graph $S = (V_S, E_S)$, or n-star as introduced in [AHK87], is a graph with $N = n!$ nodes of degree $n-1$. The $n!$ nodes are each labelled by a different permutation of n symbols from a set $\mathcal{S}= \{s_1, s_2, \ldots, s_n\}$. We will choose these symbols to be the numbers $0 \ldots n-1$. Formally, we say,

$$\begin{aligned} v &\in V_S && (5.8)\\ \Leftrightarrow v &= (v_1, v_2, \ldots, v_n) \text{ and } \{v_1, v_2, \ldots, v_n\} = \mathcal{S}. && (5.9) \end{aligned}$$

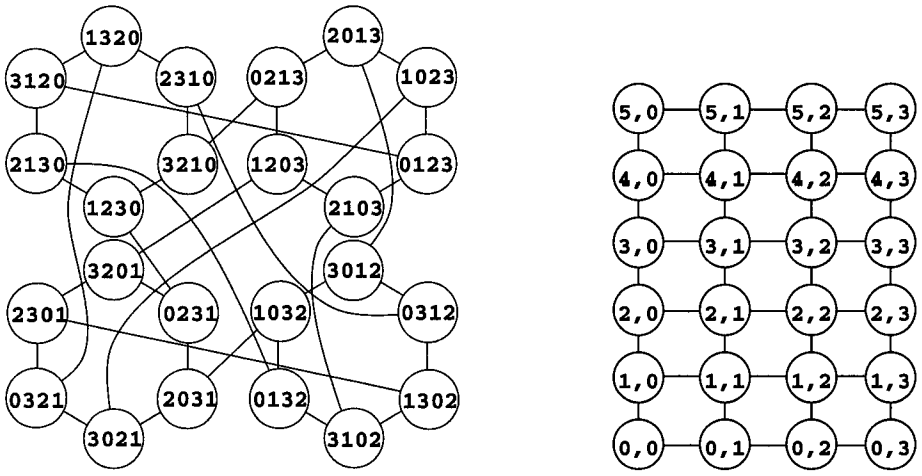


Figure 5.5: The 4-Star (left) and the 4×6-Grid

As mentioned in section 5.3.1.1, two nodes $u, v \in V_S$ are neighbours if the label of u can be transformed into the label of v by exchanging the first symbol of u's label with one of the $n-1$ remaining symbols in its label, that is

$$\{u, v\} \in E_S \tag{5.10}$$

$$\Leftrightarrow \exists i \neq 1 \; u_1 = v_i \text{ and } \forall j \notin \{1, i\} \; u_j = v_j. \tag{5.11}$$

For example, node 23410 of the 5-star has the four neighbours 32410, 43210, 13420, and 03412. Figure 5.5 shows a 4-star and a 6×4-grid.

For comparison, respectively, $n!$-node star graphs and $n!$-node square grids have degrees $n-1$ and 4, diameters $\lfloor 3(n-1)/2 \rfloor$ and $2\sqrt{n!} - 2$, and average distances $n + 2/n - 4 + \sum_{i=1}^{n} 1/i$ and $2(\sqrt{n!} - 1/\sqrt{n!})/3$.

5.3.3 Embedding

Until now, it has been unknown if the star graph could be embedded into an orthogonal structure such that all neighbours are in either the same row or column. Fortunately, we were able to determine such embeddings for arbitrary size star graphs and arbitrary dimensional host-grids. Hereafter, we will assume that the grid which is to host an n-star is of n or less dimensions. The grid may have higher dimensions, but we will not utilize more than n dimensions.

Let us introduce and prove the embedding strategy in section 5.3.3.1. In section 5.3.3.2, we will show a way of minimising the expansion of the embedding through a method we call *group optimization*, and will show some example embeddings. Finally, in section 5.3.3.3, we use a *contraction* method to improve the aspect ratio of the embedding to near unity.

5.3.3.1 Embedding Strategy

To construct an embedding, we will look at every star node, and insert it into two appropriate sets. We call these sets *clusters* and *groups*. The label of a node will determine its group and cluster, which in turn will determine the node's position in the grid. First, we describe how such an embedding is found. Then, we shall formally prove that such an embedding can always be found.

Assume we are dealing with an $n!$-node n-star, and assume we are trying to embed it into a rectangular grid of d dimensions. In this case, we will put each node into one of $n!/d!$ groups, each containing $d!$ nodes, and into one of $n!/(n-d+1)!$ clusters, each containing $(n-d+1)!$ nodes.

Clusters A cluster is a $(n-d+1)$-substar within the n-star. For each node, its cluster is determined by the last $d-1$ symbols in the label. For example, if we want to find the correct cluster for node $v = \texttt{30421}$ given that we want to embed it into a three-dimensional grid, then the last $d-1$ symbols in label of v are $\texttt{21}$, and hence v belongs into cluster C_{21}.

More formally,

$$\begin{aligned} v &\in C_{c_1 c_2 \ldots c_{d-1}} \\ \Leftrightarrow (v_{n-d+2}, v_{n-d+3}, \ldots, v_n) &= (c_1, c_2, \ldots, c_{d-1}). \end{aligned} \tag{5.12}$$

In order to specify a node's position in the d-dimensional grid, we need to specify d coordinates. The first $d-1$ coordinates are determined from the cluster, and the last coordinate follows from the node's group.

We use the $d-1$ symbols that mark a cluster as the first $d-1$ grid coordinates of the nodes in that cluster. Nodes in our example-cluster C_{21} will have 2 and 1 as their first two coordinates when embedded into the grid host-graph. It should be

noted that not all grid positions will be associated with a cluster. Host graph nodes whose labels have duplicates in their first $d-1$ coordinates cannot be associated with any star graph nodes since it would imply that such a star graph node had duplicate symbols in its label. For example, host graph node `2,2,3,1` will always remain unoccupied by guest nodes because the label of a guest node would have to end with 223. However, no permutation of different symbols has any duplicates.

Groups To find the last coordinate of a node, we look at its group. A node's group is determined by the $n-d$ symbols in positions 2 through $n-d+1$ in its label. For instance, $v = 30421$ would be a member of group G_{04} because symbols 2 through $n-d+1$ in v's label are 0 and 4. The other members of G_{04} are 10423, 10432, 20413, 20431, and 30412. Formally, we say

$$\begin{aligned} v &\in G_{g_1 g_2 \ldots g_{n-d}} \\ \Leftrightarrow (v_2, v_3, \ldots, v_{n-d+1}) &= (g_1, g_2, \ldots, g_{n-d}). \end{aligned} \tag{5.13}$$

Since $|G| = d!$, we have $n!/d!$ groups. We arbitrarily assign an ordering to the groups and number them accordingly from 0 to $n!/d! - 1$. In the proof we will call this ordering mapping O. The group's index in the ordering will serve as the d^{th} coordinate for all of the group's nodes. Thus, the group and cluster of a node determine all d coordinates of the node's embedding in the host graph. See figure 5.6 for a visualization of the grid coordinate assignment procedure.

Now, we are ready to state the embedding as a theorem.

Theorem 1 *Assume an n-star $S(V_S, E_S)$ where all nodes are labelled using symbols from set $\mathcal{S}$= $\{0, \ldots, n-1\}$, and a d-dimensional grid host graph $G(V_G, E_G)$ of size $\underbrace{n \times \cdots \times n}_{d-1} \times (n!/d!)$. Further, consider a one-to-one mapping O : {all permutations of $n-d$ symbols from $\mathcal{S}$} $\rightarrow \{0, 1, 2, \ldots, n!/d! - 1\}$. Then S can be embedded into G with no bends and load 1. Such an embedding is achieved by embedding every node $v \in V_S$ into a node $u \in V_G$ such that*

$$\begin{aligned} &(u_1, u_2, \ldots, u_{d-2}, u_{d-1}, u_d) \\ &= (v_{n-d+2}, v_{n-d+3}, \ldots, v_{n-1}, v_n, O((v_2, v_3, \ldots, v_{n-d+1}))). \end{aligned} \tag{5.14}$$

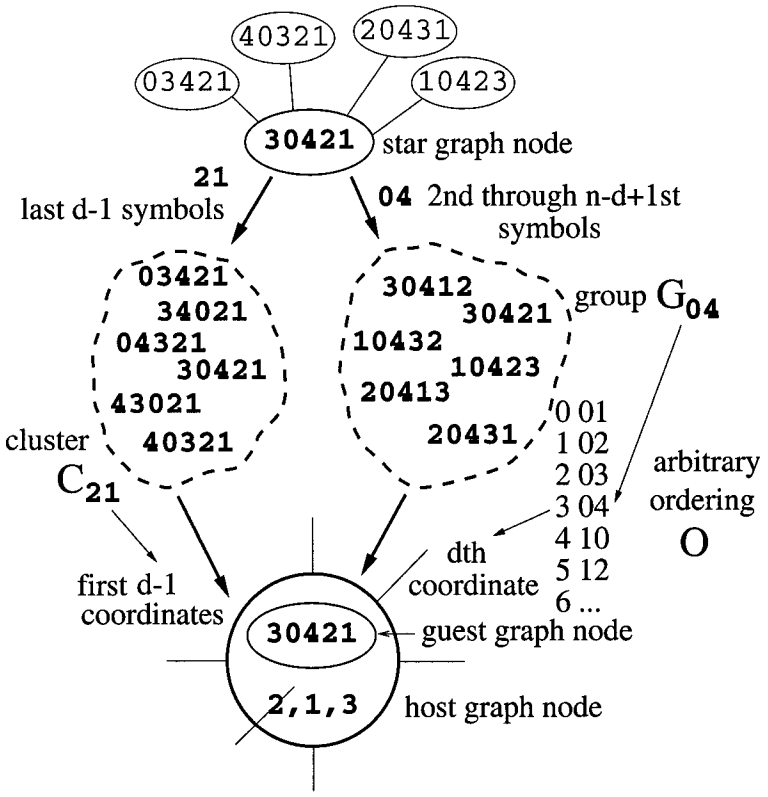


Figure 5.6: Finding the coordinates in the host graph.

Proof We call S the guest graph, and G the host graph.

No two different nodes can belong to the same group and the same cluster since nodes with the same group symbols (symbols $2 \ldots n-d+1$) have a different permutation of cluster symbols (symbols n-d+2 ... n), and vice versa. Hence, every guest graph node is embedded into a unique host graph node.

To show that there are no bends between guest graph neighbours in the host graph, we need to show that the labels of host graph nodes of neighbours in the guest graph differ by one coordinate only.

All members of one cluster differ in only one coordinate in their host graph labels since the cluster determines $d-1$ of d coordinates, which are common. Further, recall that the labels of neighbours of a star graph node are found by swapping the first symbol in a node's label with one of the $n-1$ other symbols. It follows that

a node in a cluster C must have $n-d$ of its $n-1$ neighbours in C since $d-1$ neighbours can only be reached by changing one of the last $d-1$ symbols. But $n-d$ neighbours leave the last $d-1$ symbols unchanged and those neighbours are thus within the same cluster.

The remaining $d-1$ neighbours of a node v (which are found by swapping the first symbol with one of the $d-1$ last ones of node v) are in the same group as v since the labels of those neighbours have the same symbols 2 through $n-d+1$. v and its neighbours in the same group share the same d^{th} coordinate in the host graph label. Moreover, these $d-1$ neighbours of v in v's group differ with v in only one of the last $d-1$ symbols, and since the last $d-1$ symbols are the first $d-1$ coordinates in the host graph, the labels of these guest graph neighbours differ by only one coordinate in the host graph.□

5.3.3.2 Group Optimization

According to theorem 1 we embed the 4-star of figure 5.5 into a 4×12-grid as shown in figure 5.7. In figure 5.7, every group occupies one row and some nodes are left "idle". We can improve the efficiency of the embedding by mapping two groups into every row in order to reduce the number of idle nodes, thus reducing the expansion of the embedding. Ideally, two groups will occupy distinct nodes and fully occupy all the nodes in a row, resulting in an embedding with unity load and unity expansion. To achieve this improvement, we need to find a perfect matching between pairs of groups. Figure 5.8 illustrates an embedding of the 4-star into a 4×6 grid (from the original embedding into a 4×12 grid shown on figure 5.7) obtained using a perfect group matching.

For general d-dimensional embeddings, we shall refer to a group's d^{th} coordinate as its row, just as in the 2-dimensional case. We can put two groups with all their nodes onto the same row if none of their nodes claim the same grid node, as determined by the nodes' clusters. As we will show in theorem 2, it turns out that we can put those groups that differ by two or more symbols in their labels onto the same row.

In order to determine which groups can be placed on the same row in a group-optimized embedding, we shall place all groups with the same symbols into a **meta**

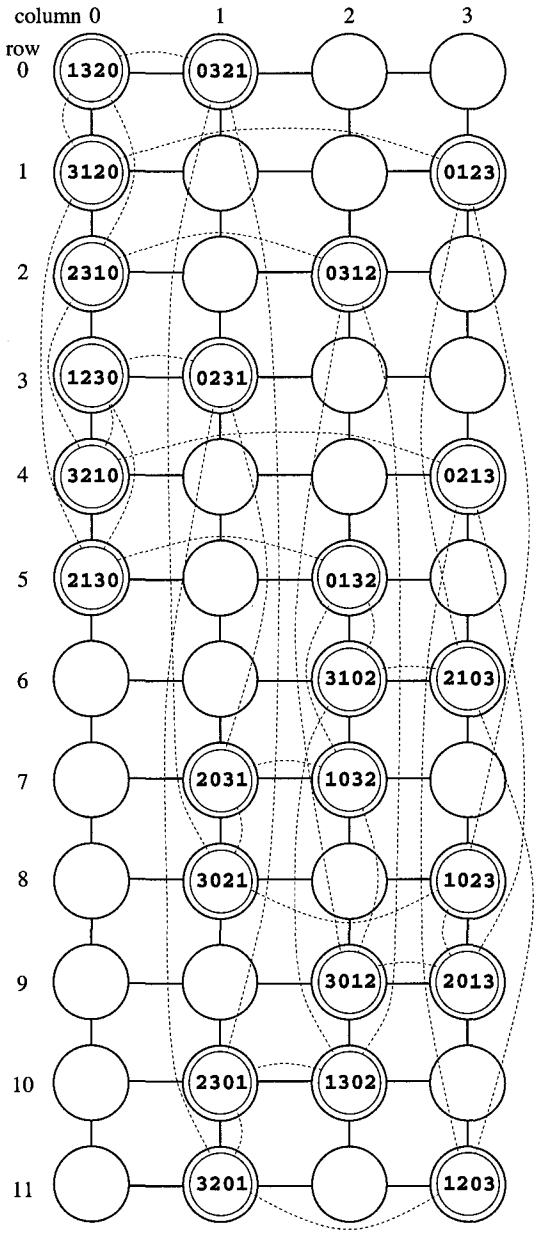


Figure 5.7: Unoptimized embedding of the 4-star into a 4×12-grid as yielded by theorem 1.

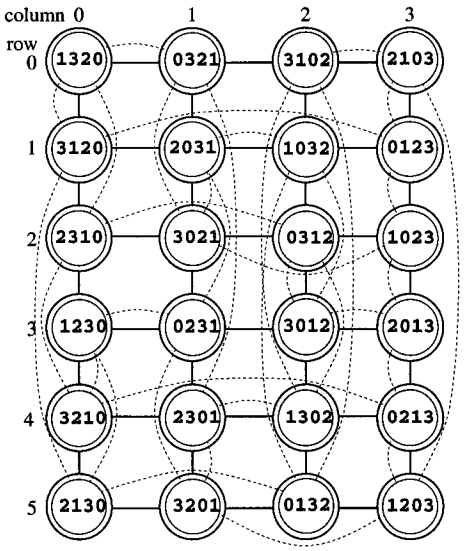


Figure 5.8: Group-optimized embedding of the 4-Star in a 2-d grid.

group Γ such that

$$G_{g_1 g_2 \ldots g_{n-d}} \in \Gamma_{\{\gamma_1, \gamma_2, \ldots, \gamma_{n-d}\}}$$
$$\Leftrightarrow (g_1, g_2, \ldots, g_{n-d}) \text{ is a permutation of } \{\gamma_1, \gamma_2, \ldots, \gamma_{n-d}\}. \quad (5.15)$$

Each meta group Γ holds $|\Gamma| = (n-d)!$ groups.

Groups that differ in two or more symbols can share a row, cf. proof of theorem 2. Thus, we construct a **meta group graph** $M_{n,d} = (V_M, E_M)$ in which nodes represent meta groups and an edge is present whenever two meta groups differ in at least two symbols:

$$V_M = \{v : v \text{ is a set of } n-d \text{ symbols from } \mathcal{S}\} \quad (5.16)$$

and

$$E_M = \{\{u, v\} : |u \cap v| \leq n-d-2\}. \quad (5.17)$$

Now, we show how we can use a method called **clique-partitioning** and a meta group graph to group-optimize the embedding of a star graph. Note, that a meta group graph $M_{n,d} = (V_M, E_M)$ can always be partitioned into c cliques for some $c \leq |V_M|$.

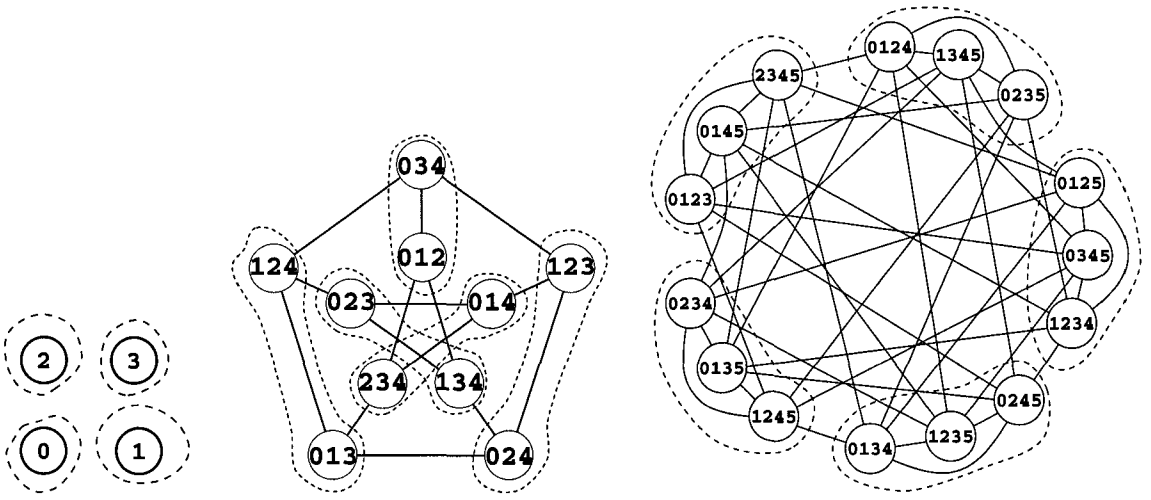


Figure 5.9: Clique partitionings of $M_{4,3}$, $M_{5,2}$, and $M_{6,2}$.

Theorem 2 *Given a clique-partitioning of a meta group graph $M_{n,d}$ into c cliques, we can embed an n-star into a d-dimensional grid of size $\underbrace{n \times \cdots \times n}_{d-1} \times c \cdot (n-d)!$ with load one and no bends.*

Proof A clique partitioning of an $M_{n,d}$ is a partitioning of the nodes of $M_{n,d}$ into cliques, i.e., complete subgraphs, such that every node is a member of exactly one of these cliques.

Take any two groups G_1 and G_2 whose meta groups are neighbours in $M_{n,d}$. G_1 and G_2's symbols differ in at least two. Consequently, their respective clusters differ in at least one symbol (recall: $s_1 \underbrace{s_2 \ldots s_{n-d+1}}_{\text{group}} \underbrace{s_{n-d+2} \ldots s_n}_{\textit{cluster}}$) and so members of G_1 and G_2 can never occupy the same grid node even if placed onto the same row.

In a clique, all meta groups are neighbours, and hence we can embed one group from each meta group in a clique into the same row. There are c cliques and $(n-d)!$ groups per meta group. Thus, we can embed the n-star in a d-dimensional grid of size $\underbrace{n \times \cdots \times n}_{d-1} \times c \cdot (n-d)!$ with load one.□

We observe that the expansion e is

$$e = \frac{n^{d-1} \cdot c(n-d)!}{n!}. \tag{5.18}$$

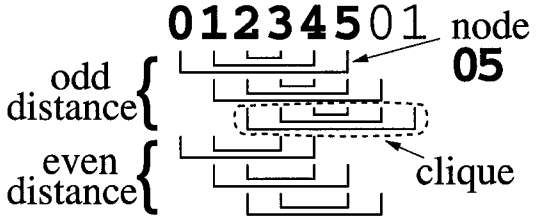


Figure 5.10: A clique-partitioning of $M_{6,4}/M_{6,2}$.

It is desirable to partition $M_{n,d}$ into as few cliques as possible in order to minimize the expansion. However, clique partitioning is a well known and hard problem. Algorithms for clique partitioning exist, and a survey can be found in [Pul82]. Optimal partitionings of $M_{4,3}$, $M_{5,2}$, and $M_{6,2}$ are shown in figure 5.9.

In two dimensions, we can determine an upper bound on the expansion by exploiting symmetry in the meta group graph $M_{n,2}$. First, we show how to partition an $M_{n,2}$.

Lemma 3 *An $M_{n,2}$ can be partitioned into n cliques if n is even, and $n+1$ cliques if n is odd.*

Proof The labels of $M_{n,2}$ consist of $n-2$ symbols. For simplicity, we may alternatively identify an $M_{n,2}$ node by the two symbols that are missing from its label. For example, node 0145 in $M_{6,2}$ can be uniquely identified by its missing symbols 23. Using these new labels, $M_{n,2}$ becomes $M_{n,n-2}$. In fact, this symmetry holds for all d and clique-partitioning an $M_{n,d}$ is equivalent to clique-partitioning an $M_{n,n-d}$.

In an $M_{n,n-2}$, neighbours have no symbols in common. We select cliques by choosing pairs of symbols, corresponding to $M_{n,n-2}$ node labels, that do not overlap. For instance, symbol pairs 01, 23, and 45 form a clique. Let us define the distance between two symbols s_1 and s_2 as the smallest integer Δ such that $s_2 = (s_1 + \Delta) \bmod n$. Every symbol needs to pair up with $\lceil \frac{n-1}{2} \rceil$ odd, and $\lfloor \frac{n-1}{2} \rfloor$ even distance symbols in order to form the $\binom{n}{2}$ meta group graph node labels.

When n is even, we pair up symbols at odd distances $n-1, n-3, \ldots, 1$ to form $n/2$ cliques that cover all the odd distance symbol pairings. Similarly, we create $n/2$

Star		2-dimensional grid					
n	Nodes	unoptimized		group optimized		contracted	
	$n!$	size	expansion	size	expansion	size	load
4	24	4×12	2	4×6	1	4×3	2
5	120	5×60	$2\frac{1}{2}$	5×30	$1\frac{1}{4}$	5×5	6
6	720	6×360	3	6×120	1	6×5	24
7	5040	7×2520	$3\frac{1}{2}$	7×840	$1\frac{1}{6}$	7×7	120
8	40320	8×20160	4	8×5040	1	8×7	720
9	362880	9×181440	$4\frac{1}{2}$	9×45360	$1\frac{1}{8}$	9×9	5040

Table 5.2: Star graph embeddings in two dimensions.

cliques to cover all the even symbols pairings. See figure 5.10 for an example. This way, we partition $M_{n,n-2}$ and consequently $M_{n,2}$ with n cliques.

When n is odd, we introduce a *ghost* symbol $n+1$ to make the number of symbols even, and then proceed as in the even case. □

Now, we use lemma 3 to get an upper bound on the expansion.

Theorem 4 *A group-optimized unity-load embedding with no bends of an n-star into a two dimensional grid has an expansion of no more than* $1\frac{1}{n-1}$ *if n is even, and* $1\frac{2}{n-1}$ *if n is odd.*

Proof: The proof follows from lemma 3 and the formula for expansion (5.18) where the number of cliques c is n or $n+1$ when n is even or odd, respectively.□

Theorem 4 guarantees us that we can always find a near unity expansion embedding in two dimensions. In tables 5.2 and 5.3, we have compiled parameters of actual embeddings. Table 5.2 obeys the expansion bound of theorem 4 and even suggests a tighter actual bound of unity if n is even, and $1\frac{1}{n-1}$ if n is odd. Table 5.3 suggests that the expansion also approaches unity as the star graph size increases. Figures 5.11 and 5.12 show example embeddings of the 5-star in two dimensions, and the 4-star in three dimensions.

In the next section we will show how the high aspect ratio of the larger embeddings can be reduced to yield aspect ratios of approximately unity.

Star		3-dimensional grid					
n	Nodes	unoptimized		group optimized		contracted	
	$n!$	size	expansion	size	expansion	size	load
4	24	$4 \times 4 \times 4$	$2\frac{2}{3}$	$4 \times 4 \times 4$	$2\frac{2}{3}$	$4 \times 4 \times 4$	1
5	120	$5 \times 5 \times 20$	$4\frac{1}{6}$	$5 \times 5 \times 10$	$2\frac{1}{12}$	$5 \times 5 \times 5$	2
6	720	$6 \times 6 \times 120$	6	$6 \times 6 \times 36$	$1\frac{4}{5}$	$6 \times 6 \times 6$	6
7	5040	$7 \times 7 \times 840$	$8\frac{1}{6}$	$7 \times 7 \times 168$	$1\frac{19}{30}$	$7 \times 7 \times 7$	24
8	40320	$8 \times 8 \times 6720$	$10\frac{2}{3}$	$8 \times 8 \times 840$	$1\frac{1}{3}$	$8 \times 8 \times 7$	120
9	362880	$9 \times 9 \times 60480$	$13\frac{1}{2}$	$9 \times 9 \times 5040$	$1\frac{1}{8}$	$9 \times 9 \times 7$	720

Table 5.3: Star graph embeddings in three dimensions.

	14203	03412	34201	23410
	14023	03142	34021	23140
	12403	04312	32401	24310
	12043	04132	32041	24130
	10423	01342	30421	21340
	10243	01432	30241	21430
23104		43102	03241	13240
23014		43012	03421	13420
21304		41302	02341	12340
21034		41032	02431	12430
20314		40312	04321	14320
20134		40132	04231	14230
13204	04213		43201	34210
13024	04123		43021	34120
12304	02413		42301	32410
12034	02143		42031	32140
10324	01423		40321	31420
10234	01243		40231	31240
03214	24103	34102		43210
03124	24013	34012		43120
02314	21403	31402		42310
02134	21043	31042		42130
01324	20413	30412		41320
01234	20143	30142		41230
32104	42103	13402	23401	
32014	42013	13042	23041	
31204	41203	14302	24301	
31024	41023	14032	24031	
30214	40213	10342	20341	
30124	40123	10432	20431	

Figure 5.11: Embedding of the 5-Star into a two-dimensional grid.

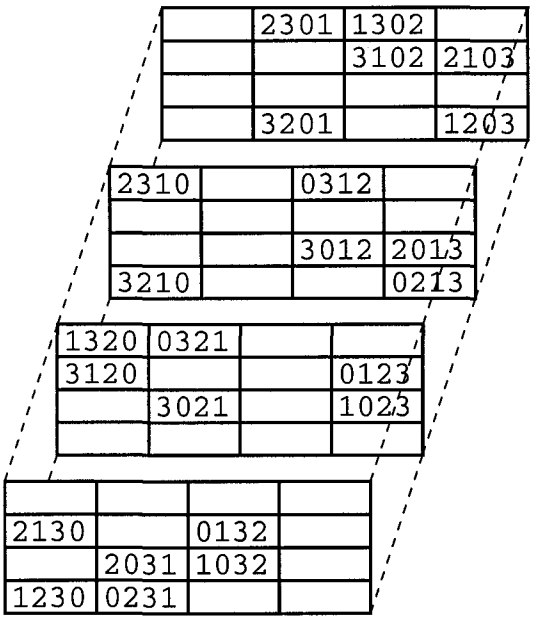


Figure 5.12: 3-d embedding of the 4-Star.

5.3.3.3 Contraction

For large star graphs, the high aspect ratio of the grid, e.g. 8×5040 for the 8-star in 2 dimensions, may complicate physical implementation. In this section, we will show how we can use contraction to efficiently improve the aspect ratio, and thus the dilation of the embedding by reducing the number of rows in the embedding to be equal to the number of cliques in a clique-partitioned meta group graph. In two dimensional embeddings, we will have no more than $n+1$ rows, implying a dilation of at most n.

We can reduce the aspect ratio by contracting several nodes into one, thus increasing the load. Two caveats have to be avoided, though. For one, contraction should not increase the degree of a node. Otherwise, the aspect ratio problem would merely be shifted from a large-size problem to a large-degree problem [BS96]. Secondly, contracted nodes should not have internal edges. If internal edges were present, potentially slow electrical intra-node connections might have to replace otherwise fast optical inter-node links.

A solution to the second caveat in the contraction problem can once again be found in the meta group graph introduced in section 5.3.3.2. Every star graph node is part of a group and meta group, which in turn make up the meta group graph, and within a meta group, all star graph nodes of a given cluster are isolated.

Theorem 5 *Star Graph nodes of the same cluster have no neighbours in their meta group.*

Proof: Let nodes $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ be any two distinct members of the same cluster and meta group. Then $(u_2, \ldots, u_n)$ is a permutation of $(v_2, \ldots, v_n)$. Consequently, $u_1 = v_1$, and u and v cannot be neighbours in the star graph. □.

By theorem 5, we can contract all $(n-d)!$ star graph nodes of the same cluster and meta group into one grid node without introducing any potentially slow electrical intra-node connections. Further, we can show that although contraction may increase the load of the embedding substantially, the degree of the host node will remain at $n-1$, i.e., the degree of the star graph.

Theorem 6 *The degree of a contracted node containing all nodes of a meta group for a given cluster is equal to the degree of the individual nodes.*

Proof For each $i = 1, \ldots, (n-d)!$, let u^i denote a subset of the nodes in a meta group $\Gamma_{\{\gamma_1,\ldots,\gamma_{n-d}\}}$ which share the same cluster. All nodes u^i are embedded in the contracted node u. For example in figure 5.13, contracted node $u = (3,0,1,2,4)$ in the upper right corner holds nodes $u^1 = (3,0,1,2,4)$, $u^2 = (3,0,2,1,4)$, $u^3 = (3,1,0,2,4)$, $u^4 = (3,1,2,0,4)$, $u^5 = (3,2,1,0,4)$, and $u^6 = (3,2,0,1,4)$. These u^i are all part of cluster C_4 and members of meta group $\Gamma_{\{0,1,2\}}$ We label u by $(u_1, \gamma_1, \ldots, \gamma_{n-d}, u_{n-d+2}, \ldots, u_n)$. We observe that for all $i = 1, \ldots, (n-d)!$, $u^i_1 = u_1$, and $(u^i_{n-d+2}, \ldots, u^i_n) = (u_{n-d+2}, \ldots, u_n)$. The cluster of all nodes in u is $C_{u_{n-d+2},\ldots,u_n}$. Within $C_{u_{n-d+2},\ldots,u_n}$, every u^i connects to a node with the same label only with symbols u^i_1 and γ_k, where $k = 1, \ldots, n-d$, exchanged. Since $u^i_1 = u^j_1 = u_1$ for all nodes in $\Gamma_{\{\gamma_1,\ldots,\gamma_{n-d}\}}$ and in cluster $C_{u_{n-d+2},\ldots,u_n}$, all γ_k-symbol edges of the nodes in u will connect to contracted node v of meta group $\Gamma_{\{u_1,\gamma_1,\ldots,\gamma_{n-d}\}-\{\gamma_k\}}$ in cluster $C_{u_{n-d+2},\ldots,u_n}$. Intra-group edges will also connect to the same contracted node in a different cluster since the neighbours meta group remains $\Gamma_{\{\gamma_1,\ldots,\gamma_{n-d}\}}$. □

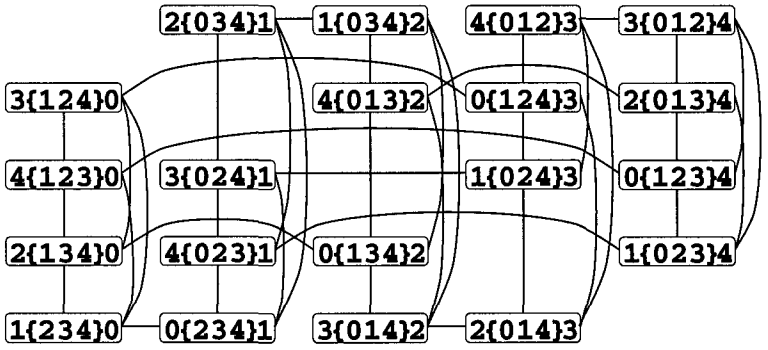


Figure 5.13: Contracted embedding of a 5-Star in two dimensions.

```
input: n,d          dimensions of star and mesh
       contraction  'true' if contraction is desired

Embed(n,d,contraction)
    Q=clique-partitioning of M_{n,d}
    row=0
    foreach clique C in Q
        foreach meta group Γ in C
            thisRow=row
            foreach group G in Γ
                foreach permutation (p_1,...,p_d) of {0,...,n-1}-{γ_1,...,γ_{n-d}}
                    guestnode of (p_2,...,p_d ,thisRow)=(p_1,g_1,...,g_{n-d} ,p_2,...,p_d)
                if contraction=false then
                    thisRow = thisRow+1
        if contraction=false then
            row=row+(n-d)!
        else
            row=row+1
```

Figure 5.14: Group-optimized embedding algorithm with optional contraction.

Since we contract all nodes of the same cluster in one meta group into one contracted node, we only require as many rows for our embeddings as there are cliques in a clique partitioned meta group graph. Thus, in two dimensions, by lemma 3, we require no more than $n + 1$ rows.

A contracted embedding of the 5-star in two dimensions is shown in figure 5.13. The aspect ratio of the embedding is now reduced from 30 : 5 to 5 : 5 with a load of $(n - d)! = 6$. The degree of the contracted nodes is still $n - 1 = 4$.

Tables 5.2 and 5.3 show parameters of contracted embeddings for star graphs of up to 9! nodes. Figure 5.14 shows the algorithm for group-optimized embeddings with optional contraction.

5.3.4 Conclusion

Star graphs can be embedded in rectangular grids in up to $n - 1$ dimensions without bends, thus allowing for one-hop optical communication between star graph neighbours in a rectangular implementation.

These embeddings exhibit load 1, and expansion of at most $n^{d-1}/d!$. Using clique-partitioning, we were able to reduce the size of the host graph and produce embeddings with expansions as low as unity for 2-dimensional embeddings and close to unity in three-dimensional embeddings. In general, the expansion of a group optimized two-dimensional embedding is guaranteed not to exceed $1\frac{2}{n-1}$.

By increasing the load in a contraction process, we were able to reduce the aspect ratio to values near unity without requiring potentially slow electrical intra-node edges. Further, we managed to keep the degree of a contracted node at $n - 1$, the degree of the embedded n-star.

After Latifi and Bagherzadeh [LB94] have overcome the scalability problem on the n!-node star graph, our embedding procedure eliminates one more obstacle that has hindered the practical use of star graphs.

Chapter 6

Conclusion

With this dissertation, we provided a discussion of fast placement methods of hypergraphs into grids.

In chapter 1, we explained why hypergraph placement into grid structures is an important problem for modern electronic and networking technology, and we provided a collection of the most relevant underlying combinatorial placement problems. Then in chapter 2.1, we showed how the grid model for placement was derived from a number of modern electronics technologies and optical networking architectures. In particular, we detailed how the wire-length placement problem, cf. problem 4b on page 25, was important for the placement of electronic circuits into emerging three-dimensional VLSI as well as traditional 2-D VLSI, and programmable VLSI. Further, we showed how the problem of finding straight line placements, cf. problem 5 on page 27, was a consideration for some optical architectures in which straight line communication was considerably faster than communication that needs to be routed between intersecting optical buses.

In chapter 3, we summarised previous approaches to solving the wire-length placement problem. Among the most successful methodologies were placement by simulated annealing, which is proven to converge to an optimal solution, placement by recursive partitioning, and force-directed placement which emulates a set of attractive and repulsive forces acting on nodes.

In chapter 4, we provided a detailed exposition of a new force directed wire-length placement algorithm, which we termed Gravity, that produces good results using

near-linear run time with respect to the input circuit size. Gravity was designed to produce good results with effectively linear run times so that it could cope with large circuit sizes that are characteristic of modern VLSI. For this purpose, and for the lack of existing benchmark results for large circuits, we compared Gravity's results with the results of a generic partitioning placer using the currently best fast partitioner available. In two dimensions, Gravity used an average 8% less wire length and 1/17 the run time of the partitioning placer. In three dimensions, Gravity required on the average 20% less wire length, and 1/13 the run time. Thus, Gravity is a fast algorithm that produces placements which result in fast circuits due to shorter wire length. Compared with standard cell placement of a small number of older, smaller benchmark circuits using a state-of-the-art standard-cell-placement algorithm[EJ98], Gravity produced weaker results with 13% longer wire lengths on the average. However, Gravity does not have a good standard cell final placement routine, and Gravity did outperform the state-of-the-art standard cell placer by 1% average wire length on circuits which had a pin distribution closer to the ones typical for the larger, more modern benchmark circuits.

Further, we provided a comparison of wire-length savings when moving from two to three dimensions. For the largest circuits, Gravity's placement required an estimated 50% savings using only a small number of layers in the third dimension, and approximately 70% less wire length in a fully three dimensional placement.

Finally, in chapter 5, we suggested how some popular traditional networking topologies, i.e., the torus, the tree, and the hypercube, can be placed into a grid without bending edges by simple transformations. Then we showed how a modern topology, the star graph, which has many superior topological features can also be placed bendlessly into arbitrary grids. We showed how such embeddings can be optimized in size and aspect ratio.

In focus of this dissertation were fast placement methods that have low run-time complexities, and that resulted in fast operation of the underlying technology, be it fast straight-line communication, or highly-clocked electronic circuits benefitting from reduced size by conserving wire length.

6.1 Originality

Original research was presented in modest part in chapter 1, wholly in chapter 4, and building on results we derived in [Obe95] in chapter 5.

The generalization of several combinatorial problems to hypergraphs and multi-dimensional grids was original research which was published in part in [SO98].

The fast placement algorithm of chapter 4, Gravity, was entirely original research, in both the two and three dimensional versions. Since there are no published wire-length results for modern larger electronic circuits, we constructed partitioning placement algorithms for 2-D and 3-D using an existing partitioner. These partitioning placers were used to compute wire-length results for a new benchmark circuit suite with modern large circuits. Without this reference partitioning placer, Gravity's placement results would have been presented in a vacuum. In three dimensions, no placement algorithms of reasonably low run-time complexity, have been published, and thus no placement results for larger circuits are available. In two dimensions, placement results where only available for an older benchmark circuit suite which we believe is no longer representative of modern circuit structures and sizes. 3-D results for the our reference partitioning placer where published in [OS99].

Finally, in chapter 5, we presented a bendless embedding of the star graph. This work was the extension of an idea developed in [Obe95]. We provided an asymptotically tight lower bound on the expansion of a bendless star graph embedding into a two dimensional grid, and we showed how such an embedding can be constructed. Further we showed how the aspect ratio of a bendless star graph embedding can be reduced to near unity by using a contraction method that increases the load, without increasing the degree of the graph, and without introducing node internal edges. This work was published in [OS97].

6.2 Future Work

Future work mainly involves the placement algorithm Gravity. Gravity needs to be made more compatible with current circuit design tools, and its performance may be improved further.

Gravity requires a post processor for efficient standard cell placement, as well as

provisions for recognizing different cell dimensions internally. The former could be a separate algorithm in the form of Domino [DJA94]. The latter can be integrated into Gravity without structural changes to Gravity, by adding cell weights proportional to the cell areas.

Gravity's run time may also be improved through parallelization. By min-cut partitioning the circuit according to the number of processors, force-directed position updates at every iteration can be computed in parallel for different subcircuits. Between iterations, only border node positions would have to be exchanged between processors. Designing an efficient implementation may be a suitable topic for a Master's thesis.

Lastly, hierarchical clusterization may be required in Gravity for very large circuits. Clusterization clusters several cells into one in order to reduce the run time requirement for the initial placement. Clusterization by edge contraction was employed, for example, in the simulated annealing placer Timberwolf [SS97], and in the hMetis partitioner [KAKS97]. Hierarchical clusterization may help with the detection of coarse circuit structures, and thus allow the separation of sparsely interconnected subcircuits which may otherwise overlay on the same placement area.

Appendix A

Detailed Gravity Results

A.1 Results in Two Dimensions

Circuit	hMetis			500 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	450	2.94	422	355	-21.17	2.22	337	-20.20
avq.large	1089	1.86	1059	1176	8.00	3.86	1121	5.87
avq.small	1054	1.30	1039	1071	1.63	3.71	1019	-2.01
baluP	167	2.27	161	145	-13.14	2.01	138	-14.15
biomedP	514	1.19	503	455	-11.55	2.64	437	-13.11
golem3	4720	0.93	4660	6944	47.11	0.83	6848	46.96
industry2	1278	1.77	1223	1297	1.48	1.43	1264	3.39
industry3	2389	1.50	2354	2088	-12.61	2.48	1991	-15.40
p1	214	2.51	204	168	-21.33	1.55	165	-19.28
p2	599	2.15	580	497	-17.01	2.43	477	-17.77
s13207P	401	2.34	377	381	-4.97	1.97	368	-2.23
s15850P	443	1.16	436	434	-2.13	1.32	423	-2.94
s35932	579	1.22	569	746	28.87	2.56	707	24.29
s38417	684	1.36	671	872	27.56	2.33	840	25.10
s38584	757	1.09	740	889	17.45	1.54	870	17.57
s9234P	350	2.27	336	327	-6.60	2.25	311	-7.38
structP	228	3.12	215	187	-18.03	2.56	180	-16.52
t2	333	2.42	317	255	-23.49	1.95	246	-22.26
t3	304	1.93	295	249	-18.06	3.26	238	-19.29
t4	292	2.33	280	239	-18.23	1.83	230	-17.67
t5	390	2.74	381	328	-15.74	2.43	316	-16.98
t6	304	1.95	296	254	-16.56	2.24	243	-18.02
Average		1.93			-4.02	2.24		-4.64

Table A.1: Detailed wire-length comparison of 500-iteration Gravity vs. hMetis for ACM/SIGDA benchmarks, 10 runs.

Circuit	hMetis			500 iterations				
	mean length	% standard deviation	minimum	mean length	% change	% standard deviation	minimum	% change
ibm01	1545	1.41	1507	1331	-13.83	3.38	1263	-16.24
ibm02	3176	2.31	3064	2847	-10.34	2.26	2744	-10.44
ibm03	3557	1.87	3472	3178	-10.64	2.05	3084	-11.19
ibm04	4020	1.55	3942	3436	-14.54	2.58	3257	-17.38
ibm05	5786	1.68	5618	4719	-18.45	2.37	4561	-18.81
ibm06	4224	3.20	4032	3472	-17.80	2.75	3298	-18.22
ibm07	5230	2.48	5027	4506	-13.86	4.57	4206	-16.34
ibm08	5533	1.44	5394	4887	-11.69	3.53	4640	-13.99
ibm09	5046	1.44	4919	4859	-3.70	2.84	4641	-5.64
ibm10	7292	2.10	7006	6546	-10.23	4.26	6073	-13.31
ibm11	6727	2.31	6475	6288	-6.53	2.37	6119	-5.50
ibm12	9168	1.48	8841	8236	-10.17	1.68	7994	-9.58
ibm13	7464	1.03	7324	6929	-7.17	2.98	6626	-9.53
ibm14	10784	3.47	9976	10524	-2.41	2.60	10144	1.67
ibm15	13095	2.68	12502	13212	0.89	2.38	12696	1.56
ibm16	13557	1.81	13236	13369	-1.39	2.73	12749	-3.68
ibm17	17033	3.45	16254	16631	-2.36	2.76	16157	-0.60
ibm18	12283	2.01	11820	13461	9.59	2.74	12670	7.19
Average		2.10			-8.03	2.82		-8.89

Table A.2: Detailed wire-length comparison of 500-iteration Gravity vs. hMetis for ISPD98 benchmarks, 10 runs.

Circuit	hMetis			1000 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	450	2.94	422	350	-22.30	1.17	340	-19.45
avq.large	1089	1.86	1059	1002	-7.99	4.70	944	-10.85
avq.small	1054	1.30	1039	955	-9.39	2.82	909	-12.52
baluP	167	2.27	161	143	-14.47	1.65	139	-13.61
biomedP	514	1.19	503	450	-12.52	2.27	433	-13.95
golem3	4720	0.93	4660	6306	33.60	0.97	6193	32.91
industry2	1278	1.77	1223	1288	0.74	2.73	1247	1.98
industry3	2389	1.50	2354	2030	-15.05	2.33	1967	-16.43
p1	214	2.51	204	166	-22.33	1.27	163	-20.20
p2	599	2.15	580	483	-19.25	2.09	472	-18.71
s13207P	401	2.34	377	360	-10.17	2.95	343	-8.89
s15850P	443	1.16	436	405	-8.74	3.13	389	-10.83
s35932	579	1.22	569	628	8.51	2.25	608	6.89
s38417	684	1.36	671	755	10.36	3.87	710	5.71
s38584	757	1.09	740	815	7.61	3.06	764	3.26
s9234P	350	2.27	336	310	-11.33	2.03	301	-10.36
structP	228	3.12	215	181	-20.81	2.76	169	-21.27
t2	333	2.42	317	253	-24.03	1.11	248	-21.82
t3	304	1.93	295	237	-22.06	2.42	228	-22.76
t4	292	2.33	280	238	-18.39	1.42	233	-16.58
t5	390	2.74	381	328	-15.97	2.49	316	-16.91
t6	304	1.95	296	258	-15.40	2.30	249	-15.82
Average		1.93			-9.52	2.35		-10.01

Table A.3: Detailed wire-length comparison of 1000-iteration Gravity vs. hMetis for ACM/SIGDA benchmarks, 10 runs.

Circuit	hMetis			1000 iterations				
	mean length	% standard deviation	minimum	mean length	% change	% standard deviation	minimum	% change
ibm01	1545	1.41	1507	1305	-15.51	2.56	1274	-15.49
ibm02	3176	2.31	3064	2771	-12.73	2.65	2609	-14.84
ibm03	3557	1.87	3472	3105	-12.71	1.74	3014	-13.21
ibm04	4020	1.55	3942	3291	-18.14	1.61	3200	-18.84
ibm05	5786	1.68	5618	4656	-19.53	1.79	4553	-18.95
ibm06	4224	3.20	4032	3355	-20.55	3.61	3169	-21.42
ibm07	5230	2.48	5027	4320	-17.40	2.46	4150	-17.46
ibm08	5533	1.44	5394	4796	-13.32	1.84	4692	-13.01
ibm09	5046	1.44	4919	4712	-6.61	1.68	4571	-7.06
ibm10	7292	2.10	7006	6282	-13.85	2.76	6069	-13.37
ibm11	6727	2.31	6475	5968	-11.29	2.49	5783	-10.70
ibm12	9168	1.48	8841	8064	-12.04	1.41	7928	-10.33
ibm13	7464	1.03	7324	6708	-10.14	1.89	6510	-11.11
ibm14	10784	3.47	9976	9800	-9.12	1.16	9640	-3.38
ibm15	13095	2.68	12502	12494	-4.59	2.47	12144	-2.86
ibm16	13557	1.81	13236	13034	-3.86	2.05	12639	-4.51
ibm17	17033	3.45	16254	15681	-7.94	2.23	15175	-6.64
ibm18	12283	2.01	11820	12809	4.28	3.04	11807	-0.11
Average		2.10			-11.39	2.19		-11.29

Table A.4: Detailed wire-length comparison of 1000-iteration Gravity vs. hMetis for ISPD98 benchmarks, 10 runs.

Circuit	hMetis			2000 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	450	2.94	422	343	-23.68	1.89	329	-22.11
avq.large	1089	1.86	1059	915	-15.94	3.21	855	-19.28
avq.small	1054	1.30	1039	869	-17.54	3.54	833	-19.82
baluP	167	2.27	161	140	-16.16	1.63	136	-15.32
biomedP	514	1.19	503	441	-14.27	1.75	431	-14.38
golem3	4720	0.93	4660	5888	24.74	1.42	5702	22.38
industry2	1278	1.77	1223	1286	0.63	2.26	1234	0.95
industry3	2389	1.50	2354	2004	-16.13	1.60	1954	-16.98
p1	214	2.51	204	166	-22.51	1.86	160	-21.27
p2	599	2.15	580	492	-17.86	2.59	472	-18.59
s13207P	401	2.34	377	351	-12.39	1.92	339	-9.90
s15850P	443	1.16	436	394	-11.06	2.07	386	-11.45
s35932	579	1.22	569	562	-3.00	1.99	545	-4.25
s38417	684	1.36	671	709	3.72	1.62	687	2.37
s38584	757	1.09	740	790	4.35	2.61	762	2.99
s9234P	350	2.27	336	305	-12.93	1.65	297	-11.61
structP	228	3.12	215	172	-24.69	1.93	168	-21.76
t2	333	2.42	317	251	-24.48	1.33	247	-22.19
t3	304	1.93	295	238	-21.87	2.37	230	-21.93
t4	292	2.33	280	236	-19.15	1.82	229	-18.15
t5	390	2.74	381	331	-15.17	1.87	320	-15.81
t6	304	1.95	296	254	-16.42	1.52	250	-15.65
Average		1.93			-12.36	2.02		-12.35

Table A.5: Detailed wire-length comparison of 2000-iteration Gravity vs. hMetis for ACM/SIGDA benchmarks, 10 runs.

Circuit	hMetis			2000 iterations				
	mean length	% standard deviation	minimum	mean length	% change	% standard deviation	minimum	% change
ibm01	1545	1.41	1507	1293	-16.29	2.95	1234	-18.15
ibm02	3176	2.31	3064	2717	-14.43	1.83	2623	-14.40
ibm03	3557	1.87	3472	3083	-13.31	1.97	2987	-13.97
ibm04	4020	1.55	3942	3279	-18.45	1.59	3203	-18.74
ibm05	5786	1.68	5618	4675	-19.21	1.76	4526	-19.43
ibm06	4224	3.20	4032	3306	-21.73	3.45	3137	-22.20
ibm07	5230	2.48	5027	4305	-17.69	3.04	4081	-18.82
ibm08	5533	1.44	5394	4688	-15.27	1.96	4506	-16.47
ibm09	5046	1.44	4919	4609	-8.67	1.54	4462	-9.29
ibm10	7292	2.10	7006	6180	-15.25	2.25	5914	-15.59
ibm11	6727	2.31	6475	5863	-12.84	1.70	5732	-11.47
ibm12	9168	1.48	8841	8026	-12.46	1.75	7744	-12.41
ibm13	7464	1.03	7324	6668	-10.67	1.44	6448	-11.96
ibm14	10784	3.47	9976	9640	-10.60	1.24	9435	-5.43
ibm15	13095	2.68	12502	12295	-6.11	2.56	11771	-5.84
ibm16	13557	1.81	13236	12524	-7.62	1.11	12210	-7.75
ibm17	17033	3.45	16254	15520	-8.88	2.68	14707	-9.52
ibm18	12283	2.01	11820	12356	0.59	1.67	12002	1.54
Average		2.10			-12.72	2.03		-12.77

Table A.6: Detailed wire-length comparison of 2000-iteration Gravity vs. hMetis for ISPD98 benchmarks, 10 runs.

Circuit	hMetis	500 iterations		1000 iterations		2000 iterations	
	time(s)	time(s)	speed-up	time(s)	speed-up	time(s)	speed-up
19ks	57.9	2.8	21.0	4.7	12.3	8.8	6.6
avq.large	487.2	31.6	15.4	57.8	8.4	111.1	4.4
avq.small	436.3	27.1	16.1	49.3	8.8	95.5	4.6
baluP	14.2	0.7	20.9	1.3	11.4	2.5	5.7
biomedP	114.1	7.0	16.4	12.7	9.0	24.4	4.7
golem3	2189.8	169.7	12.9	288.2	7.6	525.0	4.2
industry2	277.5	19.1	14.5	32.6	8.5	57.3	4.8
industry3	383.3	23.5	16.3	42.4	9.0	82.2	4.7
p1	14.9	0.9	17.3	1.6	9.4	2.8	5.3
p2	64.6	3.6	17.8	6.3	10.3	11.1	5.8
s13207P	134.3	8.7	15.4	15.8	8.5	30.0	4.5
s15850P	169.3	11.1	15.2	20.7	8.2	38.7	4.4
s35932	314.8	20.3	15.5	36.3	8.7	70.4	4.5
s38417	401.6	30.2	13.3	59.1	6.8	118.8	3.4
s38584	388.6	29.5	13.2	54.7	7.1	103.0	3.8
s9234P	92.8	5.5	17.0	9.9	9.4	18.4	5.1
structP	32.1	1.5	20.7	2.7	12.0	4.8	6.7
t2	30.5	1.6	19.1	2.8	10.8	5.3	5.8
t3	30.6	1.6	18.8	2.9	10.5	5.4	5.7
t4	31.3	1.7	18.1	3.1	10.2	5.6	5.6
t5	55.4	3.2	17.3	5.3	10.3	10.1	5.5
t6	32.8	1.9	17.7	3.0	10.9	5.7	5.8
Average			16.8		9.5		5.1

Table A.7: CPU-time comparison of Gravity vs. hMetis for ACM/SIGDA benchmarks, 10 runs.

Circuit	hMetis	500 iterations		1000 iterations		2000 iterations	
	time(s)	time(s)	speed-up	time(s)	speed-up	time(s)	speed-up
ibm01	320.0	17.3	18.5	31.1	10.3	59.3	5.4
ibm02	575.9	29.2	19.7	51.3	11.2	96.3	6.0
ibm03	626.3	35.8	17.5	64.4	9.7	121.1	5.2
ibm04	685.0	44.7	15.3	80.1	8.6	149.6	4.6
ibm05	847.5	43.8	19.4	76.2	11.1	142.0	6.0
ibm06	921.7	54.2	17.0	96.4	9.6	182.4	5.1
ibm07	1347.1	79.1	17.0	141.8	9.5	268.3	5.0
ibm08	1585.9	94.0	16.9	160.1	9.9	284.2	5.6
ibm09	1495.1	112.3	13.3	202.4	7.4	381.8	3.9
ibm10	2266.7	128.3	17.7	231.9	9.8	433.7	5.2
ibm11	2065.0	135.2	15.3	247.2	8.4	467.0	4.4
ibm12	2428.0	141.1	17.2	234.5	10.4	421.7	5.8
ibm13	2687.4	161.3	16.7	294.5	9.1	557.6	4.8
ibm14	4745.9	269.3	17.6	481.7	9.9	916.9	5.2
ibm15	5850.4	330.0	17.7	599.6	9.8	1141.0	5.1
ibm16	6763.3	362.1	18.7	648.0	10.4	1245.5	5.4
ibm17	7680.3	385.0	20.0	681.6	11.3	1251.4	6.1
ibm18	7320.2	387.4	18.9	695.2	10.5	1304.6	5.6
Average			17.5		9.8		5.2

Table A.8: CPU-time comparison of Gravity vs. hMetis for ISPD98 renchmarks, 10 runs.

A.2 Results in Three Dimensions

Circuit	hMetis			250 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	14493.3	1.61	14123	11613.2	-19.87	1.80	11293	-20.04
avq.large	104104.1	1.22	101693	88972.1	-14.54	1.09	86868	-14.58
avq.small	94688	0.88	93823	79836.9	-15.68	1.32	78204	-16.65
baluP	3263.5	1.77	3164	2739.8	-16.05	1.09	2694	-14.85
biomedP	25239.2	0.80	24839	21937	-13.08	1.35	21457	-13.62
golem3	687104.9	0.68	679554	662483.2	-3.58	0.83	655517	-3.54
industry2	78997.7	0.92	78052	73438.1	-7.04	1.55	71884	-7.90
industry3	152962.3	1.05	149592	124871.3	-18.36	1.09	123098	-17.71
p1	4156.1	2.17	4071	3420.6	-17.70	1.66	3340	-17.96
p2	18562.5	1.64	18044	15685.2	-15.50	1.98	15004	-16.85
s13207P	26501.3	2.15	25617	22972	-13.32	0.75	22649	-11.59
s15850P	30950.7	0.65	30729	28190.1	-8.92	1.20	27496	-10.52
s35932	57926.1	1.74	56120	55075.2	-4.92	0.89	54432	-3.01
s38417	73282.6	1.21	72355	75072.1	2.44	1.00	73989	2.26
s38584	72643.9	1.10	71560	71116.1	-2.10	0.70	70501	-1.48
s9234P	17670.1	0.89	17463	16454.4	-6.88	0.78	16236	-7.03
structP	7064.1	1.41	6879	6122.3	-13.33	0.97	6042	-12.17
t2	8501.9	1.44	8337	6846.3	-19.47	1.60	6661	-20.10
t3	7828.2	1.42	7658	6681.6	-14.65	0.96	6549	-14.48
t4	7375.9	1.63	7242	6499.1	-11.89	1.19	6350	-12.32
t5	12568.2	0.57	12481	11015.9	-12.35	1.56	10778	-13.64
t6	7968.1	1.51	7785	6831.3	-14.27	0.97	6724	-13.63
Average		1.29			-11.87	1.20		-11.88

Table A.9: Detailed wire-length comparison of 250-iteration 3-D Gravity vs. hMetis for the ACM/SIGDA benchmark suite, 10 runs.

Circuit	hMetis			250 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
ibm01	92601.5	0.91	90591	78560.8	-15.16	0.81	78033	-13.86
ibm02	202121.7	0.65	199264	170906.7	-15.44	1.14	167051	-16.17
ibm03	234600.9	0.94	231824	196508.4	-16.24	1.14	193369	-16.59
ibm04	301324.1	1.07	297370	232650.4	-22.79	1.02	229315	-22.89
ibm05	367004.5	1.27	360659	279691.6	-23.79	1.06	273955	-24.04
ibm06	333985	3.21	323919	261922.1	-21.58	1.53	253716	-21.67
ibm07	473844.3	1.82	458047	378412	-20.14	1.36	369334	-19.37
ibm08	531860.7	1.38	521203	412004.1	-22.54	2.31	398982	-23.45
ibm09	617201.7	2.56	587107	472501.7	-23.44	1.48	461762	-21.35
ibm10	832125.1	3.79	796763	647698.6	-22.16	1.61	632117	-20.66
ibm11	872118.2	2.72	842491	643165.8	-26.25	1.34	629650	-25.26
ibm12	991783.9	1.63	959567	785136.4	-20.84	1.34	766362	-20.13
ibm13	1000941.8	2.05	971330	804966.6	-19.58	0.69	796481	-18.00
ibm14	1657408.1	1.12	1630502	1366391.4	-17.56	1.41	1335308	-18.10
ibm15	1994685.8	1.38	1957427	1784492.6	-10.54	0.65	1767102	-9.72
ibm16	2222138	1.05	2190510	1904852.6	-14.28	1.66	1855758	-15.28
ibm17	2745042.7	1.16	2680986	2244711	-18.23	0.81	2219649	-17.21
ibm18	2639356.6	4.29	2504083	1989565.2	-24.62	1.08	1938687	-22.58
Average		1.83			-19.73	1.25		-19.24

Table A.10: Detailed wire-length comparison of 250-iteration 3-D Gravity vs. hMetis for the ISPD98 benchmark suite, 10 runs.

Circuit	hMetis			500 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	14493.3	1.61	14123	11456.9	-20.95	0.71	11308	-19.93
avq.large	104104.1	1.22	101693	82725.1	-20.54	1.29	80963	-20.38
avq.small	94688	0.88	93823	75536.1	-20.23	0.88	74522	-20.57
baluP	3263.5	1.77	3164	2740.8	-16.02	0.90	2703	-14.57
biomedP	25239.2	0.80	24839	21544.6	-14.64	1.03	21082	-15.13
golem3	687104.9	0.68	679554	602956.9	-12.25	0.59	595765	-12.33
industry2	78997.7	0.92	78052	73435.2	-7.04	1.10	72168	-7.54
industry3	152962.3	1.05	149592	123694.6	-19.13	0.77	122626	-18.03
p1	4156.1	2.17	4071	3427.2	-17.54	0.88	3363	-17.39
p2	18562.5	1.64	18044	15515.3	-16.42	1.18	15149	-16.04
s13207P	26501.3	2.15	25617	22278	-15.94	0.44	22125	-13.63
s15850P	30950.7	0.65	30729	27187.1	-12.16	0.85	26909	-12.43
s35932	57926.1	1.74	56120	51887.5	-10.42	0.77	51354	-8.49
s38417	73282.6	1.21	72355	71251.7	-2.77	1.17	70101	-3.12
s38584	72643.9	1.10	71560	69189.5	-4.76	0.89	68074	-4.87
s9234P	17670.1	0.89	17463	16175.4	-8.46	0.66	15979	-8.50
structP	7064.1	1.41	6879	5998.4	-15.09	1.34	5814	-15.48
t2	8501.9	1.44	8337	6735.5	-20.78	1.01	6641	-20.34
t3	7828.2	1.42	7658	6669.8	-14.80	0.75	6591	-13.93
t4	7375.9	1.63	7242	6476.2	-12.20	1.02	6339	-12.47
t5	12568.2	0.57	12481	11099	-11.69	1.77	10731	-14.02
t6	7968.1	1.51	7785	6836.3	-14.20	1.29	6694	-14.01
Average		1.29			-14.00	0.97		-13.78

Table A.11: Detailed wire-length comparison of 500-iteration 3-D Gravity vs. hMetis for the ACM/SIGDA benchmark suite, 10 runs.

Circuit	hMetis			500 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
ibm01	92601.5	0.91	90591	77424.2	-16.39	0.64	76673	-15.36
ibm02	202121.7	0.65	199264	170217.9	-15.78	1.13	167303	-16.04
ibm03	234600.9	0.94	231824	191398.2	-18.42	0.54	190048	-18.02
ibm04	301324.1	1.07	297370	230593.3	-23.47	0.58	228139	-23.28
ibm05	367004.5	1.27	360659	274860.4	-25.11	1.16	271107	-24.83
ibm06	333985	3.21	323919	257418.3	-22.93	1.23	249835	-22.87
ibm07	473844.3	1.82	458047	369258.2	-22.07	0.95	364914	-20.33
ibm08	531860.7	1.38	521203	406123.5	-23.64	1.35	394683	-24.27
ibm09	617201.7	2.56	587107	466055.6	-24.49	0.67	460634	-21.54
ibm10	832125.1	3.79	796763	627657.9	-24.57	0.85	619369	-22.26
ibm11	872118.2	2.72	842491	629399.5	-27.83	0.69	621890	-26.18
ibm12	991783.9	1.63	959567	782332.6	-21.12	1.07	768835	-19.88
ibm13	1000941.8	2.05	971330	789944.2	-21.08	0.67	783356	-19.35
ibm14	1657408.1	1.12	1630502	1323871.9	-20.12	1.27	1293537	-20.67
ibm15	1994685.8	1.38	1957427	1729273.6	-13.31	0.77	1710715	-12.60
ibm16	2222138	1.05	2190510	1843830.2	-17.02	1.28	1785091	-18.51
ibm17	2745042.7	1.16	2680986	2187596.3	-20.31	0.81	2152381	-19.72
ibm18	2639356.6	4.29	2504083	1926444.7	-27.01	1.25	1892589	-24.42
Average		1.83			-21.37	0.94		-20.56

Table A.12: Detailed wire-length comparison of 500-iteration 3-D Gravity vs. hMetis for the ISPD98 benchmark suite, 10 runs.

Circuit	hMetis			1000 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	14493.3	1.61	14123	11365.3	-21.58	0.93	11234	-20.46
avq.large	104104.1	1.22	101693	79211.4	-23.91	0.53	78570	-22.74
avq.small	94688	0.88	93823	72251.2	-23.70	0.96	70508	-24.85
baluP	3263.5	1.77	3164	2718.6	-16.70	0.99	2684	-15.17
biomedP	25239.2	0.80	24839	21308.6	-15.57	0.85	20973	-15.56
golem3	687104.9	0.68	679554	573173.2	-16.58	0.71	566362	-16.66
industry2	78997.7	0.92	78052	73598.3	-6.83	0.97	72397	-7.25
industry3	152962.3	1.05	149592	123638.6	-19.17	0.83	121940	-18.48
p1	4156.1	2.17	4071	3404.3	-18.09	1.97	3297	-19.01
p2	18562.5	1.64	18044	15596.9	-15.98	1.32	15361	-14.87
s13207P	26501.3	2.15	25617	21914.5	-17.31	0.56	21742	-15.13
s15850P	30950.7	0.65	30729	26776.8	-13.49	0.50	26572	-13.53
s35932	57926.1	1.74	56120	49519.1	-14.51	0.66	49078	-12.55
s38417	73282.6	1.21	72355	70281.3	-4.10	0.98	69118	-4.47
s38584	72643.9	1.10	71560	68699.7	-5.43	1.07	67563	-5.59
s9234P	17670.1	0.89	17463	16007.3	-9.41	0.69	15816	-9.43
structP	7064.1	1.41	6879	5934.8	-15.99	0.87	5833	-15.21
t2	8501.9	1.44	8337	6674.7	-21.49	0.91	6583	-21.04
t3	7828.2	1.42	7658	6689.6	-14.54	1.31	6544	-14.55
t4	7375.9	1.63	7242	6412.6	-13.06	0.78	6300	-13.01
t5	12568.2	0.57	12481	11160.7	-11.20	1.83	10741	-13.94
t6	7968.1	1.51	7785	6795.3	-14.72	1.33	6646	-14.63
Average		1.29			-15.15	0.98		-14.91

Table A.13: Detailed wire-length comparison of 1000-iteration 3-D Gravity vs. hMetis for the ACM/SIGDA benchmark suite, 10 runs.

Circuit	hMetis			1000 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
ibm01	92601.5	0.91	90591	77198.4	-16.63	0.57	76481	-15.58
ibm02	202121.7	0.65	199264	169815.4	-15.98	0.79	167707	-15.84
ibm03	234600.9	0.94	231824	190025.4	-19.00	0.69	188051	-18.88
ibm04	301324.1	1.07	297370	229038.2	-23.99	0.87	226212	-23.93
ibm05	367004.5	1.27	360659	274843.8	-25.11	0.90	271640	-24.68
ibm06	333985	3.21	323919	253564.6	-24.08	0.75	249904	-22.85
ibm07	473844.3	1.82	458047	363000.5	-23.39	1.27	351606	-23.24
ibm08	531860.7	1.38	521203	402003.2	-24.42	0.88	392906	-24.62
ibm09	617201.7	2.56	587107	461399	-25.24	0.91	455776	-22.37
ibm10	832125.1	3.79	796763	622283	-25.22	0.65	613295	-23.03
ibm11	872118.2	2.72	842491	617703.6	-29.17	0.57	612347	-27.32
ibm12	991783.9	1.63	959567	777615.3	-21.59	1.27	765212	-20.25
ibm13	1000941.8	2.05	971330	782729.9	-21.80	0.66	775145	-20.20
ibm14	1657408.1	1.12	1630502	1299724.5	-21.58	0.80	1280357	-21.47
ibm15	1994685.8	1.38	1957427	1714002.6	-14.07	0.62	1697900	-13.26
ibm16	2222138	1.05	2190510	1820492.6	-18.07	0.80	1795440	-18.04
ibm17	2745042.7	1.16	2680986	2159900.5	-21.32	1.02	2119537	-20.94
ibm18	2639356.6	4.29	2504083	1876039.6	-28.92	0.87	1845377	-26.31
Average		1.83			-22.20	0.83		-21.27

Table A.14: Detailed wire-length comparison of 1000-iteration 3-D Gravity vs. hMetis for the ISPD98 benchmark suite, 10 runs.

Circuit	hMetis			2000 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
19ks	14493.3	1.61	14123	11232	-22.50	1.30	10965	-22.36
avq.large	104104.1	1.22	101693	77368.4	-25.68	0.73	76497	-24.78
avq.small	94688	0.88	93823	71065.2	-24.95	1.21	69772	-25.63
baluP	3263.5	1.77	3164	2727.9	-16.41	0.57	2707	-14.44
biomedP	25239.2	0.80	24839	21199	-16.01	0.83	20852	-16.05
golem3	687104.9	0.68	679554	557303.9	-18.89	0.55	551397	-18.86
industry2	78997.7	0.92	78052	74138.1	-6.15	0.82	73360	-6.01
industry3	152962.3	1.05	149592	123573.2	-19.21	0.69	121455	-18.81
p1	4156.1	2.17	4071	3357.5	-19.22	0.95	3309	-18.72
p2	18562.5	1.64	18044	15650.7	-15.69	1.06	15349	-14.94
s13207P	26501.3	2.15	25617	21943.4	-17.20	0.84	21677	-15.38
s15850P	30950.7	0.65	30729	26948.1	-12.93	1.14	26527	-13.67
s35932	57926.1	1.74	56120	48848.3	-15.67	0.75	48417	-13.73
s38417	73282.6	1.21	72355	70769.2	-3.43	1.27	69071	-4.54
s38584	72643.9	1.10	71560	68822.4	-5.26	1.37	67084	-6.25
s9234P	17670.1	0.89	17463	16046.1	-9.19	0.88	15886	-9.03
structP	7064.1	1.41	6879	5899.7	-16.48	0.85	5792	-15.80
t2	8501.9	1.44	8337	6709.5	-21.08	1.41	6580	-21.07
t3	7828.2	1.42	7658	6669.5	-14.80	1.61	6526	-14.78
t4	7375.9	1.63	7242	6381.2	-13.49	1.23	6256	-13.62
t5	12568.2	0.57	12481	11211.6	-10.79	1.15	10977	-12.05
t6	7968.1	1.51	7785	6811.5	-14.52	1.10	6658	-14.48
Average		1.29			-15.43	1.02		-15.23

Table A.15: Detailed wire-length comparison of 2000-iteration 3-D Gravity vs. hMetis for the ACM/SIGDA benchmark suite, 10 runs.

Circuit	hMetis			2000 iterations				
	length	% standard deviation	minimum	length	% change	% standard deviation	minimum	% change
ibm01	92601.5	0.91	90591	76677.5	-17.20	0.89	75549	-16.60
ibm02	202121.7	0.65	199264	168871.8	-16.45	0.69	167382	-16.00
ibm03	234600.9	0.94	231824	190093.1	-18.97	0.82	186704	-19.46
ibm04	301324.1	1.07	297370	227659.4	-24.45	0.94	222997	-25.01
ibm05	367004.5	1.27	360659	271572.3	-26.00	0.59	267895	-25.72
ibm06	333985	3.21	323919	252607.6	-24.37	1.00	248437	-23.30
ibm07	473844.3	1.82	458047	364976.6	-22.98	1.00	358872	-21.65
ibm08	531860.7	1.38	521203	400305.7	-24.73	0.70	395468	-24.12
ibm09	617201.7	2.56	587107	461238.2	-25.27	0.49	457856	-22.01
ibm10	832125.1	3.79	796763	614281.7	-26.18	0.72	606702	-23.85
ibm11	872118.2	2.72	842491	611976.8	-29.83	0.88	605889	-28.08
ibm12	991783.9	1.63	959567	772627.1	-22.10	0.97	762112	-20.58
ibm13	1000941.8	2.05	971330	778011.1	-22.27	0.70	771750	-20.55
ibm14	1657408.1	1.12	1630502	1294570.1	-21.89	0.85	1271832	-22.00
ibm15	1994685.8	1.38	1957427	1710050.3	-14.27	0.50	1700391	-13.13
ibm16	2222138	1.05	2190510	1807631	-18.65	0.72	1785101	-18.51
ibm17	2745042.7	1.16	2680986	2148845.2	-21.72	0.67	2132214	-20.47
ibm18	2639356.6	4.29	2504083	1857957	-29.61	0.53	1839756	-26.53
Average		1.83			-22.61	0.76		-21.53

Table A.16: Detailed wire-length comparison of 2000-iteration 3-D Gravity vs. hMetis for the ISPD98 benchmark suite, 10 runs.

APPENDIX A. DETAILED GRAVITY RESULTS

Glossary

$\{\ldots\}$ $\{\ldots\}$ denotes a set whose elements are listed between the braces.

{element : condition}

{element : condition} is the set of all elements that satisfy the given condition.

$\subseteq$ $A \subseteq B$, set A is a subset of B.

$\forall$ $\forall e$: For all elements e...

$\in$ $e \in A$ means element e is a member of set A.

$\exists$ $\exists e$: There exists and element e...

$\cup$ $A \cup B$ is the union of sets A and B.

$\bigcup$ $\bigcup_{e \in E} e$ is the union of all elements in E.

$\cap$ $A \cap B$ is the intersection of sets A and B.

$\sum$ $\sum_{e \in E}$ is the sum over all elements e in set E.

$\omega()$ Asymptotic strict lower bound of the argument [CLR90]:

$$\omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } \forall n \geq n_0\ 0 \leq cg(n) < f(n)\}.$$

$\Omega()$ Asymptotic lower bound of the argument [CLR90]:

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } \forall n \geq n_0\ 0 \leq cg(n) \leq f(n)\}.$$

$\Theta()$ Asymptotically tight bound of the argument [CLR90]:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } \forall n \geq n_0\, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$$

bendless embedding

See *straight-line placement.*

bisection cut width

C_{ij} is the bisection cut width of an embedding. It is the number of times that routed edges cross a cut line between coordinates points i and $i+1$ in dimension j.

C_{ij} See *bisection cut width.*

cache Cache memory stores a small copy of part of the main memory. Data in the cache can be accessed much faster than main memory. A *cache hit* means that a required piece of data was found in the cache, and no slow main memory access was required. The *cache hit ratio* is the ratio of cache hits over total memory accesses. Modern CPUs have several levels of caching, with level-one cache being the fastest.

cardinality

The cardinality of an edge (net) is the number of nodes (cell, pins) connected by the edge.

cell A cell is a node in a circuit. Cell and node are used interchangeably in this dissertation.

channel

In a grid with k parallel channels, parallel grid edges are labelled from 1 to k. A grid edge labelled i belongs to channel i. All grid edges labelled i together form channel i.

clique A clique is a graph or subgraph in which every node is connected to every other node.

cluster

Cf. paragraph "Clusters" in section 5.3.3.1 on page 120.

coarse-grain placement

A *coarse-grain placement* determines node positions in a placement without regard to final placement constraints such as grid coordinates or standard cell layout. A *fine-grain placement* needs to be performed to achieve a good final placement.

congestion

The *congestion* of a grid edge is the number of hypergraph edges that get routed through thie grid edge.

connected

In a graph $G(V, E)$, two nodes u, v are *connected* if either $\{u, v\} \in E$, or a node connected to u is connected to v.

CPU Central Processing Unit.

depth first search

A depth first search of a graph visits every node of a graph by visiting every unvisited node along one path continuing along the next possible path.

die A small, usually rectangular area of silicon, onto which an integrated circuit is deposited.

dilation

The **dilation** of an embedding is the maximum path length in the host graph between neighbours in the guest graph.

diameter

The *diameter*(V, E) of a graph $G(V, E)$ is the maximum distance between any two nodes.

distance

The *distance* between two nodes u and v in a set of edges E is the cardinality of the smallest subset of E in which u and v are connected. The distance may be thought of as the number of hops over edges separating u and v.

dynamic range

A *dynamic range*, also called a log range, covers several orders of magnitude. Entities distributed evenly over a dynamic range have an equal logarithm spacing, e.g. 10, 100, 1000, etc.

edge An *edge* interconnects nodes in a hypergraph. An edge is formally a set of nodes.

EPROM

Electrically Programmable Read-Only Memory

expansion

We call the ratio of the number of host graph nodes over the number of guest graph nodes the **expansion** of the embedding.

$f()$ $f(S)$:If S is a set and f is a function, then $f(S)$ is the set of values mapped by applying f to the elements of S.

$f()_i$ $f(u)_i$ indicates the dimension-i grid coordinate of $f(u)$.

fine-grain placement

A *fine grain placement* take the output of a *coarse-grain placement* and generates a placement that conforms to placement constraints such as standard cell layouts.

FPGA

Field-Programmable Gate Array

genetic algorithms

Genetic algorithms find a solution to an optimization problem by randomly combining two sample solutions and generating new solutions.

grid See definition 3 on page 8.

grid line

A *grid line* is the subgraph of a grid induced by all grid nodes that differ only in one coordinate. Thus, in a d-dimensional grid, a grid line is identified by the $d-1$ coordinates that the grid-line nodes have in common.

group Cf. paragraph "Groups" in section 5.3.3.1 on page 121.

interconnection

See *edge*.

leg The *legs* of a routed edge are the either horizontal or vertical sequences of grid edges that host the routed edge.

load The maximum number of guest graph nodes that are embedded into the same host graph node is called the **load** of an embedding.

multiplicity

The *multiplicity* of an edge is the number of parallel edges connecting the same nodes. Formally, the multiplicity of an edge is simply a positive integer associated with this edge.

net An interconnection, such as a wire, that connects several elements of an electronic circuit is often called a *net.* The terms *edge* and *net* are often used interchangeably.

$O()$ Asymptotic upper bound of the argument [CLR90]:

$$O(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } \forall n \geq n_0\, 0 \leq f(n) \leq cg(n)\}.$$

partitioning

Partitioning divides a graph or hypergraph into two or more subsets. Normally attempts are made to reduce the number of edges crossing the partition.

pin A pin is a connection of a circuit cell (node) to a net (edge).

segment cut width

$C_{\{x,y\}}$ is the segment cut width of an embedding. It is the number of edges that are routed through grid segment $\{x, y\}$.

simulated annealing

Simulated annealing algorithms find a solution to an optimization problem by mimicking the freezing of a liquid.

SRAM

Static Read-Only Memory

straight-line placement

A *straight-line placement,* or *bendless embedding,* of a hypergraph places all nodes onto a grid such that all hypergraph neighbours share a grid line.

VLSI Very Large Scale Integration

wafer A thin slice from a purified silicon cylinder which can be cut into many silicon dies.

x_i x_i is the i-th coordinate of x.

$\mathbb{Z}$ $\mathbb{Z}$ is the set of all integers. $\mathbb{Z}^n$ is the n-dimensional integer space.

Bibliography

[ACH+97] C. J. Alpert, T. Chan, D. J.-H. Huang, I. Markov, and K. Yan. Quadratic placement revisited. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 752–757, 1997.

[AHK87] Sheldon B. Akers, Dov Harel, and Balakrishnan Krishnamurthy. The star graph: An attractive alternative to the n-cube. In *Proceedings of the International Conference on Parallel Processing*, pages 393–400, 1987.

[AJK82] Kurt J. Antreich, Frank M. Johannes, and Fritz H. Kirsch. A new approach for solving the placement problem using force models. In *1982 IEEE International Symposium on Circuits and Systems*, pages 481–486, 1982.

[AK93] C. J. Alpert and A. B. Kahng. Geometric embeddings for faster and better multi-way netlist partitioning. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 743–748, 1993.

[AK94] C. J. Alpert and A. B. Kahng. A general framework for vertex orderings, with applications o netlist clustering. In *IEEE International Conference on Computer-Aided Design*, pages 63–67, 1994.

[Akl97] Selim G. Akl. *Parallel computation: models and methods.* Prentice Hall, Upper Saddle River, New Jersey, 1997.

[Alp98] Charles J. Alpert. The ISPD98 circuit benchmark suite. In *Proceedings of the International Symposium on Physical Design*, pages 85–90, 1998.

[AY95] Charles J. Alpert and So-Zen Yao. Spectral partitioning: The more eigenvectors, the better. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 195–200, 1995.

[Ban94] Prithviraj Banerjee. *Parallel Algorithms For VLSI Computer-Aided Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[BBG+97] Bruno Beauquier, Jean-Claude Bermond, Luisa Gargano, Pavol Hell, Stéphane Perennes, and Ugo Vaccaro. Graph problems arising from wavelength-routing in all-optical networks. In *Proceedings of the 2nd Workshop on Optics and Computer Science (WOCS '97)*, April 1997.

[Ber85] Claude Berge. *Graphs*, volume 6.1 of *North-Holland Mathematical Library.* Elsevier Science Publishing Company, New York, 1985.

[Ber89] Claude Berge. *Hypergraphs: Combinatorics of Finite Sets*, volume 45 of *North-Holland Mathematical Library.* Elsevier Science Publishing Company, New York, 1989.

[BFRV92] Steven D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays*, volume SECS 180 of *Kluwer international series in engineering and computer science.* Kluwer Academic Publishers, Boston, 1992.

[Bre77] Melvin A. Breuer. A class of min-cut placement algorithms. In *Proceedings of the 14th ACM/IEEE Design Automation Conference*, pages 284–290, 1977.

[Brg93] Franc Brglez. ACM/SIGDA design automation benchmarks: Catalyst or anathema? *IEEE Design & Test of Computers*, 10(3):87–91, September 1993.

[BS87] Jayaram Bhasker and Sartaj Sahni. Optimal linear arrangement of circuit components. *Journal of VLSI and Computer Systems*, 2:87–109, 1987.

[BS96] A. Barak and E. Schenfeld. Embedding classical communication topologies in the scalable OPAM architecture. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):962–978, September 1996.

[CH93] Ray-I Chang and Pei-Yung Hsiao. Force directed self-organizing map and its application to VLSI cell placement. In *1993 IEEE International Conference on Neural Networks*, pages 103–109, 1993.

[CL92] Maw-Haw Chen and Sing-Ling Lee. Linear time algorithms for k-cutwidth problem. In *Proceedings of the Third International Symposium on Algorithms and Computation ISAAC '92*, pages 21–30, 1992.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1990.

[CLS96] Jason Cong, Wilburt Juan Labio, and Narayanan Shivakumar. Multiway VLSI circuit partitioning based on dual net representation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(4):396–409, April 1996.

[CS87] James P. Cohoon and Sartaj Sahni. Heuristics for backplane ordering. *Journal of VLSI and Computer Systems*, 2(1-2):37–60, 1987.

[CS92] Liang-Fang Chao and Edwin Hsing-Mean Sha. Algorithms for min-cut linear arrangements of outerplanar graphs. In *1992 IEEE International Symposium on Circuits and Systems*, volume 4, pages 1851–1854, 1992.

[CSZ94] Pak K. Chan, Martine D. F. Schlag, and Jason Y. Zien. Spectral k-way ratio-cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):1088–1096, September 1994.

[CV98] Silviu M. S. A. Chiricescu and M. Michael Vai. A three-dimensional FPGA with an integrated memory for in-application reconfiguration data. In *1998 IEEE International Symposium on Circuits and Systems*, volume 2, pages 232–235, 1998.

[CW91] Chung-Kuan Cheng and Yen-Chuen A. Wei. An improved two-way partitioning algorithm with stable performance. *IEEE Transactions on Computer-Aided Design*, 10(12):1502–1511, December 1991.

[DBETT99] Guiseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: algorithms for the visualization of graphs.* Progress in Computer Science. Prentice Hall, Upper Saddle River, New Jersey, 1999.

[DD96] Shantanu Dutt and Wenyong Deng. A probability-based approach to VLSI circuit partitioning. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, pages 100–105, 1996.

[DJA94] Konrad Doll, Frank M. Johannes, and Kurt J. Antreich. Iterative placement improvement by network flow methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(10):1189–1200, October 1994.

[DNVM+94] J. Depreitere, H. Neefs, H. Van Marck, J. Van Campenhout, R. Baets, B. Dhoedt, H. Thienpont, and I. Veretennicoff. An optoelectronic 3-d field programmable gate array. In *Field-programmable logic : architectures, synthesis, and applications : 4th International Workshop on Field-Programmable Logic and Applications*, volume 849 of *Lecture notes in computer science*, pages 352–360, 1994.

[Don79] Wilm E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Transactions on Circuits and Systems*, CAS-26(4):272–277, April 1979.

[Dow92] Patrick W. Dowd. Wavelength division multiple access channel hypercube processor interconnection. *IEEE Transactions on Computers*, 41(10):1223–1241, October 1992.

[DT94] Khaled Day and Anand Tripathi. A comparative study of topological properties of hypercubes and star graphs. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):31–38, January 1994.

[EJ98] Hans Eisenmann and Frank M. Johannes. Generic global placement and floorplanning. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 269–274, 1998.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A liner-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[Gav77] Fănică Gavril. Some NP-complete problems on graphs. In *Proceedings of the 11th Conference on Information Sciences and Systems*, pages 91–95, Baltimore, Maryland, 1977. Johns Hopkins University.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[GJMP80] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, 1980.

[GLC93] T. Gao, C. L. Liu, and K. C. Chen. A performance driven hierachical partitioning placement algorithm. In *European Design Automation Conference*, pages 33–38, 1993.

[GMH+91] Zicheng Guo, Rami G. Melhem, Richard W. Hall, Donald M. Chiarulli, and Steven P. Levitan. Pipelined communications in optically interconnected arrays. *Journal of Parallel and Distributed Computing*, 12:269–282, 1991.

[Got81] Satoshi Goto. An efficient algorithm for the two-dimensional placement problem in electrical circuit layout. *IEEE Transactions on Circuits and Systems*, CAS-28(1):12–18, January 1981.

[Han66] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14(2):255–265, March 1966.

[Har86] Mark R. Hartoog. Analysis of placement procedures for VLSI standard cell layout. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 314–319, 1986.

[HB94] David A. Hoelzeman and Saïd Bettayeb. On the genus of star graphs. *IEEE Transactions on Computers*, 43(6):755–759, June 1994.

[HK97] Lars W. Hagen and Andrew B. Kahng. Combining problem reduction and adaptive multistart: A new technique for superior iterative partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(7):709–717, July 1997.

[HS71] Akihiro Hashimoto and James Stevens. Wire routing by optimizing channel assignment within large aperture. In *Proceedings of the SHARE-ACM-IEEE Design Automation Workshop*, pages 155–169, 1971.

[Int97] Intel Corporation. *Pentium II Processor Developer's Manual*, October 1997.

[KAKS97] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 526–529, 1997.

[Kar99] George Karypis, May 1999. private communication.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[KK89] Yong-Seok Kim and Myunghwan Kim. A generalization of the debruijn graph for dense symmetric interconnection networks in multicomputer systems. *The Transactions of the IEICE*, E 72(6):691–694, June 1989.

[KK98a] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme of irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[KK98b] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49:291–307, February 1970.

[Kof98] James S. Koford. Method and system for improving a placement of cells using energetic placement units alternating contraction and expansion operations. *United States Patent*, (5754444), 1998.

[KP87] Fadi J. Kurdahi and Alice C. Parker. REAL: A program for register allocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 210–214, 1987.

[KPST94] Philip Klein, Serge Plotkin, Clifford Stein, and Éva Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, June 1994.

[Kri84] Balakrishnan Krishnamurty. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, May 1984.

[KSJA91] Jürgen M. Kleinhans, Georg Sigl, Frank M. Johannes, and Kurt J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, 10(3):356–365, March 1991.

[KSS98] Karl Kurbel, Bernd Schneider, and Kirti Singh. Solving optimization problems by parallel recombinative simulated annealing on a parallel computer - an application to standard cell placement in VLSI design. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 28(3):454–461, June 1998.

[LB94] Sharam Latifi and Nader Bagherzadeh. Incomplete star: An incrementally scalable network based on the star graph. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):97–102, January 1994.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architechtures: Arrays, Trees, Hypercubes.* Morgan Kaufman Publishers, Inc., San Mateo, California, 1992.

[LMV+98] Miriam Leeser, Waleed M. Meleis, Manukuan M. Vai, Silviu Chiricescu, Weidong Xu, and Paul M. Zavracky. Rothko: A three-dimensional FPGA. *IEEE Designs and Test of Computers*, 15(1):16–23, January 1998.

[LR71] Bernard S. Landman and Roy L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transaction on Computers*, C-20(12):1469–1479, December 1971.

[LR86] Frank Thomson Leighton and Arnold L. Rosenberg. Three-dimensional circuit layouts. *SIAM Journal on Computing*, 15(3):793–813, August 1986.

[MD97] Burkhard Monien and Ralf Diekmann. A local graph partitioning heuristic meeting bisection bounds. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[MM98] Aleksandar Milenkovic and Veljko Milutinovic. A quantitative analysis of wiring lengths in 2D and 3D VLSI. *Microelectronics Journal*, 29(6):313–321, June 1998.

[MR99] Pinaki Mazumder and Elizabeth M. Rudnick. *Genetic Algorithms for VLSI Design, Layout & Test Automation.* Prentice Hall, Upper Saddle River, 1999.

[MRSV85] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. In *Proceedings of the 24th IEEE Conference on Decision and Control*, pages 761–767, 1985.

[Obe95] Stefan Thomas Henning Obenaus. Topology of a high speed free-space photonic network. Master's thesis, McGill University, 1995.

[Ohm98] Michiroh Ohmura. An initial placement algorithm for 3-d VLSI. In *1998 IEEE International Symposium on Circuits and Systems*, volume 6, pages 195–198, 1998.

[OS97] Stefan Thomas Obenaus and Ted H. Szymanski. Embeddings of star graphs into optical meshes without bends. *Journal of Parallel and Distributed Computing*, 44(2):97–106, August 1997.

[OS99] Stefan Thomas Obenaus and Ted H. Szymanski. Placement benchmarks for 3-d VLSI. In Luis Miguel Silveira, Srinivas Devadas, and Ricardo Reis, editors, *VLSI: Systems on a chip*, pages 447–455. Kluwer Academic Publishing, December 1999.

[PA96] Sandy Pavel and Selim G. Akl. Area-time trade-offs in arrays with optical pipelined buses. *Applied Optics*, 35(11):1827–1835, April 1996.

[PBS98] Phiroze N. Parakh, Richard B. Brown, and Karem A. Sakallah. Congestion driven quadratic placement. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 275–278, 1998.

[Pul82] Norman J. Pullman. Clique coverings of graphs - a survey. In *Combinatorial Mathematics X - Proceedings of the Conference*, pages 72–85, 1982.

[QAM94] Ke Qiu, Selim G. Akl, and Henk Meijer. On some properties and algorithms for the star and pancake interconnection networks. *Journal of Parallel and Distributed Computing*, 22:16–25, 1994.

[RT86] M. Reber and R. Tielert. Benefits of vertically stacked integrated circuits for sequential logic. In *1996 IEEE International Symposium on Circuits and Systems*, pages 121–124, 1986.

[RWY93] Sanjay Ranka, Jhy-Chun Wang, and Nangkang Yeh. Embedding meshes on the star graph. *Journal of Parallel and Distributed Computing*, 19(2):131–135, October 1993.

[Saa95] Youssef G. Saab. A fast and robust network bisection algorithm. *IEEE Transactions on Computers*, 44(7):903–913, July 1995.

[San89] Laura A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, January 1989.

[SH98] Ted H. Szymanski and H. Scott Hinton. Architecture of a terabit free-space intelligent optical backplane. *Journal of Parallel and Distributed Computing*, 55(1):1–31, 1998.

[SK88] Peter R. Suaris and Gershon Kedem. An algorithm for quadrisection and its application to standard cell placement. *IEEE Transactions on Circuits and Systems*, 35(3):294–303, March 1988.

[SK96] Kwang-Su Seong and Chong-Min Kyung. A clustering based linear ordering algorithm for netlist partitioning. *IEICE Transactions on Fundamentals*, E79-A(12):2185–2191, December 1996.

[SKK+98] Ranko Scepanovic, James S. Koford, Valeriy B. Kudryavtsev, Alexander E. Andreev, Stanislav V. Aleshin, Alexander S. Podkolzin, and Douglas B. Boyle. Computer implemented method for leveling interconnect wiring density in a cell placement for an integrated circuit chip. *United States Patent*, (5835378), 1998.

[SM91] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, June 1991.

[Smi97] Micheal John Sebastian Smith. *Application-Specific Integrated Circuits*. VLSI Systems. Addison-Wesley, 1997.

[SO98] Ted H. Szymanski and Stefan Thomas Obenaus. Embedding properties of reconfigurable partitionable optical networks. In Pascal Berthome and Afonso Ferreira, editors, *Optical Interconnections and Parallel*

Processing: Trends at The Interface, pages 235–257. Kluwer Academic Publishing, January 1998. Invited chapter.

[SS93] Wern-Jieh Sun and Carl Sechen. Efficient and effective placement for very large circuits. In *IEEE International Conference on Computer-Aided Design*, pages 170–177, 1993.

[SS95] Wern-Jieh Sun and Carl Sechen. Efficient and effective placement for very large circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(3):349–359, March 1995.

[SS97] Wern-Jieh Sun and Carl M. Sechen. A parallel standard cell placement algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(11):1342–1357, November 1997.

[SS98] Boonchuay Supmonchai and Ted Szymanski. High speed VLSI concentrators for terabit intelligent optical backplanes. In *Proceedings of SPIE - the International Society for Optical Engineering*, volume 3490, pages 306–310, 1998.

[SSB98] Horst D. Simon, Andrew Sohn, and Rupak Biswas. HARP: A dynamic spectral partitioner. *Journal of Parallel and Distributed Computing*, 50:83–103, 1998.

[SvC97] Dirk Stroobandt and Jan van Campenhout. Estimating interconnection lengths in three-dimensional computer systems. *IEICE Transactions on Information and Systems*, E80-D(10):1024–1031, October 1997.

[SY95] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation.* IEEE Press, Piscataway, New Jersey, 1995.

[Szy95] Ted H. Szymanski. "Hypermeshes": Optical interconnection networks for parallel computing. *Journal of Parallel and Distributed Computing*, 26(1):1–23, April 1995.

[Tan96] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 3 edition, 1996.

[TK91] Ren-Song Tsay and Ernest Kuh. A unified approach to partitioning and placement. *IEEE Transactions on Circuits and Systems*, 38(5):521–533, May 1991.

[TKH88] Ren-Song Tsay, Ernest S. Kuh, and Chi-Ping Hsu. PROUD: A fast sea-of-gates placement algorithm. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 318–323, 1988.

[TL93] Too-Seng Tia and C.L. Liu. A new performance driven macro-cell placement algorithm. In *European Design Automation Conference*, pages 66–71, 1993.

[Tuc75] Alan Tucker. Coloring a family of circular arcs. *SIAM Journal on Applied Mathematics*, 29(3):493–502, November 1975.

[Tut60] W. T. Tutte. Convex representations of graphs. *Proceedings of the London Mathematical Society*, 10(3):304–320, 1960.

[Tut63] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13(3):743–767, 1963.

[TW95] Chao Chi Tong and Chuan-lin Wu. Routing in a three-dimensional chip. *IEEE Transactions on Computers*, 44(1):106–117, January 1995.

[VMVC97] H. Van Marck and J. Van Campenhout. Three-dimensional optoelectronic architechtures for massively parallel processing systems. In *Proceedings of the Fourth International Conference on Massively Parallel Processing Using Optical Interconnections*, pages 178–182, 1997.

[Vyg97] Jens Vygen. Algorithms for large-scale flat placement. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 746–751, 1997.

[Wol94] Wayne Wolf. *Modern VLSI Design - A Systems Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[WWM82] G. J. Wipfler, M. Wiesel, and D. A. Mlynski. A combined force and cut algorithm for hierachical VLSI layout. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 671–677, 1982.

[YCL94] Ching-Wei Yeh, Chung-Kuan Cheng, and Ting-Ting Y. Lin. A general purpose, multiple-way partitioning algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(12):1480–1488, December 1994.

[YW96] Chingwei Yeh and Chi-Shong Wang. On the integration of partitioning and global routing for rectilinear placement problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(1):83–91, January 1996.

Index