SEQUENTIAL/PARALLEL HEURISTIC ALGORITHMS FOR VLSI STANDARD CELL PLACEMENT

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

GUANGFA LU

In partial fulfilment of requirements

for the degree of

Master of Science

August, 2004

ABSTRACT

SEQUENTIAL/PARALLEL HEURISTIC ALGORITHMS FOR VLSI STANDARD CELL PLACEMENT

Guangfa Lu
University of Guelph, 2004

Advisor:
Dr. Shawki Areibi

With advanced sub-micron technologies, the exponentially increasing number of transistors on a VLSI chip causes placement within physical design automation to become more and more important and consequently extremely complicated and time consuming.

This thesis addresses the placement for VLSI standard cell designs. A number of heuristic optimization techniques for placement are studied and implemented, in particular, local search, Tabu Search, Simulated Annealing and Genetic Algorithm. The Tabu Search reduces wire length on average by 52.4% while Simulated Annealing yields a 61% improvement on average. Furthermore, two parallel island-based GA models are implemented on a loosely-coupled parallel computing architecture to pursue better performance. With the synchronous model, an average speedup of 6.2 was achieved by seven processors. On the other hand, the asynchronous model achieved a 7.6 speedup. The former obtained the speedups while maintaining equal or better quality of solutions than a serial GA. In addition to the above developed algorithms, preprocessing and postprocessing procedures were analyzed and developed to further enhance solution quality.

Acknowledgements

I would like to take this opportunity to express my sincere appreciation and thanks to my advisor Dr. Shawki Areibi for his strong support and great guidance throughout this research. Without his help, this work would never have been possible.

I also want to thank SHARCNET for their financial and technical support to the parallel implementations in this research.

To

my parents

whose love and encouragement helped accomplish this thesis;

And to

my wife Yuanshi Lin and my son Joesphan Lu

who have supported me vigorously as I chase my dream.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Integrated Circuit technology has significantly evolved from Small Scale Integration (SSI) in the 1960s, which contained only tens or hundreds of transistors on a chip, to Very Large Scale Integration (VLSI) today, which integrates tens of millions of modules. It is predicted that designers will be able to build VLSI chips with the integration of billions of transistors running at tens of GHz by the next decade [Rese02]. This rapid development of circuit technologies will not be realized without the wide usage of computer aided design (CAD) tools that assist designers in this process. The automation of various steps in the design as well as fabrication of VLSI circuits has made this rapid growth in integration technology possible. However, the exponential increase of circuit complexity also brings a tremendous challenge to all phases in the VLSI design cycle, and forces designers to seek better approaches for solving the problems efficiently.

A typical VLSI design cycle involves architecture design, functional design, logic design, circuit design, physical design, fabrication and packaging. Physical design is accomplished in several stages such as partitioning, placement, routing and compaction. Our emphasis is on physical design automation and more specifically the placement stage.

1.1 Motivations

The technologies of VLSI have evolved rapidly for the last four decades. In 1965, only 30 transistors could be realized on a single chip to implement very simple logic functions. By 1975, the device count on a chip increased up to 65,000. Now with the introduction of deep sub-micron semiconductor technologies, the number of transistors on a chip has grown dramatically to tens of millions or even more [Rese02]. For example, an Intel Pentium 4 processor fabricated with 0.13 micron technology in 2002 packs 55 million transistors onto a fingernail size silicon.

In the semiconductor industry, the common belief is “smaller is better”. Smaller feature size allows more transistors to be integrated into a single chip, therefore more logic functions can be implemented within the chip. In addition, small size often leads to two other advantages: lower power consumption and higher performance. In fact, silicon’s power dissipation has improved as much as has the integration of transistors over recent years. In addition, speed has increased even more rapidly than the number of transistors. For example, the Intel Pentium i486 in 1989 ran at 25 MHz, whereas the Pentium 4 processor in 2004 runs at 3.4 GHz!

However, sub-micron technologies also bring many challenges to the VLSI design and fabrication process. Firstly, when the feature size is scaled down, the gate delay usually decreases and leads to a faster circuit speed, but interconnect capacitance does not scale down as much as the gate delay [Raba03], as shown in Figure 1.1. Thus, the effect of interconnect delay dominates the circuit delay in deep sub-

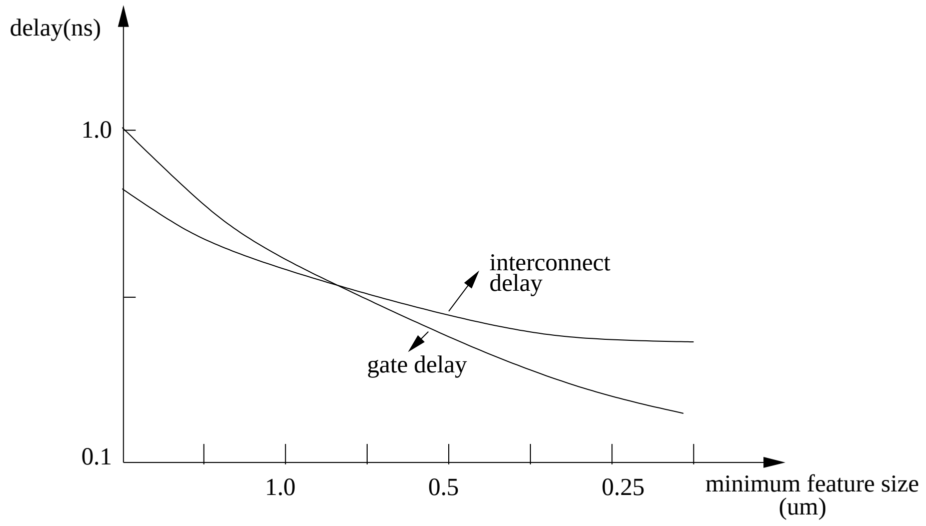


Figure 1.1: Delays *vs.* Feature Size [Arei04]

micron VLSI circuits. Secondly, the nearly exponentially increasing number of transistors on a chip makes layout of transistors more complicated, and it may take a much longer time to obtain good results. Therefore, effective placement techniques need to be developed to meet the fast-paced nature of VLSI CAD design. In this research, we study and implement a series of heuristic algorithms, such as Local Search algorithms (LS), Tabu Search (TS), Genetic Algorithms (GA), and Simulated Annealing (SA), to iteratively optimize layout solutions for VLSI standard cell placement. Some preprocessing and postprocessing techniques are also implemented to further improve solution qualities.

Almost all the subtasks in VLSI physical design automation are NP-hard[1] prob-

[1]NP-hard problems are the complexity class of decision problems that are intrinsically harder

lems [Blan85] and therefore require large run-times on existing sequential computers. Consequently, parallel processing for VLSI CAD applications can be used to speed up these processes and help deal with complex designs. In this research, we investigate synchronous and asynchronous models of Island-based Genetic Algorithms [Lu04], and implement these techniques on a loosely-coupled parallel architecture to produce layout solutions efficiently.

1.2 Overall Methodology

The creation of a standard cell circuit involves many activities, including logic design, circuit design, physical design, simulation, testing, and fabrication. Each of these activities can be further divided into a number of tasks due to its high complexity. In this work, we focus on the task of placement in the physical design process for standard cell circuits, specifically heuristic optimization techniques that solve the underlying subtasks.

VLSI circuit placement is a complex problem. Although there can be many objectives in standard cell placement, we have restricted our research to focus on the main objective, which is minimizing the total wire length. Other objectives such as routability, low power and performance can also be applied based on this main objective.

The approaches used in this thesis are illustrated in Figure 1.2. The first stage in the research is to develop a set of algorithms to preprocess an initial constructed solution. The second stage attempts to develop iterative improvement algorithms

---

than those that can be solved by a nondeterministic Turing machine in polynomial time. [Atal99]

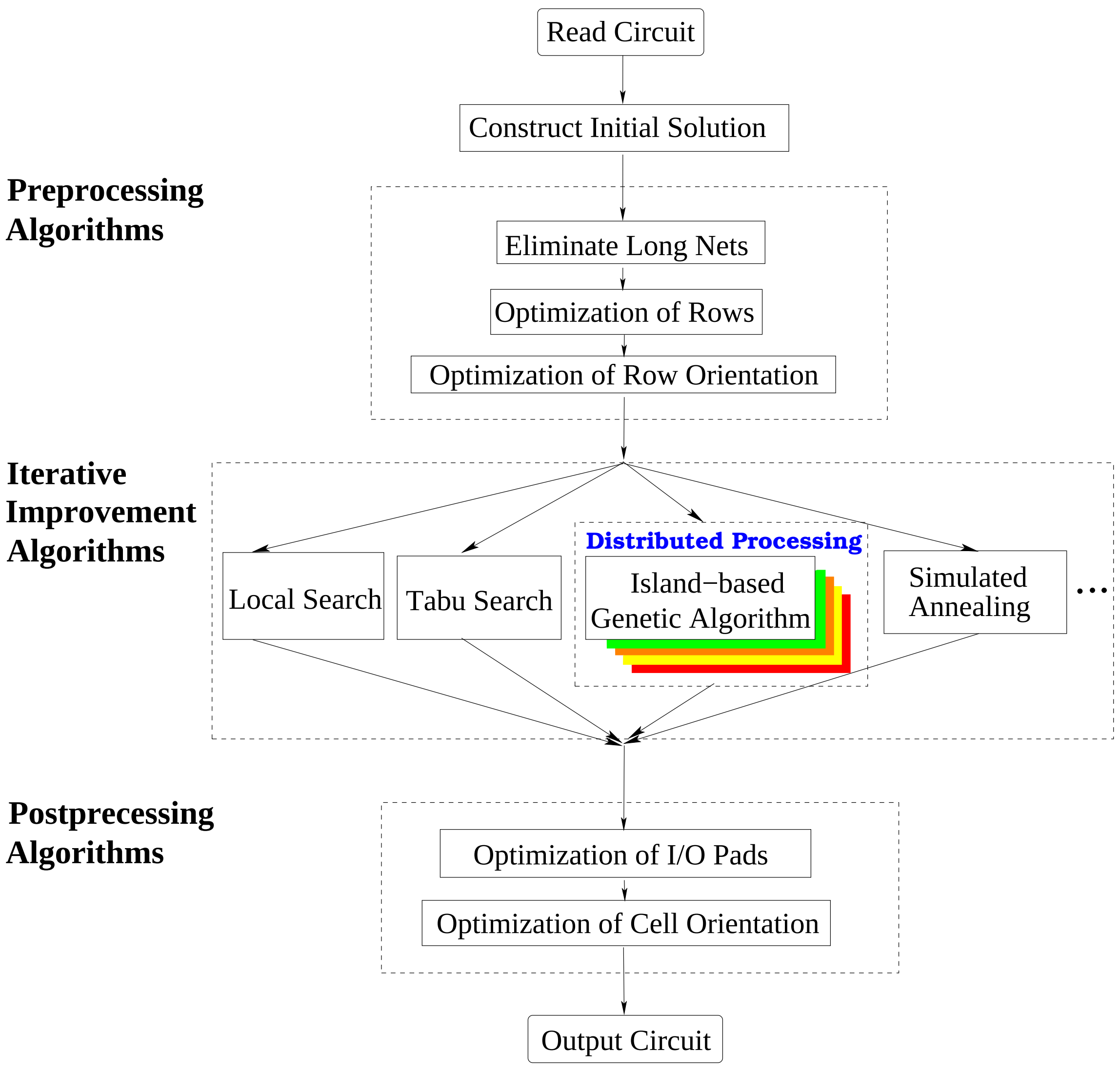


Figure 1.2: Our Overall Methodology for the Placement Problem

to effectively solve the problem in reasonable time. Two placement strategies are studied and implemented as iterative heuristic search techniques: (a) Search only within the valid solution space; (b) Both valid and invalid moves are accepted but a penalty is imposed on invalid moves (a legalization procedure is required at the end of the search). Based on the second strategy, we developed several meta-heuristic algorithms for placement: Tabu Search, Genetic Algorithms, and Simulated Annealing. Two Island-based Genetic Algorithms were successfully implemented on a cluster of networked Sun SunBlade 2000 workstations in parallel to speed up the sequential Genetic Algorithm implementation. Finally, some postprocessing techniques are employed to further enhance solution quality.

1.3 Contributions

The main contributions of this thesis can be summarized as follows:

- Extensively evaluated two placement strategies, and improved the cost function for the second strategy.
- Implemented several heuristic algorithms to effectively reduce wire length, and compared their performance; Proposed an approach to dynamically change weights in Simulated Annealing.
- Implemented an Island-based Genetic Algorithm on s distributed processing system to speed up the sequential GA process, and proposed an efficient migration scheme for the model.
- Developed several preprocessing and postprocessing techniques to further improve solution quality.
- A conference publication has resulted from this thesis, which was presented in the Genetic and Evolutionary Computation Conference (GECCO) 2004 in Seattle, USA [Lu04].

1.4 Thesis Organization

The chapters in this thesis are organized as follows. In Chapter 2, a background of the placement problem is provided, and previous approaches for standard cell placement are introduced, as well as some of their parallel implementations. Chapter 3 describes heuristic and meta-heuristic techniques for standard cell placement. Following this, parallel implementations of synchronous and asynchronous Island-based Genetic Algorithms are presented in Chapter 4. Chapter 5 describes some preprocessing and postprocessing techniques for placement. Finally, Chapter 6 provides conclusions and a summary of future work.

Chapter 2

Background

A typical VLSI design cycle is illustrated in Figure 2.1. The design flow starts with the specification of a circuit, followed by functional design, logic design, circuit design, physical design, and fabrication/packaging/testing [Sher98].

System specification involves the design goals specified by the customer (e.g., performance, functionality, and size), fabrication constraints, design techniques, and market requirement. Functional design is also called behavioral design. The system is divided into a number of functional units and the behaviour/functions of each is identified, as well as the interconnect requirements between the units. In the logic design stage, Register Transfer Level (RTL) descriptions are derived and tested, which consists of Boolean expression and timing information. The correctness of logic functions of the system is simulated and verified in this step. In the circuit design phase, Boolean expressions are converted into circuit representations such as detailed circuit diagrams or netlists. Layout of the circuit is created in the physical design stage according to the circuit representation obtained from previ-

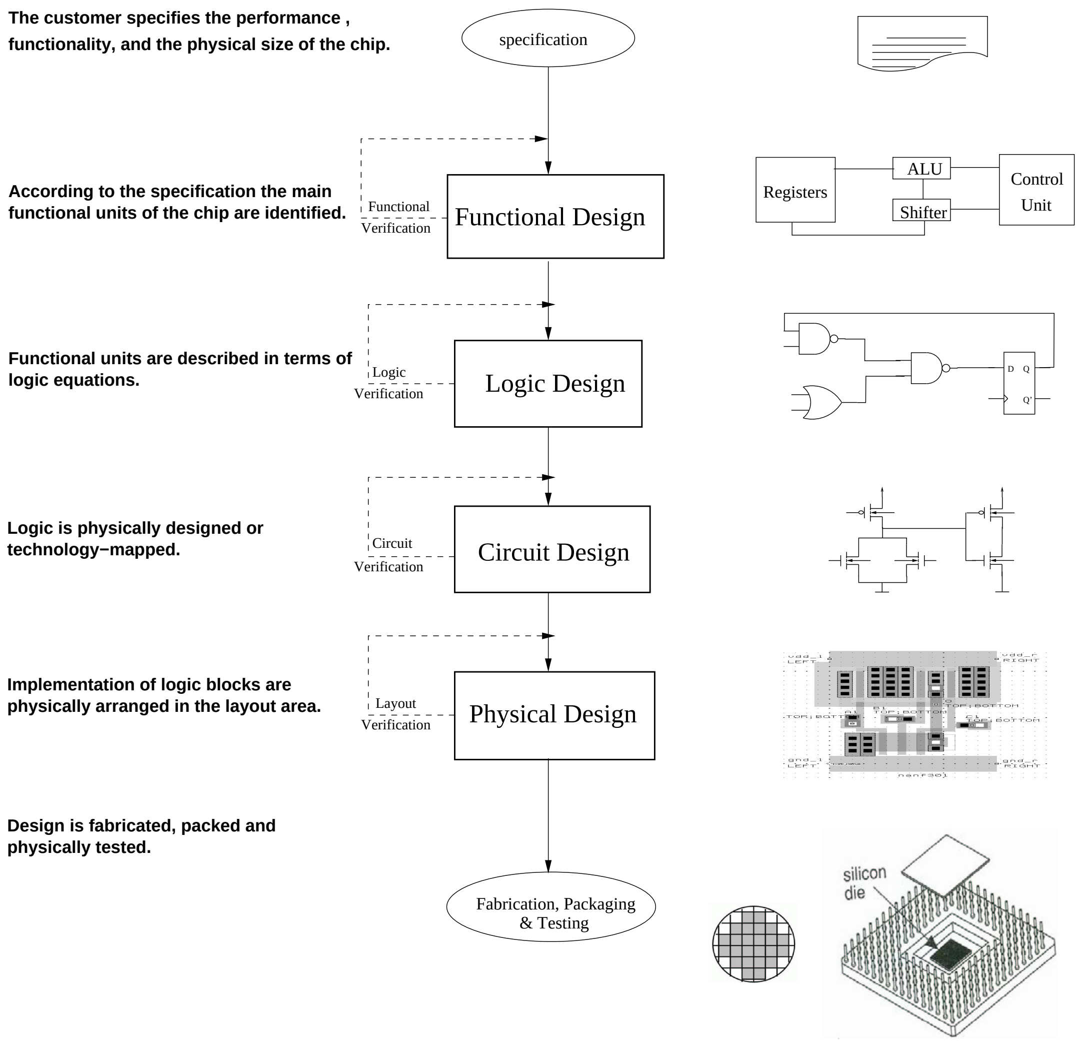

The customer specifies the performance , functionality, and the physical size of the chip.
specification
According to the specification the main functional units of the chip are identified.
Functional Verification
Functional Design
Registers
ALU
Shifter
Control Unit
Functional units are described in terms of logic equations.
Logic Verification
Logic Design
D Q
Q'
Logic is physically designed or technology−mapped.
Circuit Verification
Circuit Design
Implementation of logic blocks are physically arranged in the layout area.
Layout Verification
Physical Design
Design is fabricated, packed and physically tested.
Fabrication, Packaging & Testing
silicon die

Figure 2.1: VLSI Design Cycle

ous stages. A layout is the geometric description of a circuit, specifying a set of planar geometric shapes in multiple layers. Our work mainly focuses on developing effective heuristic techniques to solve the VLSI circuit placement efficiently. The last step is fabrication, packaging, and testing. The output of the layout phase is fractured into photo-lithographic masks on the wafer during the fabrication process.

This process is complicated, involving deposition and diffusion of various materials on the wafer. Finally, individual VLSI chips are diced from the fabricated wafer, and each chip is then packaged and tested.

To design large, complex VLSI chips quickly to meet the market requirement, efficient powerful automated design tools are essential.

2.1 Physical Design Cycle

Physical design is the process of converting the specification of a circuit into the geometric description of a layout [Sher98]. Due to the large number of components (transistors), physical design is an extremely complex process [Gare79]. It is usually broken down into several stages, such as partitioning, placement, and routing, as shown in Figure 2.2. In general, partitioning divides a large system into a set of subsystems. Placement is performed to place the modules on the chip by satisfying certain constraints, and finally routing determines how the wires will connect the modules.

Partitioning is the task of decomposing a circuit into smaller parts. A VLSI chip may contains millions or even tens of millions of transistors. Dealing with the entire circuit at a time could be difficult and inefficient. The goal of partitioning is to transform a large problem into a set of smaller sub-problems by dividing a circuit into several sub circuits of manageable size. Good partitioning tools can efficiently reduce layout costs. To make it easier for independent design of these subproblems, the interconnections (cuts) between partitions should be minimized. Minimization of the cuts between partitions is the most important objective for VLSI circuit

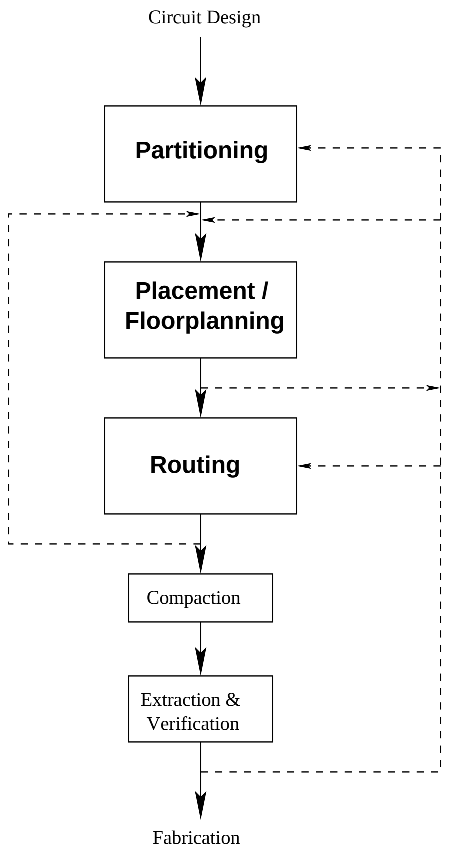


Figure 2.2: Physical Design Process

partitioning. Usually the delay between partitions is much greater than the delay within a partition. Therefore, a circuit must be carefully partitioned, since this process may degrade the performance of the final design if one or more critical paths are cut. Minimizing the cut critical path is part of the objective function for high performance circuits. Other factors that constrain circuit partitioning include the number and area of partitions.

Placement is the problem of placing modules on a chip while satisfying certain constraints: (a) to minimize the area of the VLSI chip, (b) to meet the timing specifications, (c) solutions produced should be routable. Placement objectives can vary, depending on the layout style of the chip. In some situations this problem can be treated as a floor planning problem. Although for simple circuits, global optimal solutions can be obtained automatically by a computer. However, most circuits in practice have many modules and constraints and therefore it is virtually impossible to solve them by exact methods. Heuristic approaches are widely used to find suboptimal placements solutions quickly.

Routing attempts to determine how the wires connect the modules. Two steps are involved in routing: global routing and detailed routing. Global routing attempts to partition the routing region into a set of sub-regions to simplify the routing problem. Besides minimizing the total wire length, congestion is evenly distributed over the routing area. Detailed routing effectively realizes interconnections among modules in the chip. The two-layer Manhattan model is widely used for detailed routing, where horizontal wires are routed in one layer (metal), and vertical wires are routed in the other layer (polysilicon).

2.2 Layout Styles

Layout styles are broadly classified as either full custom style or semi-custom style. The **full custom** layout style allows the designer to fully control the circuit layout. Usually in full custom designs, a circuit is hierarchically partitioned into a collection of sub-circuits of any size that can be placed at any location on the silicon wafer of a chip. This layout style can produce very compact and high performance circuits because designers can justify a highly optimized layout without most constraints existing in other design styles. However, the full custom design process has the highest complexity among other design styles. With millions of transistors involved, it is extremely difficult to manually lay out the entire chip. Accordingly, it is suitable only for large production of small chips with high performance requirement.

On the other hand, semi-custom layout styles are simpler, where circuits are predesigned and specifically placed on certain locations on a chip [Wolf02]. As Figure 2.3 illustrates, cell-based semi-custom styles include standard cells and macro cells,

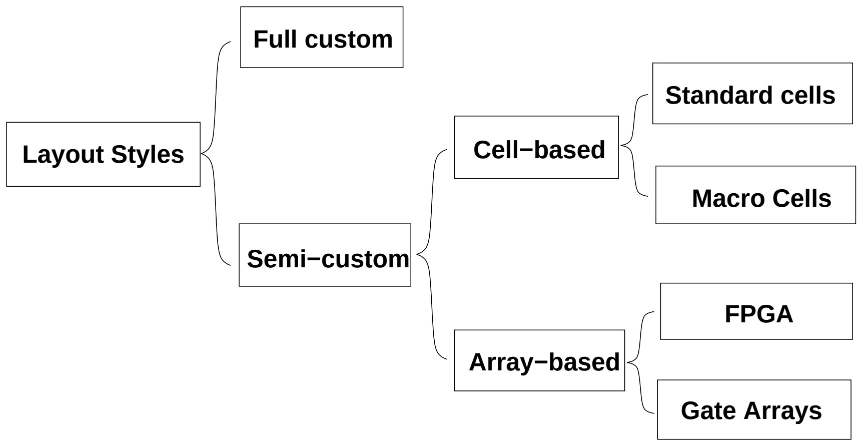
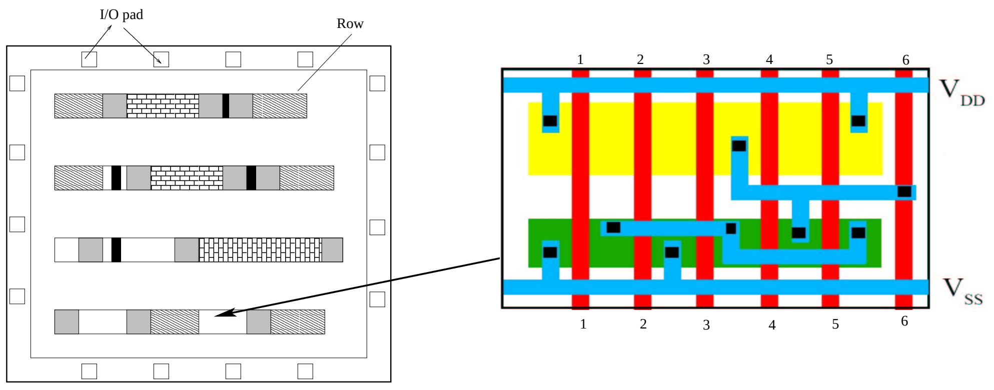


Figure 2.3: Layout Styles

while array-based semi-custom styles include Gate Arrays and Field Programmable Gate Arrays [Sher98].

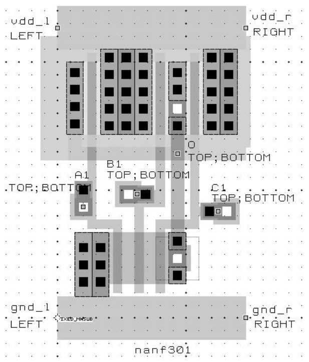
In order to simplify and automate the physical design process, cell-based approaches have been in use for many years, and have become extremely popular for VLSI circuit design. The basis of the cell-based approach is to implement any logic function by reusing a limited number of predesigned library cells called **standard-cells** or macro cells. Figure 2.4 demonstrates that a circuit based on standard cell design style consists of several modules each representing a standard cell. Prede-

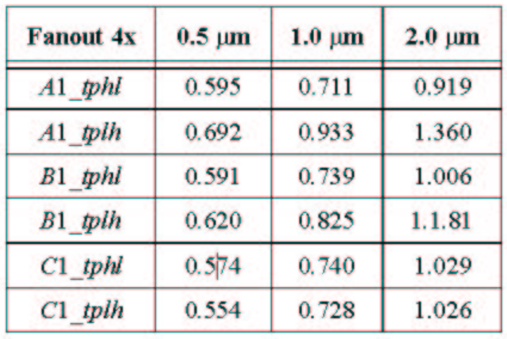


(a) A Circuit (b) A Standard Cell

Figure 2.4: Standard Cell Design Style

fined cells in a cell library have the same height but varying width. They are tested, analyzed, and specified by the vendor. A cell library can contain a set of basic logic functions over varied fan-in/out counts and capabilities, so that virtually any logic function can be implemented. These basic logic functions include AND, OR, NOT, NAND, NOR, EXOR, NXOR, flip-flops, e.t.c. Usually for each function, there are several versions of the same cells with different characteristics (area, speed, and power) in the library. For example, Figure 2.5 illustrates a four fan-out 3-input NAND standard cell as taken from the Mississippi State Library [Raba03], where the figure in the left is the cell layout, and the table on the right is the cell characterization for 0.5 um, 1.0 um, and 2.0 um technologies. Besides containing general purpose basic logic, standard cell libraries may also contain some specifically optimized complex circuits to achieve special features (faster speed, smaller area, or low power). Modules on a standard cell circuit are organized in rows separated by empty spaces called routing channels. Figure 2.6 shows an example of a standard cell circuit. Since the cells are designed once and are reusable in standard cell





Fanout 4x	0.5 μm	1.0 μm	2.0 μm
A1_tphl	0.595	0.711	0.919
A1_tplh	0.692	0.933	1.360
B1_tphl	0.591	0.739	1.006
B1_tplh	0.620	0.825	1.1.81
C1_tphl	0.574	0.740	1.029
C1_tplh	0.554	0.728	1.026

Figure 2.5: A Four Fan-out 3-input NAND Standard Cell

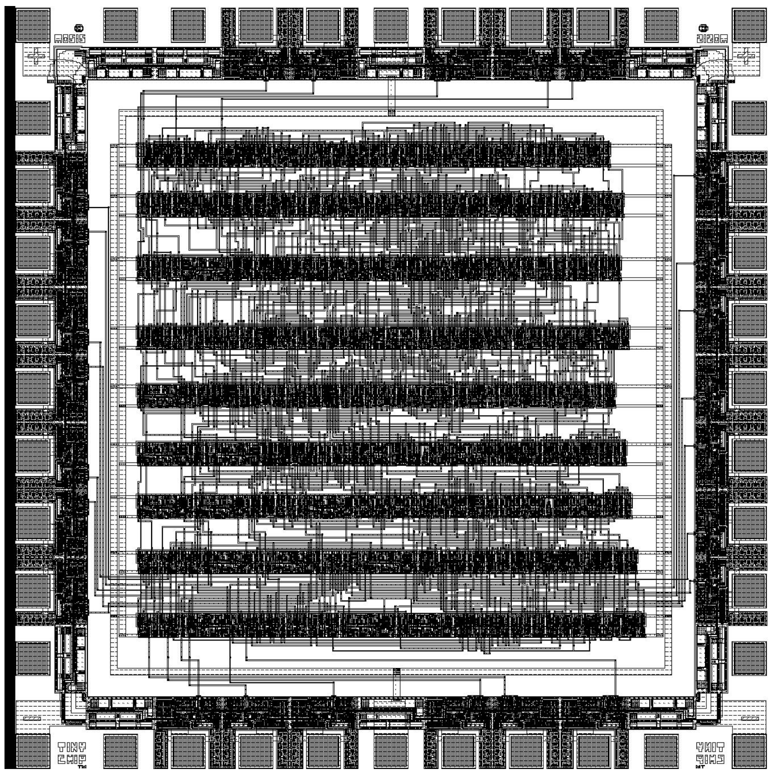


Figure 2.6: A Standard Cell Circuit [Raba03]

design style, the physical design automation step involved is significantly simpler and cheaper than that of the full custom design approach. The design time is also dramatically reduced. In the real world, short design time and low design cost are two critical factors that may contribute to the success of a circuit in the market. However, since standard cells are predefined by vendors, there is no way to finely tune the cells during the development of circuits. In addition, a standard cell design usually occupies more area than a hand-crafted full custom design.

Macro cell design style is similar to the standard cell style except that it uses predefined macro cells (processors, RAM, etc.) instead of standard cells. Macro cells are circuit blocks made as full custom by the vendor, and they have an arbitrary shape and size. Due to the irregular shape and size of the blocks, the automation of macro cell design is more difficult than that of standard cell design [Cai94]. High density, high performance, and short design time are the advantages of macro cell design style.

Gate Array design style is even simpler than standard cell style because all the cells in gate array are identical, such as a three-input NAND gate. Predefined gates are separated by both vertical and horizontal channels. Cells are interconnected by using these channels. Due to the simplicity of identical cells, gate arrays are cheaper and easier to produce than the previous cell-based styles and full custom style, but with inferior circuit density and performance.

The **Field Programmable Gate Array** (FPGA) is a powerful approach that can significantly reduce manufacturing turn-around time to ASIC design, as well as the cost for low-volume manufacturing [ea89]. FPGAs consist of programmable logic blocks and programmable interconnections. A logic block can be viewed as a memory block and programmed with a “look-up table”. The user simply programs the interconnect between logic blocks. Technologies of FPGAs include

SRAM [Xili04], EEROM [Alte04], Anti_fuse [Acte04], etc. Since the customization of a FPGA is rather simple, development cost is low and design time is fast.

Table 2.1 summarizes the differences between full custom, standard cell, macro cell, gate array and FPGA layout styles in area, performance, and design cost [Sher98]. Note that a mixture of different layout styles is also common.

	Layout Styles				
	Full Custom	Standard Cell	Macro Cell	Gate Array	FPGA
Area	compact	compact to moderate	compact	moderate	large
Performance	high	high to moderate	high	moderate	low
Design Cost	high	medium	high	medium	low

Table 2.1: Comparison of Different Design Styles

2.3 The Standard Cell Placement Problem

The placement problem is a very important step in the physical design cycle. Inferior placements can have drastic effect on chip area, power and performance [Sher98].

Standard cells in a library have fixed shapes and terminal locations. They are modules with identical height but varied widths as illustrated in Figure 2.7. In general, given a cell library and a netlist providing the interconnections between these modules, the standard cell placement problem involves properly placing the cells in rows on a VLSI circuit chip. For the cells in the same row or in adjacent rows, their interconnection wires lie in the routing channels. For the cells in non-

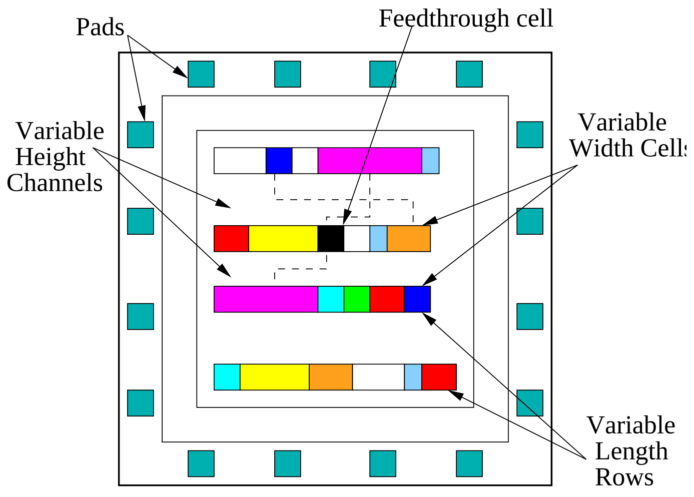


Figure 2.7: A Standard Cell Circuit

adjacent row, wires pass through either (1) the special empty space between cells, which is call feedthrough; or (2) the over-the-cell regions if the chip is fabricated in a two or more metal layers process. The total chip area consists of two parts: the area required for the cell rows (including feedthroughs) and the area required for routing of wires (channel area). Figure 2.8 emphasizes the routing channels of a standard cell circuit. Minimizing the width of the longest row and minimizing the width of channel heights leads to a minimum layout area [Shah91]. The standard cell placement problem can be stated as follows: Given a circuit consisting of fixed standard cells, and a netlist interconnecting the terminals of these cells, produce a layout indicating the positions of each cells in rows such that all the nets can be routed and the total layout area is minimized.

Graph theory plays a crucial role in modelling many VLSI physical design problems including standard cell placement [Sher98]. A circuit can be represented by a hypergraph $G(V, E)$, where $V = \{v_1, v_2, \cdots, v_n\}$ is the vertex set of the hypergraph,

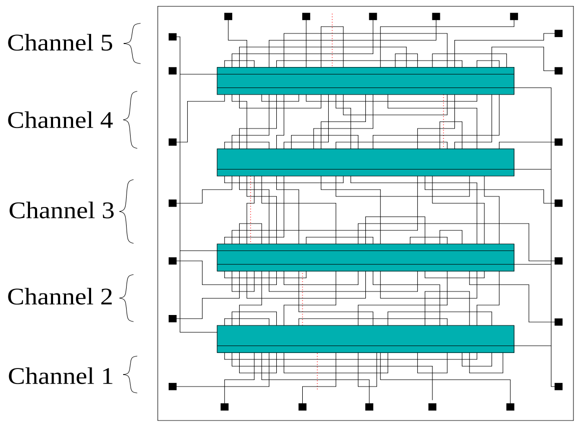


Figure 2.8: The Routing Channels

and $E = \{e_1, e_2, \cdots, e_m\}$ is the set of edges of the hypergraph. V represents the set of cells to be placed while E represents the set of nets connecting the cells. Each edge e_j is an ordered pair of vertices with a non-negative weight w_j assigned to it. The hypergraph is then transformed into a graph (a hypergraph with all hyperedge sizes equal to 2). In other words, each vertex in the graph corresponds to a cell on the chip and an edge is defined between two nodes in the graph if and only if the two cells are connected by a net. The placement task seeks to assign all cells of the circuit to legal locations such that cells do not overlap. Each cell i is assigned to a location (x_i, y_i) on the XY-plane. The cost of an edge connecting two cells i and j with locations (x_i, y_i) and (x_j, y_j) is computed as the product of the squared l_2 norm of the difference vector $(x_i - x_j, y_i - y_j)$ and the weight of the connecting edge w_{ij}. The total cost of a placement layout, denoted $\phi(x, y)$, can be estimated by the sum of wire length over all nets [Yang03]:

$$\phi(x,y) = \sum_{1 \le i < j \le N} w_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2] \tag{2.1}$$

Formulation (2.1) can be rewritten in matrix form as:

$$\phi(x,y) = \frac{1}{2}\mathbf{x}^T\mathbf{C}\mathbf{x} + \mathbf{d}_x^T\mathbf{x} + \frac{1}{2}\mathbf{y}^T\mathbf{C}\mathbf{y} + \mathbf{d}_y^T\mathbf{y} + \mathbf{t} \tag{2.2}$$

Vectors $\mathbf{x}$ and $\mathbf{y}$ denote the coordinates of the N movable cells; matrix $\mathbf{C}$ is the Hessian matrix; vectors $\mathbf{d}_x^T$ and $\mathbf{d}_y^T$ and the constant term $\mathbf{t}$ result from the contributions of the fixed cells. Normally the first moment constraints are added to force the distribution of the cells to be uniform around the center of the placement area. It follows that the quadratic placement model is given by:

$$\begin{aligned} &\text{Min } \phi(x,y) \\ &\text{s.t. } A_x x = b_x \\ &\quad A_y y = b_y \\ &\quad l_x \le x_i \le u_x \\ &\quad l_y \le y_i \le u_y \end{aligned}$$

where A_x and A_y are $q \times n$ matrices; q is the number of regions into which the placement area has been partitioned. The $q \times 1$ vectors b_x and b_y represent the centres of the q regions. The parameters l_x, u_x, l_y and u_y are lower and upper bounds on the x and y coordinates of the cells.

The quality of a placement can be specified mainly by the result of chip area, as well as other evaluation metric employed such as the routability of nets and circuit performance. These three primary objectives are frequently used to evaluate solutions obtained by the placement tool. However, sometimes these objectives may contradict with each other. For instance, minimizing layout area may increase the longest path (decrease circuit performance). As mentioned above, for standard cell layout style, the total chip area is approximately equal to the area of the cells plus the area occupied by the interconnecting wires, minimizing the wire-length is approximately equivalent to minimizing the chip area. In this research, we focus on the main objective, which is minimizing the chip area, or minimizing the total wire length. Other objectives such as routability and performance can also be applied based on this objective.

Several approaches have been proposed to evaluate the wire length of circuits. To name a few, half-perimeter method, complete graph method, minimum chain method, source-to-sink connection method, Steiner-tree approximation approach and minimum spanning tree method [Sech86, Shah91].

At the placement stage, detailed routing is not available yet, and the exact total wire length of the layout cannot be obtained. Therefore, the Half-perimeter method is commonly used to approximately estimate the total wire-length, namely half perimeter wire length (HPWL). To evaluate the cost of a given net that is connecting several modules, the half perimeter distance of the bounding box of this net is often used to measure the estimated wire length of the net. Figure 2.9 shows how a bounding box approximately measures a net that is connecting four cells.

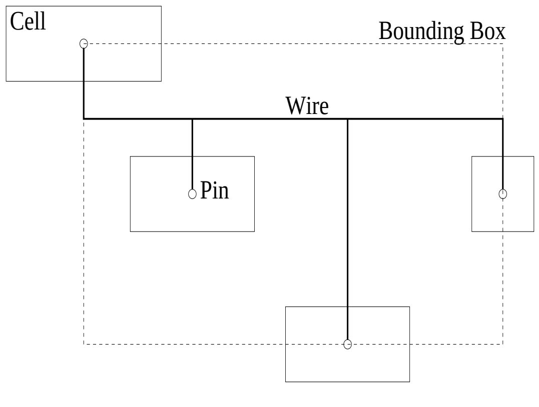


Figure 2.9: Wire Length Estimated by a Bounding box

For a given circuit, the sum of half perimeters of all nets is the estimated total wire length of the layout. That is,

$$\sum_{i \in nets} HPWL_i = \sum_{i \in nets} \frac{p_i}{2} = \sum_{i \in nets} [span\_x(i) + span\_y(i)] \tag{2.3}$$

where p_i denotes the perimeter of net i; $span\_x(i)$ and $span\_y(i)$ denotes the vertical and horizontal spans of net i's bounding box respectively. Different non-negative weights can be assigned to the nets, such as critical nets.

2.4 Heuristic Optimization Techniques

In the last two decades, interest in heuristic[1] search methods for NP-hard combinatorial optimization problems has continuously grown and many significant studies have been carried out. A number of heuristics techniques are now popular and widely used in various fields. VLSI physical design automation is a rich area where heuristic optimization approaches can be applied effectively.

2.4.1 Classification of Placement Algorithms

The standard cell placement problem involves placing a set of cells on a chip layout, given a netlist that provides the connectivity information between cells and a library that contains cell layout information. Module placement is known as an NP-hard problem and cannot be solved in polynomial time [Blan85]. Therefore, a number

[1]A heuristic is a technique that reduces or limits the search for solutions in domains that are difficult and poorly understood. Heuristics do not guarantee optimal, or even feasible, solutions and are often used with no theoretical guarantee [Stev00].

of heuristic approaches were developed to solve this problem, which vary in their efficiency, robustness and layout quality.

Placement algorithms can be classified into two major categories [Sher98]: *constructive based placement* and *iterative improvement* as illustrated in Figure 2.10. Constructive placement algorithms usually build up a placement layout from scratch.

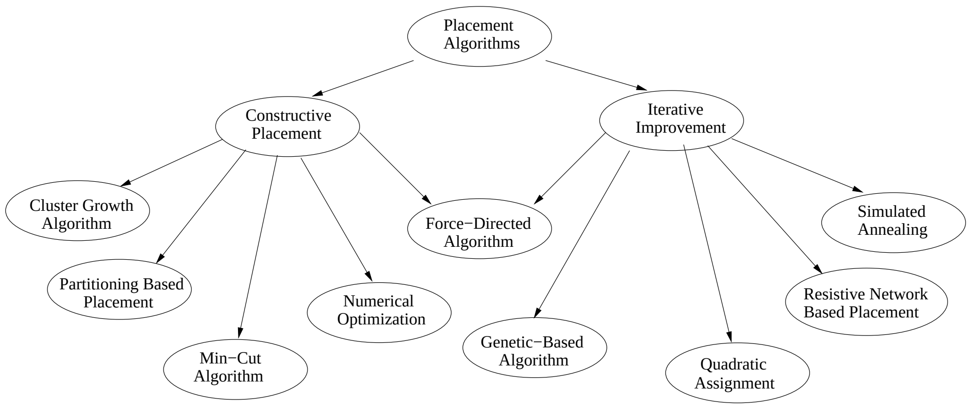


Figure 2.10: Classification of Placement Techniques

Good initial placements are usually crucial for several iterative improvement techniques to succeed in obtaining high quality solution. Once an initial placement is obtained, it is necessary to improve the solution by an iterative improvement technique. Iterative improvement techniques attempt to repeatedly modify initial solutions in search of a better solution until no further improvement can be found. In this section, several methods utilized to iteratively improve placement solutions are reviewed. The following schema describes a typical placement procedure:

Step 1: Use at least one constructive algorithm to obtain an initial placement.

If multiple initial solutions are used, the best solution is selected.

Step 2: (a) n:=1;

(b) employ the improvement algorithm;

(c) if successful then goto (b), else goto (d);

(d) if stopping criteria is NOT met, then n:=n+1 and goto (b).

2.4.2 Module Interchange Based Methods

Iteratively searching for better solutions from the neighbourhood in an attempt to ultimately reach the optimum is the basis of many iterative improvement algorithms. A neighbourhood move is usually produced by perturbing one of two modules of a current solution. Some module interchange methods are reviewed in the following context.

In [Chyu83], the adjacent pairwise exchange method uses a simple placement model that simplifies all modules of a layout by a connection matrix, and iteratively exchanges only the adjacent cells (see Figure 2.11(a)). If the total wire length associated with the pair of cells can be decreased by the exchange, the move is accepted; otherwise it is rejected.

Although any two cells have the chance to be swapped in the approach above, if they are far away from each other, their interchange may require many iterations. Therefore in some algorithms in the literature, two swapped cells are not necessarily adjacent, as Figure 2.11(b) illustrates.

In [Swar95], two kinds of interchanges are used for creating a new move: cell displacement and cell exchange. As Figure 2.11(c) displays, cell displacement selects a random cell and moves it to a random location; Cell exchange on the other hand selects two random cells and exchanges their locations. The algorithm chooses a source cell and a destination first; if the destination is vacant, a cell displacement is performed, otherwise a cell exchange is performed. This method has proven to be effective in searching for better placement solutions from the neighbourhood.

When discussing iterative improvement algorithms for placement, two different strategies are discussed in terms of the feasibility of the solution space of the algorithms. The first strategy was proposed by Grover [Grov86]. This approach searches in a solution space containing only valid placement configurations. The second strategy was used by the successful placement and routing package TimberWolf [Sech85]. It allows module overlaps in its intermediate solutions, that is, invalid placements are contained in the solution space. The author claims that the cost function can be updated efficiently after each move.

2.4.3 Tabu Search Algorithm

The Tabu Search meta-heuristic algorithm is a simple yet effective combinatorial optimization strategy. Tabu Search is based on a hill-climbing search method. However, by introducing a list of forbidden moves recorded from the recent search trajectory, Tabu Search guides the search to escape local minima and at the same time avoids revisiting previous solutions [Glov90]. The origin of Tabu Search goes back to the 1970s and its current form was first proposed by Fred Glover in 1986. A general Tabu Search algorithm is shown in Figure 2.12, where s is the current solution, s' is the best available candidate in the neighbourhood while $s*$ is the best solution so far. $N(s)$ denotes the set of neighbour solutions, T is tabu memory, and $V*$ is a set of candidates from tabu-based neighbourhood.

The goal of Tabu Search is to make systematic use of tabu memory to guide a hill-climbing neighbourhood search that accepts improving moves as well as deteriorating moves. Tabu memory T involves historical information about a finite-sized list of past moves. With such a short term memory and strategy, Tabu Search can escape from local minima and avoid cycling to previously visited solutions [Glov90]. To increase the flexibility of the Tabu Search while maintaining the above basic features, aspiration rules can be embedded in the algorithm. A commonly used aspiration rule is based on the following: if the cost associated with a Tabu move is less than the aspiration value associated with the cost of the current solution, then the move's Tabu status is temporarily ignored. The appropriate use of aspiration sometimes is crucial to achieve good solution quality.

Tabu Search has been successfully applied to a wide range of applications in VLSI CAD design. Song *et al* proposed their iterative improvement algorithms for placement using a Tabu Search technique, which attempts to overcome the limitation of a local optimal point and obtain better placement quality [Song92]. In this algorithm, the neighbourhood $N(s)$ of a solution s is defined as the set of all solutions s' that can be obtained from s by exchanging any two connected modules. Moves are created by the pairwise exchange method by first selecting a random module m_j and then finding a target module m_j that is connected to m_i and exchanging their locations. Thus the candidate set at each move is $V^* \subseteq N(s) - T$. Their algorithm yielded solutions that are 6-14% better than the well-known TimberWolfSC 5.4 based on Simulated Annealing algorithms in a fraction of the time.

Areibi *et al* [Arei93] employed Tabu Search techniques to the circuit partitioning problem for VLSI design. Results of netlist partitions with 10% fewer cut nets than those of Simulated Annealing were reported, with a 3$\times$ speedup.

In [Lim91] and [Lim96], Tabu Search based algorithms are also applied to placement and global routing for standard cell circuits. They used a divide and conquer strategy based on successive partitioning approach.

Sait *et al* [Sait01] implemented a TS algorithm for standard cell placement. In each iteration, neighbourhood solutions are generated by the exchanges of the locations of any two randomly selected cells. The neighbourhood size is determined by the circuit size (e.g., 24 for a circuit with 2081 cells, and 70 for the 3540-cell circuit). The tabu memory used involves the indices information of the interchanges.

Emmert *et al* [Emme99, Emme03] developed a method for FPGA floor planning, which combines a Tabu Search based approach and clustering techniques. The authors claim that their approach achieves a large speedup in execution time without deterioration in solution quality, compared with commercially available CAD tools.

The size of tabu memory is difficult to determine prior to execution, but the proper choice of the size is critical to most problems. Therefore, advanced schemes for Tabu Search methods have been studied by a number of researchers. Dammeyer *et al* [Damm93] presented an approach to dynamically manage the size of the tabu list using the reverse elimination method. Battiti *et al* [Batt94] proposed a complicated tabu memory management scheme called *Reactive Tabu Search*. Based on the basic principle of the generic Tabu Search, Battiti's approach employed a fully automatic reaction and escape mechanism to maintain the escape diversification and the exploitation of fast memory structures. Both Dammeyer and Battiti reported better results than the original Tabu Search algorithm; however, more computation time was involved.

2.4.4 Genetic Algorithms

John Holland [Holl75] first used the term Genetic Algorithm (GA) in 1975. He described how an evolutionary alike process can be applied to solving mathematical problems and engineering optimization problems. His approach was based on Charles Darwin's theories of evolution and natural selection. Darwin observed that, in each generation, as variations are introduced into the population, the less fit individuals have a higher chance of dying off than the more fit ones. GAs attempt to use a similar concept of reproduction and survival of the fittest to solve various optimization problems, based on naturally occurring genetic operators.

GAs work on a population of individuals called a chromosomes that represent a potential solution to a given problem. The size of a population determines the amount of information stored by the GA. A fitness function is used to evaluate each individual's fitness value to measure the goodness of a solution to the problem. In each iteration, the algorithm breeds a population of individuals by applying genetic operators such as selection, crossover, inversion, and mutation. A new generation then evolves from the existing population hopefully with better solutions. Due to the stochastic selection process, good schemata are more likely to be inherited by the individuals of new generations. The fitness of the entire population is therefore improved over a number of generations [Gold89].

The selection operator chooses members from the current generation for reproduction. The crossover operator on the other hand combines portions from the two parents to create two new offsprings. The mutation operator makes an incremental change to an individual to form a slightly new chromosome. Finally, the replacement operator decides which offsprings are going to replace which members of the current generation to create a new generation of individuals. Generational GAs are commonly used, where the new generation replaces the old one in each iteration. When a GA has overlapping generations (only a fraction of the individuals are replaced in each generation), it is called a steady-state GA [Mazu99b].

Figure 2.14 illustrates two different types of Genetic Algorithms: (a) a generational GA, and (b) a steady-state GA. The main difference is in the replacement strategy explained earlier. The steady-state GA has a higher growth pressure on the promising individuals than that of a generational GA. However, it is susceptible to stagnation [Mazu99a].

One advantage of GAs is their exploration capabilities. GAs are multi-point heuristics and are less likely to get trapped in the local optima than most of other optimization techniques [Reev03]. In other words, although GAs are not guaranteed to find the global optimal solutions for a problem, they generally can provide fairly good solutions. Moreover, they are domain-independent stochastic search techniques [Koza98]. The power of GAs comes from the fact that the technique is robust, and can deal successfully with a wide range of problem areas.

GINIE is a VLSI placement technique based on the Genetic Algorithm approach [Leon88]. Each layout configuration is represented by a string where the *i-th* allele contains the cell in the *i-th* position of the chip. Starting from randomly constructed solutions, GINIE applies crossover, mutation and inversion to a population of individuals over a number of generations. The stopping criteria is met when there is no further improvement for 10,000 generations. The population size was set to 50, and 12 offsprings are created for each generation. Comparable quality of placement results to those obtained by Simulated Annealing algorithms were reported [Leon88].

Attempting to eliminate the complicated interactions between placement and routing stages, Krashinsky combined the objectives of the two stages and used a Genetic Algorithm to simultaneously optimize the placement and routing of a circuit[Kras00]. Each individual encodes the placement and routing information of a layout. A genome consists of a number of genes (blocks of cells), each of which contains a set of routing numbers: x and y placement coordinates, a routing algorithm specifier, and a routing ordering number. There are two parts in the scoring function: the most important one is the number of connections that was successfully routed, and the second is the wire length. The author claims that this method can produce layouts in which modules are more fully connected and therefore the chips have a smaller wire length and use a smaller area.

However, one disadvantage of GAs is that they are usually very time consuming, compared with other techniques [Reev03]. To speed up the GA process, researchers have studied parallel computing techniques to apply to Genetic Algorithms.

2.4.5 Simulated Annealing Algorithms

In the early 1980's, Kirkpatrick *et al* introduced the concepts of annealing for optimization [Kirk83]. The Simulated Annealing (SA) algorithm is based on a strong analogy between the physical annealing process of solids and solving combinatorial optimization problems [Aart90]. It starts with an initial solution, and continuously attempts to transform the current configuration into one of its neighbours. Mathematically, this can be described with a **Markov chain**: a sequence of trials, in which the probability of the outcome of a given trial depends only on the outcome of the previous trial. In Simulated Annealing algorithms, the *acceptance probability* of a move is determined by:

$$\wp_c(s') = \begin{cases} 1, & \text{if } C(s') \leq C(s); \\ \exp(\frac{C(s)-C(s')}{T}), & \text{if } C(s') \geq C(s). \end{cases} \tag{2.4}$$

where T is the current temperature, $C(s)$ and $C(s')$ denote the costs of the current solution and a neighbourhood solution, respectively.

The SA procedure tends to accept bad moves in an attempt to reduce the probability of being stuck at a locally optimal solution. The *temperature* in the Simulated Annealing is a control parameter which controls the probabilities of accepting bad moves in the iteration [Mitr86]. A general form of Simulated Annealing procedure is given in Figure 2.15. Simulated Annealing has been shown to work well on many difficult applications such as VLSI modules layout and often produces results close to the global optima. However, Simulated Annealing has several drawbacks. First, poor setting of algorithm parameters can cause the process to converge prematurely on sub-optimal solutions or continue searching unnecessarily. Second, since SA relies heavily on random decisions and statistical behavior, Simulated Annealing must perform a larger number of iterations to arrive at good results. This may be time consuming, especially if the time for evaluating a move is large.

Sechen *et al* have successfully applied the Simulated Annealing algorithm to the standard-cell/macro-cell placement problem in different versions of their well-known Timberwolf packages [Sech85] [Sun95]. There are two ingredients in their cost function: total estimated wire length and total sum of overlap penalties. The latter allows illegal layouts during the search, and a legalization is required at the end of the procedure. The cooling schedule is designed as follows: initial temperature is 4000000; final temperature is 0.1; α is set to a value between 0.8 and 0.9. The value of the inner loop size is determined according to the circuit size, e.g., 100 moves for a 200-cell circuit and 700 moves for a 300-cell circuit. The authors report that with these finely tuned parameters, final chip areas are reduced by 15 to 57 % for various benchmarks.

2.5 Parallel/Distributed Processing

Solving large optimization problems such as placement in VLSI CAD design is computationally intensive. Researchers and industries are always seeking efficient algorithms as well as faster hardware to speed up the process. High performance parallel computing can offer enormous computation power that may speed up the algorithms in VLSI CAD design. Parallel computing was extremely expensive in the past but today it is becoming more affordable and cost-effective. This section gives a brief introduction to parallel processing, and summarizes some parallel algorithms for VLSI placement.

2.5.1 Motivation

Advances in semiconductor technology have contributed to the doubling of the number of devices on a silicon chip every 18 months, which follows the prediction by Gordon Moore in 1965 [Rese02]. Along with the continuous increase in circuit complexity, circuit placement is becoming a very computation intensive process.

For example, placement for a large circuit may take hours or even days on a single sequential workstation. However, the fast-paced nature of VLSI CAD design demands advanced computational resources. Therefore, researchers are studying parallel processing to achieve faster performance [Bane94].

Parallel processing is a computing paradigm that allows multiple processors to solve one large problem efficiently. Essentially, the following merits may be offered by the use of this approach for VLSI CAD applications [Bane94]:

- **Solving problems faster:** If a computation task can be effectively processed by multiple processors simultaneously, the parallelism may speed up the task.
- **Solving larger problems:** Parallel systems usually come with larger memories and more computational resources, hence generally they have the capability of solving larger problems.

Besides the above main features, other benefits may possibly be yielding better quality of results and making use of efficient parallel algorithms.

Depending on the nature of the algorithms, many placement algorithms have been proven to be suitable for parallelization, while some others are generally difficult to parallelize effectively.

2.5.2 Parallel Processing Architectures

Flynn's classification scheme of computer architectures is classical and widely accepted [Flyn66]. It has been used for the classification of high-performance computers as well. According to Flynn's definitions, computer systems can be classified into four distinct categories: (1) **SISD**: which stands for single instruction, single data stream and that is where most PCs and workstations fall into this category. (2) **SIMD**: single instruction stream, multiple data stream. Vector machines belong to this class. (3) **MISD**: multiple instruction stream, single data stream. However, this is not a practical architecture [Cosn95]. (4) **MIMD**: multiple instruction stream, multiple data stream. This type of machine is capable of executing independent programs with distinct data simultaneously. Note that SPMD (standing for Single Program Multiple Data) is essentially similar to MIMD since the latter can be made SPMD.

In reality, some parallel systems are SIMD machines, but most are MIMD structures. The latter can be further classified as shared memory systems (e.g., SGI Origin 2000, Cray T3D), distributed memory systems (e.g., Intel Paragon, IBM SP, clusters of networked workstations), and distributed shared memory systems (e.g., HP/Convex Exemplar) in which memory is physically distributed but logically shared.

Advanced parallel machine models such as the computer systems in SHARCNET [SHAR04], are generally organized recursively using clusters in a hierarchical manner [Kuck96]. The advanced computational facilities of SHARCNET in Ontario provides scalable computational resources through a hierarchy of processing capability of over 400 HP/Compaq Alpha processors and large symmetric multiprocessor computers. With the extremely fast speed network ORION, users including researchers at the University of Guelph are able to pursue computationally intensive research on this high-performance facility. The work in Chapter 4 was originally targeted to this distributed/shared memory parallel computing platform. However, due to a compatibility issue with our sequential placer that was originally designed and optimized for Sun Solaris operating systems, we had to temporarily port our parallel implementation to a Solaris-based distributed memory parallel platform, which is a cluster of ten networked workstations. Strategies of parallelizing an algorithm on these two platforms may be quite different, and therefore, Island-based GAs were implemented (see Chapter 4) since they are less influenced by the parallel architectures.

Clusters of networked computer nodes are classified as coarse-grain parallel processors for their distributed local memory and slow interconnection. They are usually not as fast as massively parallel processors. The attraction of clusters lies in the relative low cost of hardware and software as well as the large number of knowledgeable human resources for these lower end computers. Some organizations even use a variety of different computing systems such as personal computers or workstations to form parallel computing platforms and obtain high computation power with a reasonable low cost.

Figure 2.16 illustrates the trends in the fastest 500 computers for the last ten years, which is very interesting [Proj04]. The statistic figure in (a) illustrates the system architectures of these machines, and their overall performance is displayed in (b). Figure 2.16 shows that Massively Parallel Processors (MPP) dominate the high performance computer systems for their super computational power. On the other hand, the percentage of clusters is increasing rapidly while Symmetric Multi Processor (SMP) appears to be dying.

2.5.3 Performance Measure

The performance improvement gained by a parallel implementation is usually measured by the ratio of speedup, which is given by:

$$S_p = \frac{T_1}{T_p} \tag{2.5}$$

where T_p is the execution time for a parallel algorithm using "p" processors, and T_1 is the execution time for a serial algorithm. For the same amount of work, there are two reasons why S_p cannot exceed or be equal to "p" in the real world: (1) the cost for communications and synchronization, and (2) unbalanced work load among processors [Bane94]. Figure 2.17 illustrates the actual S_p and ideal speedup of a p processors parallel system.

Amdahl's law attempts to find out the theoretical limit of speedup [Amda67]. The execution time of an algorithm on a single processor is denoted by T_1 and α is the sequential fraction of the algorithm. As the algorithm is executed on p processors, only the parallel portion can be speeded up:

$$T_p = \alpha T_1 + \frac{(1-\alpha)T_1}{p}$$

Therefore, speedup of p processor parallel system is limited by

$$S_p = \frac{T_1}{T_p} = \frac{1}{\alpha + (1-\alpha)/p} \leq \frac{1}{\alpha} \tag{2.6}$$

Equation 2.6 is derived based on an optimistic assumption that the parallel portion of the algorithm can be evenly distributed among p processors and no other issue would damage the parallelism or cause extra time. In the real world, the performance of parallel systems is limited by architectural, algorithmic, and software factors. Small changes in many aspects, for example work load or I/O bandwidth, can vary an affect the parallel performance.

To measure the average usage of the processors, the efficiency of a parallel algorithm can be found by:

$$E_p = \frac{S_p}{p} \tag{2.7}$$

The better the parallel algorithm is , the nearer E_p is to 1.

However, [Gust90] indicated that in practice, since problem size is reduced in parallelism, a super-linear speedup ($S_p > p$) is possible. Other sources of super-linear speedup include hidden memory latency, subdivision of system overhead, and randomized algorithms [Helm90].

2.5.4 A Message Passing Standard: MPI

Parallel programming models exist as an abstraction level above hardware and memory architectures. Several parallel programming models [Fost95] are commonly used:

- Shared Memory Model
- Thread Model
- Message Passing Model
- Data Parallel Model

- Hybrid Model

The Message Passing model has the following characteristics: (1) during computation, a set of tasks uses its own local memory; (2) tasks exchange data through communications by sending and receiving messages; (3) processors cooperate with each other in performing data transfer. Today, the message passing model has become an expressive, efficient, and well-understood paradigm for parallel programming [Grop96]. The Message Passing Interface (MPI) is a portable software library in common use to support the message passing model, acting as a middle-ware on top of the parallel architectures. The success of MPI causes it to become the *de facto* industry standard for message passing, and almost all major vendors of parallel systems provide support for MPI on their systems. Therefore, the parallel algorithms in this thesis were implemented based on MPI.

MPI is originally a specification developed by the MPI Forum, which is a group of academics, researchers, and software developers. The goal of the specification is to develop a standard for writing efficient, flexible, and portable message-passing parallel programs [Mess04]. The first MPI standard (1.0) was published in 1993, and the most recent version (MPI-2) was released in 1997. The MPI standard is suitable for executing SPMD, MIMI, and MPMD programs. Several implementations of MPI libraries are available publicly, for example, MPICH, CHIMP and LAM. Some system vendors also have their own MPI implementations.

2.5.5 Parallel Algorithms for Standard Cell Placement

Parallel/distributed architectures offer an opportunity to design procedures that explore the solution space more efficiently. Generally speaking, this efficiency may be achieved by accelerating some tedious computational intensive phases of a sequential algorithm via distributed processing. Usually implementations of sequential algorithms need to be rearranged in order to explore and make use of the possible parallelism. In this section, a review of some optimization techniques (*esp.* the ones used in VLSI CAD design) is carried out, explicitly addressing their strategies for parallel implementations.

2.5.5.1 Tabu Search

Malek *et al* implemented a Tabu Search algorithm for the traveling salesman problem (TSP) [Male89]. There are four child processes and one main control process in their implementation. Each child runs a serial Tabu Search algorithm with different parameters. After a specified time interval, the child processes are stopped and the solutions are compared. Bad solutions are discarded and their child processes are restarted with a good solution and their tabu lists are emptied. The authors reported that the parallel TS outperforms the serial implementation.

Taillard [Tail90] proposed a master/slave parallel strategy for Tabu Search. A master process is used to send an initial solution to a number of slave processes. At each iteration, each slave examines part of the neighbourhood and sends the best move to the master process. Having received moves from all slaves the master chooses the best move among them, and synchronizes this move to all slaves. If this move is not tabu, it is performed as the next candidate move. The stopping criteria is identical to the serial Tabu Search algorithm, which is controlled by the master process. In this implementation, parallelism is achieved by the simultaneous neighborhood examinations. In a loosely-coupled parallel platform, this approach can achieve good parallelism if the synchronization does not require a large amount of communication. However, it may not be practical to apply such an approach to the placement of large circuits.

Fiechter [Fiec94] used a decomposition method to parallelize Tabu Search for the same classical TSP problem. For the intensification phase, each process optimizes a portion of the tour. At the end of this phase, processes synchronize their data and recombine the tour and continue processing. For the diversification phase, each process determines promising moves in its portion of the tour and exchanges the data with the others, so that all processes use the same candidate list and apply the moves. The algorithm was implemented on a network of transputers configured as a ring topology. The author claimed that almost linear speedups are achieved, and near-optimal solutions are obtained.

For the application of VLSI standard cell placement, AL-Yamani *et al* [AY02] described a parallelization of TS algorithm on a network of heterogeneous workstations using PVM [PVM 04]. A solution is evaluated by using a fuzzy goal-based cost computation that integrates multiple objectives for cell placement. The algorithm consists of three types of processes: (i) a master process, (ii) Tabu Search Workers (TSWs), and (iii) Candidate List Workers (CLWs). Two levels of parallelization occur simultaneously in the algorithm: (1) the master process initiates a number of TSWs to perform their own Tabu Searches; and (2) each TSW has a number of CLWs for parallel neighborhood examination. Initially, a TSW gets a set of parameters and an initial solution from the master, and performs a diversification step so that the starting search area will not easily overlap with other TSWs. A TSW works somewhat like a master process of its CLWs, using the CLWs to examine neighbourhood solutions in parallel. Reasonable results of faster run-time and better solution quality are reported by the authors, compared to a sequential Tabu Search implementation. This algorithm is actually a combination of *multi-search threads* strategy (by multiple Tabu Search workers) and *functional decomposition* strategy (by multiple candidate list workers for each TSW). While the bandwidth of networks is still a constraint, using more TSWs may produce improved solution quality, and adding more CLWs may result in better solutions in less time.

2.5.5.2 Simulated Annealing Algorithms

The first reported parallel Simulated Annealing algorithm for standard-cell placement was proposed by Retenbar and Kravitz [Krav87], running on a shared memory multiprocessor system. In this method, a move was divided into several subtasks which were then executed in parallel, such as selecting a feasible move, applying block-length penalty, evaluating overlaps, evaluating wire length, and so on. Parallelism is obtained by delegating these subtasks of a move to different processors, and therefore it is very limited. A speedup of 2 on 3 processors was reported with this algorithm. Also, with this approach, the synchronization between subtasks must be performed very carefully, otherwise communications would eliminate the majority of the speedup. The result is that the speedup increases very slowly while more processors are being involved.

Instead of basing it on move decomposition, another Simulated Annealing algorithm proposed by Retenbar and Kravitz parallelizes only the non-interacting moves[Krav87]. To have the maximum parallelism, a maximum set of non-interacting moves must be determined. However, finding this set among all moves is a difficult task. In addition, only independent moves are parallelized and many other possibly good moves are ignored, which leads to poor performance.

Darema *et al* [Dare87] presented a Simulated Annealing algorithm with parallel move evaluation. According to Darema a speedup around 11$\times$ was obtained with 16 processors. However, this algorithm is suitable only for shared memory multiprocessors, not for distributed memory networked workstations.

Banerjee, Jones, and Sargent describe a parallel SA for hypercube-based distributed memory multiprocessors systems [Bane86]. Cells are mapped onto all the processors in the hypercube, using either a grid-wise or row-wise scheme. With such a distributed data structure, parallel moves are supported. The exchanging of pairs of cells connected by direct hypercube links can be evaluated in parallel. To eliminate errors caused by some iteration between the individual moves, synchronization with a global broadcast mechanism is used in each move. This kind of synchronization is rather expensive. Speedups of 11 to 21 with 64 processors were reported.

In [Chan96], the parallel approaches of Simulated Annealing for standard cell placement are summarized as follows: (1) Parallelize the subtasks within a move (has a very limited scope for parallelism). (2) Parallelize the evaluation of multiple moves that do not interact with each other. The disadvantage of such an approach is that many good interacting moves are left behind, and identifying non-interactive moves is difficult. (3) Parallelize all moves, and utilize special methods to control inaccurate information caused by interactive moves. This approach has the maximum scope of parallelism.

For loosely coupled multiprocessors (distributed memory parallel systems with low bandwidth interconnections), several studies to parallelize the Simulated Annealing algorithm were carried out. Partitioning approaches were used to divide a chip into several regions and assign these regions to individual processors. The easiest way to do this is simply by using a basic fixed partitioning scheme. However, since the best positions of some cells may be in other partitions, some cells are not placed in their optimal positions. To solve this problem, Wern-Jieh et al employed a 'dynamic' partitioning method [Sun94]. Instead of being split into fixed areas, a chip is divided vertically into several slices on odd iterations and divided horizontally into several slices on even iterations. This allows a cell to move from one position to any other position in the chip in at most two iterations. After each move of cells, an update must be broadcast to all other processors. The authors reported that this method yields similar or better results than its serial SA while offering nearly linear speedups. The maximum number of workstations used in their experiments was six. It is expected that, if a large number of processors is used, the parallel efficiency will drop dramatically due to the high cost of global synchronization on a low bandwidth parallel system.

2.5.5.3 Parallel Genetic Algorithms

Genetic Algorithms are computational procedures that mimic the natural process of evolution. They have been proven to be able to produce high-quality placement solutions for standard-cell circuits as competitive as those of other sophisticated algorithms [Shah90a] [Klin90]. However, with recent advanced VLSI technologies, circuit sizes are increasing, and the runtime of Genetic Algorithms (relatively slower than other algorithms) is becoming an issue [Kiln89, Shah90b]. Therefore, researchers are studying parallel GAs for better performance.

Similar to natural evolution, (i) individuals are distributed in distinct areas, and they all exist at the same time and evolve in parallel; (ii) the size of subpopulations varies; and (iii) the individuals inside an area usually have much more interaction on than individuals between distinct areas. For most GA applications, the majority of computation power is consumed on the fitness evaluations. The calculation of fitness value for a single individual in the population is usually independent and decoupled from the fitness calculation of other individuals. Therefore, parallel computing techniques can be applied to Genetic Algorithms with high efficiency [Koza98]. Several parallel GAs have been proposed by researchers and can be classified into the following three categories:

1. Global Populations with Parallelism

   Global parallel GA was suggested in the early 70's. Reproduction operators such as crossover, mutation, and the evaluation of parents are relatively independent of the GA operations to other individuals. Therefore the most direct way to parallelize a standard generational GA is to distribute pairs of parents to multiple processors for further processing. One master processor stores information on the whole population, performing selection and mating globally within the population of one generation. If there are N chromosomes in each generation, then $N/2$ slave processors are needed to carry out the GA operations for $N/2$ pairs of parents in parallel. That is, each slave processor holds two parents selected by the master processor and applies crossover, inversion, mutation and evaluation to the parents. Synchronous versions of global parallel GA are commonly used in this class. The term "synchronous" indicates that after a master sends selected individuals to the slaves, it waits to receive the fitness values before proceeding with the next generation. This approach is relatively easy to implement, and a fairly good speedup can be obtained if communication (i.e synchronization) does not dominate the overall execution time. Obviously, a synchronous global parallel GA shares exactly the same searching characteristics as a serial classic GA. Asynchronous global parallel GAs are also possible. With these methods, selection waits until only a fraction of the population has been processed. Global parallelization GA is a considered a fine-grain model that is not suitable for processing by a cluster of networked workstations. Scalability is yet another problem, since exactly 1+N/2 processors are needed.

2. Island Based Models

   One way to have a more coarse-grain parallel model while having better scalability than the global parallel GA is to break down the whole population into subpopulations and distribute them to multiprocessors. These semi-isolated subpopulations execute as normal Genetic Algorithms, except that they swap a few chromosomes occasionally. These are called Island-based Parallel GA. In a simple parallel GA (PGA), without exchanging genetic information, a number of independent GA searches with smaller populations occur, starting from different initial points and moving along various directions in the solution space. These sub-algorithms together are basically the same as the sum of the number of serial GA runs but with smaller population sizes. Since such a simple parallel GA has more potential to converge prematurely, the best search result with this method might be worse than a single serial GA run with a longer population size. However, in an Island-based PGA, subpopulations periodically swap a portion of their information with one another in the model (see Figure 2.18), potential good genes can be transferred from one island to another, adjusting the designated trajectory of an island to a better direction [CP99a]. By introducing migration, Island-based Parallel GAs are able to exploit the differences among the subpopulations, based on the vast explorations by various islands. For the last ten years, many Island-based GA model implementations have been reported to display near-linear speedup or even super-linear speedup, compared to serial GAs [CP98].

3. Cellular Genetic Algorithms

   Cellular Genetic Algorithms are sometimes called "massively-parallel" GAs. A large number of simple processors are laid out on a grid, with edges wrapped around so that the vertex on one side of the grid is adjacent to the vertex on the opposite side (see Figure 2.19) [Whit93]. Each processor (cell) is assigned to one individual. The mating between individuals is restricted to within demes[2]. At each generation, individuals are processed with their GA

[2]A deme is the set of potential mates for an individual, e.g. its neighbours

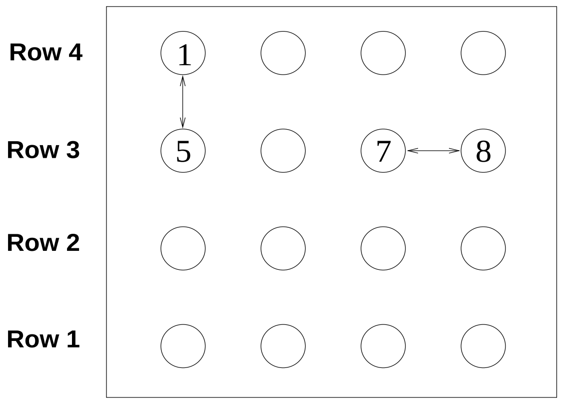
operators in parallel. Various kinds of cellular Genetic Algorithms have been employed in the literature. In a fixed topology approach, a deme of a cell is defined as the individuals in some particular grid locations near this cell. In a random walk approach, a deme is all the individuals along a random walk route started from this cell. In an island based approach, by dividing the grid into multiple subgrids, the algorithms use one of the above methods to choose demes and perform migrations between subgrids for search. For cellular Genetic Algorithms, the subalgorithms in each cell are tightly connected. Therefore, they are suitable for massively parallel hardware, instead of distributed parallel systems.

All the above parallel GAs have some type of locality on their matings. Experiments have shown that parallel GAs benefit from this when solving combinatorial optimization problems, compared to their serial counterpart versions with global mating [Gord93].

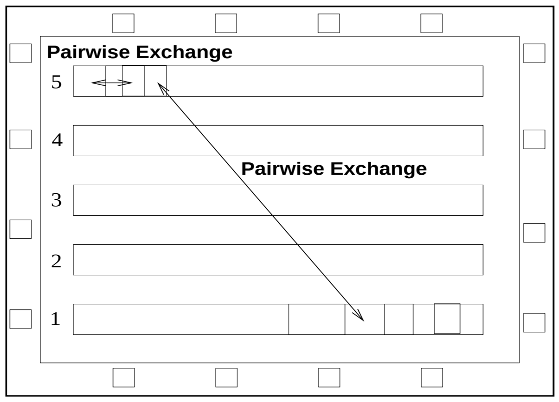
2.6 Summary

As a critical step in the VLSI design cycle, physical design is incredibly a complex process that has to be decomposed into more tractable subtasks such as partitioning, placement, routing. Approaches used by physical design automation highly depends on various layout styles. This thesis concentrates on the NP-hard placement problem of standard cell circuits. Several generic heuristic techniques are reviewed, namely Tabu Search, Simulated Annealing and Genetic Algorithm. Since solving large circuit placement is computationally intensive, some approaches of paralleling the above heuristic algorithms are discussed.

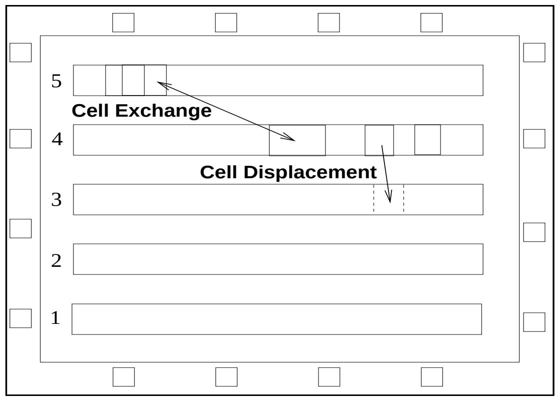
In the following chapters some of these heuristic based techniques as well as a parallel implementation of the Genetic Algorithm are implemented for standard cell placement.



(a) Adjacent Pairwise Exchange



(b) Pairwise Exchange



(c) Module Perturbation

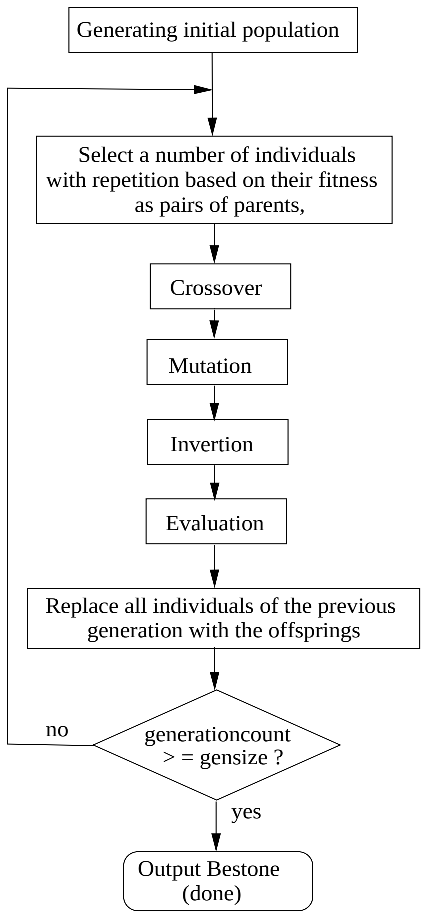
Figure 2.11: Module Interchange Methods

```
(1) Choose an initial solution $s$
    $s* := s$

(2) while (the stopping criteria is not satisfied)
        Generate a candidate set $V* \subseteq N(s) - T$
        Find the best $s'$ in $V*$
        $s := s'$
        if $f(s') < f(s*)$ then
            $s* := s'$
        Update tabu memory T
    end while

(3) return $s*$
```

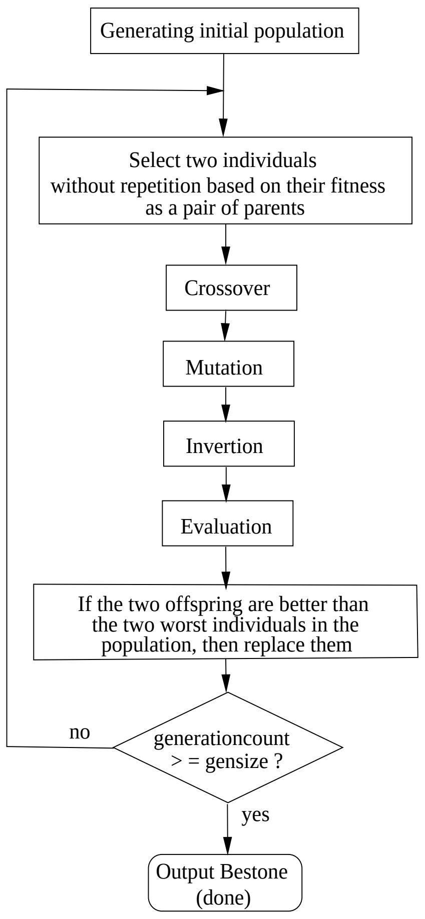
Figure 2.12: A General Form of Tabu Search

```
(1) Choose an initial placement solution $s$
    initialize $s$, $|T|$, $nbiter$, $bestsolution$, $bestvalue$, $bestiter$

(2) while $(nbiter - bestiter) < nbmax$
        $nbiter{+}{+}$
        Generate the move $s' \in N(s)$ and $s$ is not in Tabu list $T$
                Randomly select a module $m_i$
                Find a best target module $m_j \in N_\delta(m_i)$ for interchange,
                and module pair $(m_i, m_j)$ is not in Tabu list $T$
        if $f(s') < bestvalue$ then
                $bestvalue := f(s')$, $bestsolution := s'$, $bestiter := nbiter$
        Update tabu memory T
        Perform the move
    end while

(3) return $bestsolution$
```

Figure 2.13: A Placement Optimization Approach Using Tabu Search [Song92]


Generating initial population
Select a number of individuals
with repetition based on their fitness
as pairs of parents,
Crossover
Mutation
Invertion
Evaluation
Replace all individuals of the previous
generation with the offsprings
no
generationcount
> = gensize ?
yes
Output Bestone
(done)

(a) A Generational GA

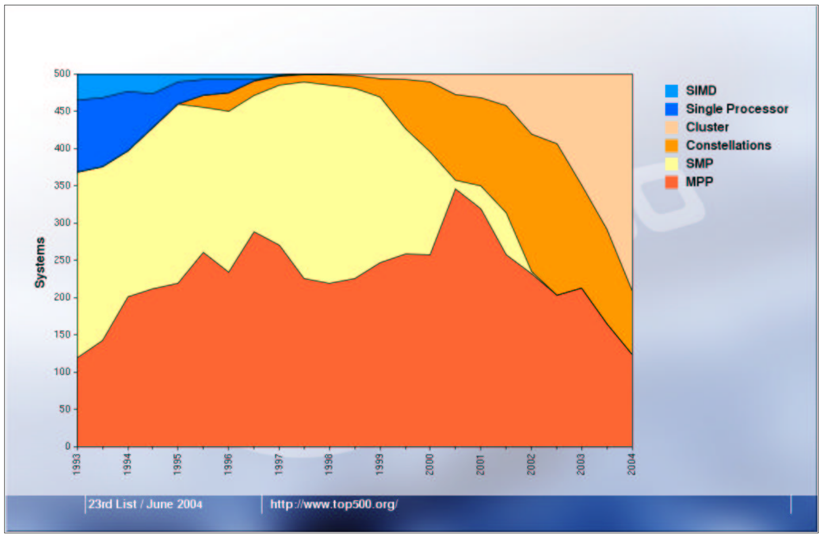

Generating initial population
Select two individuals
without repetition based on their fitness
as a pair of parents
Crossover
Mutation
Invertion
Evaluation
If the two offspring are better than
the two worst individuals in the
population, then replace them
no
generationcount
> = gensize ?
yes
Output Bestone
(done)

(b) A Steady-State GA

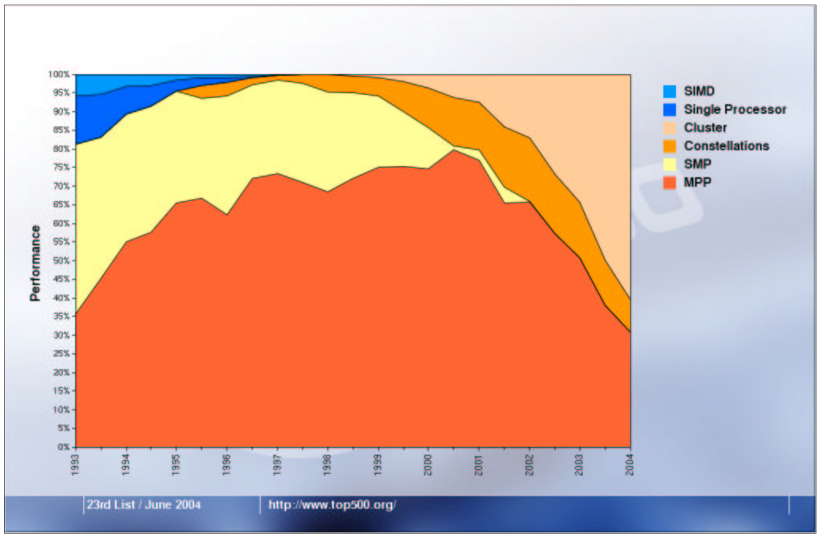
Figure 2.14: Flowchart of GAs

```
(1) Choose an initial solution $s := s_0$
    $T := T_0$

(2) while (the stopping criteria is not satisfied)
        while (not yet in equilibrium)
            Generate a random neighbour $s'$ from $s$
            if $(C(s') < C(s))$ then $s := s'$
            else if $(e^{\frac{C(s)-C(s')}{T}} \geq random(0,1))$ then $s := s'$
            if $(C(s') < C(s*))$ then $s* := s'$
        end while
        $T := next\_temp(T) \approx \alpha T$
    end while

(3) return $s*$
```

Figure 2.15: A General Form of SA



(a) Architectures, by [Proj04]



(b) Performance, by [Proj04]

Figure 2.16: Trends in the Fastest Computers since 1993

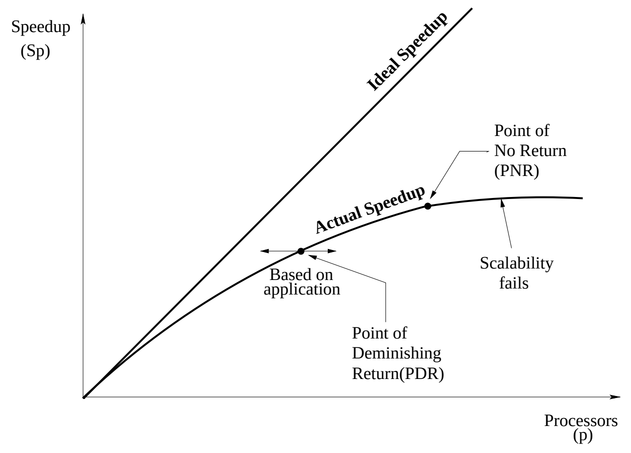


Figure 2.17: Performance of Parallel Systems [Kuck96]

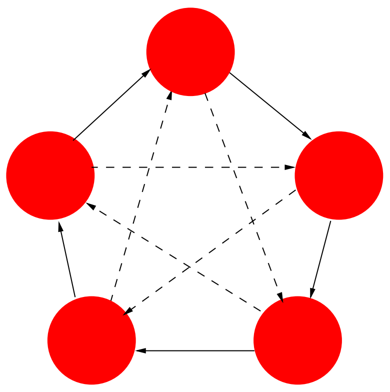


Figure 2.18: An Example of Island-based GA

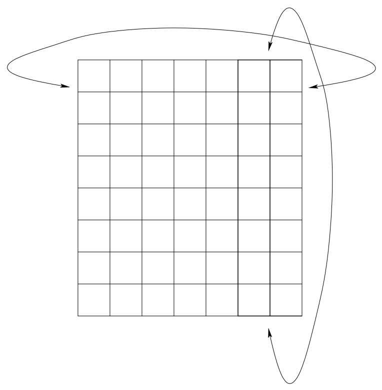


Figure 2.19: An Example of Cellular Genetic Algorithms

Chapter 3

Local Search and Meta Heuristics

The input to any placer tool is a set of standard cells and netlist information. The goal of the placer is to produce an optimized layout for the given cells. Initially, a parser is required to properly read, parse and store the circuit information. The data structure used is usually very complicated yet it is extremely important since it significantly affects the efficiency of the algorithms used. A good data structure sometimes demand less computing resources and may be easier to manipulate. Figure 3.1 illustrates the basic work flow of a typical VLSI placer.

The algorithms of the placer mainly involves the following two stages:

1. **Constructively generating one or more good initial solutions**;

2. **Iteratively improving placement solutions**.

This chapter focuses on the iterative improvement phase for circuit placement. Several heuristic algorithms are implemented in this dissertation to improve solution quality for this NP-hard problem, based on the cell displacement and cell exchange

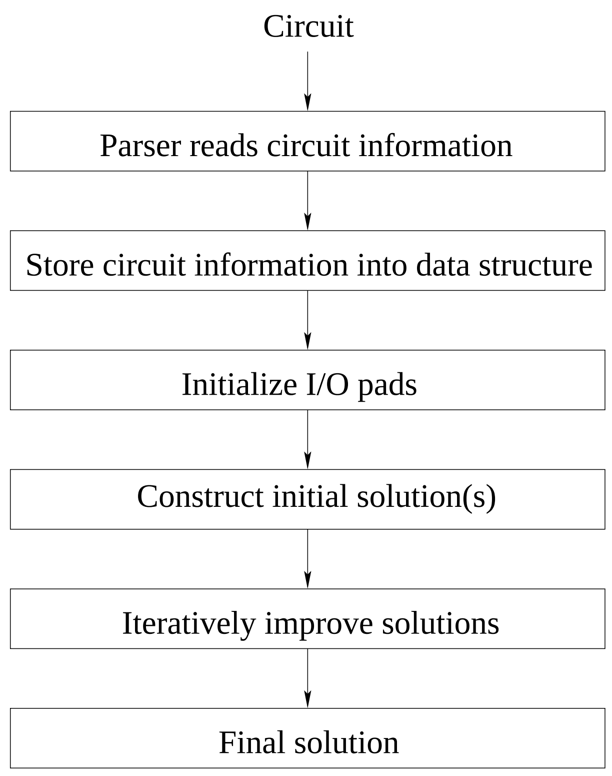

Circuit
Parser reads circuit information
Store circuit information into data structure
Initialize I/O pads
Construct initial solution(s)
Iteratively improve solutions
Final solution

Figure 3.1: The Flow Chart of the SC3 Placer

approaches.

The total wire length of a layout is approximately estimated by an efficient half-perimeter method in the implementation. Two local search algorithms based on different placement strategies are discussed and implemented in Section 3.1. Results obtained show that the two placement strategies can produce comparable results however the second approach executes in less CPU time. Since these search algorithms tend to get trapped in local optima, meta-heuristic algorithms are further developed in the forms of Tabu Search, Simulated Annealing and Genetic Algorithm. These advanced search heuristics attempt to guide local search techniques to effectively explore and exploit the solution space efficiently.

3.1 Local Search Algorithms

To minimize a cost function, Local Search algorithms start from an initial placement solution, and iteratively attempt to obtain a better solution by finding the best neighbourhood solution of the current solution. If a move yields less cost in the objective function then it is accepted and the search continues; otherwise the Local Search stops. For such hill-climbing heuristics, a neighbourhood is created by performing one of the following two types of module interchanges:

- **Cell displacement**: moves a cell from its current location and places it to a random new location on the chip.
- **Cell exchange**: swaps the locations of any two cells.

Cell displacement and cell exchange can happen in the same row or in different rows. A third type of move can be used to locally modify a placement solution, which is based on changing the module orientation. Optimizing cell orientation will be discussed and implemented in Chapter 5.

The following context describes the implementation of two placement strategies based on Local Search Algorithm: the first directly minimizes the wire length and requires shifting some modules at each move, while the other allows overlaps and legalizes the solution in the final phase of the procedure.

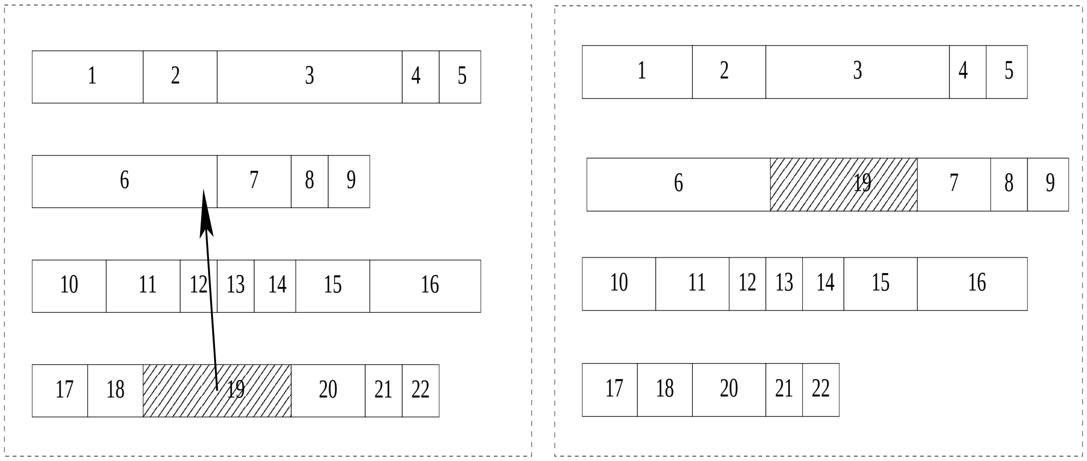
3.1.1 Strategy-I

Minimizing the cost function of Equation (2.3) will minimize the total chip area, therefore this equation can be directly used to evaluate the acceptability of new neighbour solutions in a Local Search. A simple Local Search algorithm implemented is illustrated in Figure 3.2. Starting with an initial placement solution, the algorithm continuously finds the best neighbour and performs moves until no further improvement can be achieved.

```
Local Search Algorithm #1
Input:
The net list of the circuit

Initialization:
Choose an initial placement solution

Mainloop:
while(1)
    BestImprovement = -1
    for( i = 0; i < NeighbourhoodSize; i++ )
        CurrentNeighbour = Generate_Legal_Interchange()
        if ( CurrentNeighbour == NULL ) then
            break
        improvement = Evaluate_the_Interchange()
        if ( improvement > BestImprovement ) then
            BestImprovement = improvement
            BestNeighbour = CurrentNeighbour
    end for
    if ( BestImprovement ≥ 0 ) then
        Perform the interchange of BestNeighbour
        Update circuit information
        CurrentSolution = BestNeighbour
    else break
end while

Output:
CurrentSolution
```

Figure 3.2: Local Search Implementation I

A neighbour solution is generated by virtually performing either a "cell displacement" or a "cell exchange" from the current solution. A valid solution is an arrangement of the given cells into rows of layout area without any overlap or empty space between adjacent modules. To ensure the validity of the solution, this algorithm inspects the following: (1) modules are placed into adjacent rows, (2) modules are placed inside the layout area, and (3) no module overlaps nor empty space between adjacent modules. Shifting of some affected cells usually has to be performed after changing the location of a module to make room or remove overlap. An illustration of cell displacement is presented in Figure 3.3 where cell 19 virtually moves to a location in row #2. Modules 7, 8, and 9 will have to shift to the right

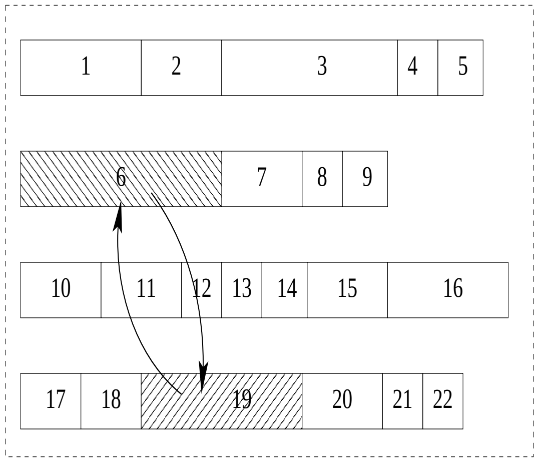


(a) Before Move (b) After Move

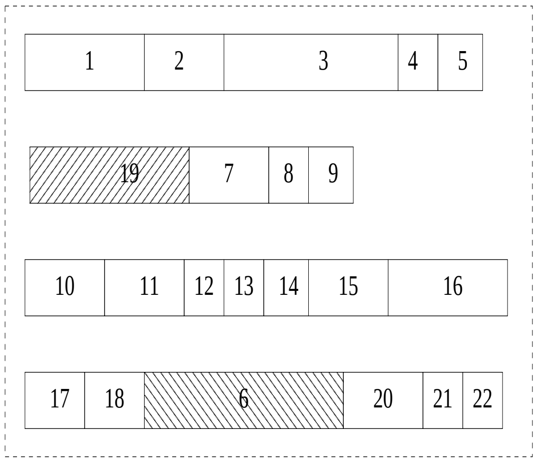
Figure 3.3: Cell Displacement

to make room for the incoming cell, while modules 20, 21, and 22 will shift to the left to eliminate the empty space. An illustration of the "cell exchange" concept is presented in Figure 3.4 where module 6 and 19 are exchanged. Since module 6 is larger than 19, modules 7 - 9 shift to the left and modules 20 - 22 shift to the right. When a new move is generated for evaluation, affected cells are virtually shifted. If the best move is accepted, shifting is actually performed.

The function *Generate_Legal_Interchange()* in Figure 3.2 creates a new neigh-



(a) Before Move



(b) After Move

Figure 3.4: Cell Exchange

bourhood configuration by randomly selecting one of the above two types of moves, with a probability determined by a user-specified “SWAP/MOVE” ratio. The value of the “SWAP/MOVE” ratio can obviously affect the final search result, since it determines the quality of solutions generated.

The algorithm uses the Half-Perimeter method to estimate the wire length of a net. More specifically, the function *Evaluate_the_Interchange()* in Figure 3.2 determines the affected nets due to relocation of the cells, and evaluates the ΔC of these nets based on this method. In each move, many modules are displaced or exchanged or shifted, and all the nets connected to these modules are affected. To evaluate the cost change of a neighbourhood solution, the involved computational effort is not trivial.

3.1.2 Strategy-II

The previous algorithm can produce good results but the main disadvantage is attributed to the shifting of modules when evaluating a neighbour solution. This may affect a larger number of nets, resulting in excessive CPU time. To clarify this issue, assume each row in a circuit has 80 cells. Averagely, generating a neighbour by either cell displacement or cell exchange virtually shifts 80 cells, and they may affect a large number of nets.

Therefore, a second Local Search strategy is implemented as shown in Figure 3.5. Instead of consuming time in shifting the modules, this algorithm allows overlaps of the modules without shifting them. These overlaps are penalized in the cost function. A legalizing procedure is then executed to convert the final solution into a valid placement. Based on this strategy, the second algorithm may save significant computation effort compared to the previous one.

Similar to Strategy-I, the algorithm estimates the wire length of all nets based on the half-perimeter of the bounding box. However, it allows illegal configurations and penalizes them in the cost function. These illegal configurations include overlaps between modules in the same row, overshoot or undershoot of row length over the ideal row length, and overcrowded modules in rows. This dramatically eliminates the high computation demand for shifting modules and evaluating the affected nets. Another possible benefit of Strategy-II is that it gives more flexibility to explore the solution space. A cell can be placed anywhere as long as this move reduces more wire length than other neighbour solutions.

In the implementation of the second strategy, the cost function is the sum of

```
Local Search Algorithm 2
Input:
The net list of the circuit

Initialization:
Choose an initial placement solution

Mainloop:
while(1)
    $BestImprovement = 0$
    for( i = 0; i < NeighbourhoodSize; i++ )
        CurrentNeighbour = Generate_an_Interchange()
        improvement = Evaluate_the_Interchange()
        if ( improvement > BestImprovement ) then
            BestImprovement = improvement
            BestNeighbour = CurrentNeighbour
    end for
    if ( BestImprovement $\geq 0$ ) then
        Perform the interchange of BestNeighbour
        CurrentSolution = BestNeighbour
    else break
end while
Legalize_Circuit()

Output:
CurrentSolution
```

Figure 3.5: Local Search Implementation II

the following four terms: (i) C_1, wire length cost, (ii) C_2, module overlap penalty, (iii) C_3, row length undershoot and overshoot penalty, and (iiii) C_4, modules overcrowded penalty. The overall cost function is given by:

$$C = C_1 + C_2 + C_3 + C_4 \tag{3.1}$$

The first term C_1 is the wire length cost, which is estimated using the Half-

Perimeter method (HPWL).

$$C_1 = \sum_{i \in nets} [W_H span\_x(i) + W_V span\_y(i)] \tag{3.2}$$

Independent weights can be assigned to horizontal and vertical wire length to give preference in one direction over the other. For example, assigning a slightly larger weight to W_V than W_H sometimes helps to produce a better final solution.

The second term C_2 is the penalty of overlaps between adjacent modules in the same row. Two cost functions of C_2 are considered, where W_2 is the weight of the overlap penalty:

$$C_2 = W_2 \sum_{i \neq j} [O_{i,j}] \tag{3.3}$$

$$C_2 = W_2 \sum_{i \neq j} [O_{i,j}]^2 \tag{3.4}$$

Without shifting, moving a cell may introduce overlaps to the destination or release some overlaps from the source location. Similarly, an exchange of two cells that differ in size may result in change of overlaps. The quadratic Equation (3.4) applies a larger penalty on large overlaps than on small ones, while Equation (3.3) penalizes the size of overlaps linearly.

The third term C_3 compares the actual row length with the desired row length, and applies a penalty to the amount of overshoot or undershoot of these rows over the desired row length:

$$C_3 = W_3 \sum_{i \in rows} |row\_length(i) - Desire\_row\_length(i)| \tag{3.5}$$

where W_3 is the weight of row length penalty.

In the TimberWolf placement package [Sech85], the cost function consists of three terms that are similar to C_1, C_2 and C_3 described above. In our work, the cost function further incorporates C_4. In an attempt to prevent the algorithm from placing too many small modules in a single row, a penalty must be applied if the number of modules in a row exceeds the allowable maximum number. The fourth term C_4 is given by:

$$C_4 = W_4 \sum_{i \in rows} RM(i) \tag{3.6}$$

where

$$RM(i) = \begin{cases} 0, & \text{if } Modules(i) \leq Max\_Mods; \\ \text{Modules(i) - Max\_Mods}, & \text{if } Modules(i) > Max\_Mods. \end{cases} \tag{3.7}$$

where W_4 is the weight of the modules overcrowded penalty, $Modules(i)$ is the number of modules in row i, and Max_Mods is the allowable maximum number of modules that can be placed in one row. Max_Mods is determined by the specification of a standard-cell circuit.

The function *Generate_an_Interchange()* generates a neighbour from a current solution by virtually performing either a cell displacement or a cell exchange. The function randomly produces a new move according to the SWAP/MOVE ratio and neglects module shifting and validity checking.

The function *Evaluate_the_Interchange()* evaluates the improvement $-\Delta C$ for a neighbour solution. Using the HPWL method, the decrease in wire length of affected nets is calculated for the term $\Delta C1$. Since only one or two cells change their locations at each move, the computation for wire length estimation is trivial. The calculations of $\Delta C2$, $\Delta C3$ and $\Delta C4$ are not computationally demanding either since none of them involves large numbers of nets or modules. The function *Evaluate_the_Interchange()* is illustrated in Figure 3.6.

In each iteration, the Local Search algorithm generates a set of new neighbour solutions and evaluates the possible improvement for each of them. If a cell displacement or cell exchange does not increase in cost function, it is accepted and performed. Shifting is not required after each move. The search continues until no further improvement can be found. At the final stage of the algorithm, a legalizing procedure is performed to convert the illegal solution to a legal placement that meets layout specifications. Figure 3.7 illustrates the four stages in the function *Legalize_Circuit()*. These stages make the layout solution legal, while attempting to impact the solution quality as little as possible.

Local Search methods typically perform blind search where they sequentially accept good moves that yield improvements in the objective function. Local Search heuristics terminate as soon as the first local minima is encountered. The quality of search results relies heavily on the initial solution, and the local optimum can be very far away from the global optima. Several techniques are usually used in an attempt to overcome this problem. One of these approaches is to use multiple starting points. Another approach is to dynamically change the size of neighbourhood. However, the above two methods are not effective enough in helping the search nor robust enough, due to their lack of "intelligence" and "knowledge" of the search. Meta-heuristics are advanced algorithms that can gather crucial information necessary to guide local search explore and exploit the solution space

```
procedure Evaluate_the_Interchange()
comment: $\Delta C = \Delta C1 + \Delta C2 + \Delta C3 + \Delta C4$
if ( new_configuration == MOVE cell i to the same row ) then
    Evaluate $C_1$ (wirelength of nets connected to cell i)
    Evaluate $C_{2\_i\_original}$ (overlaps of area covered original location of cell i)
    Evaluate $C_{2\_i\_new}$ (overlaps of area covered by new location of cell i)
    Virtually move cell i to the new location
    Evaluate $C'_1$ (wirelength of nets connected to cell i)
    Evaluate $C'_{2\_i\_original}$ (overlaps of area covered by original location of cell i)
    Evaluate $C'_{2\_i\_new}$ (overlaps of area covered by new location of cell i)
    $\Delta C_1 = C'_1 - C_1$
    $\Delta C_2 = (C'_{2\_i\_original} + C'_{2\_i\_new}) - (C_{2\_i\_original} + C_{2\_i\_new})$
    $\Delta C_3 = 0$, $\Delta C_4 = 0$

if ( new_configuration == MOVE cell to a different row ) then
    Evaluate $C_1$ (wirelength of nets connected to cell i)
    Evaluate $C_{2\_i\_original}$ (overlaps of area covered original location of cell i)
    Evaluate $C_{2\_i\_new}$ (overlaps of area covered by new location of cell i)
    Evaluate $C_3$ (row length penalty of both rows)
    Evaluate $C_4$ (modules overcrowded penalty of both rows)
    Virtually move cell i to the new location
    Evaluate $C'_1$ (wirelength of nets connected to cell i)
    Evaluate $C'_{2\_i\_original}$ (overlaps of area covered by original location of cell i)
    Evaluate $C'_{2\_i\_new}$ (overlaps of area covered by new location of cell i)
    Evaluate $C'_3$ (row length penalty of both rows)
    Evaluate $C'_4$ (modules overcrowded penalty of both rows)
    $\Delta C_1 = C'_1 - C_1$, $\Delta C_3 = C'_3 - C_3$, $\Delta C_4 = C'_4 - C_4$
    $\Delta C_2 = (C'_{2\_i\_original} + C'_{2\_i\_new}) - (C_{2\_i\_original} + C_{2\_i\_new})$

if ( new_configuration == SWAP cells i and j in the same row ) then
    Evaluate $C_1$ (wirelength of nets connected to cell i and/or cell j)
    Evaluate $C_{2\_i\_original}$ and $C_{2\_i\_new}$
    Evaluate $C_{2\_j\_original}$ and $C_{2\_j\_new}$
    Virtually move cell i to the new location
    Evaluate $C'_1$ (wirelength of nets connected to cell i and/or cell j)
    Evaluate $C'_{2\_i\_original}$ and $C'_{2\_i\_new}$
    Evaluate $C'_{2\_j\_original}$ and $C'_{2\_j\_new}$
    $\Delta C_1 = C'_1 - C_1$
    $\Delta C_2 = (C'_{2\_i\_original} + C'_{2\_i\_new}) - (C_{2\_i\_original} + C_{2\_i\_new}) + (C'_{2\_j\_original} + C'_{2\_j\_new}) - (C_{2\_j\_original} + C_{2\_j\_new})$
    $\Delta C_3 = 0$, $\Delta C_4 = 0$

if ( new_configuration == SWAP cells i and j in different rows ) then
    Evaluate $C_1$ (wirelength of nets connected to cell i and/or cell j)
    Evaluate $C_{2\_i\_original}$ and $C_{2\_i\_new}$
    Evaluate $C_{2\_j\_original}$ and $C_{2\_j\_new}$
    Evaluate $C_3$ (row length penalty of both rows)
    Virtually move cell i to the new location
    Evaluate $C'_1$ (wirelength of nets connected to cell i and/or cell j)
    Evaluate $C'_{2\_i\_original}$ and $C'_{2\_i\_new}$
    Evaluate $C'_{2\_j\_original}$ and $C'_{2\_j\_new}$
    Evaluate $C'_3$ (row length penalty of both rows)
    $\Delta C_1 = C'_1 - C_1$, $\Delta C_3 = C'_3 - C_3$, $\Delta C_4 = 0$
    $\Delta C_2 = (C'_{2\_i\_original} + C'_{2\_i\_new}) - (C_{2\_i\_original} + C_{2\_i\_new}) + (C'_{2\_j\_original} + C'_{2\_j\_new}) - (C_{2\_j\_original} + C_{2\_j\_new})$

return improvement $= -(\Delta C_1 + \Delta C_2 + \Delta C_3 + \Delta C_4)$
```

Figure 3.6: Evaluate_the_Interchange() Procedure

```
procedure Legalize_Circuit()

Stage 1 for ( i = 0; i < All_Rows; i++ )
            Shift the modules to remove overlaps

Stage 2 for ( i = 0; i < All_Rows; i++ )
            if the total width in row i is more than
            the maximum row length allowed, then
                move the cells having the least
                connection in row i to the row j
                having the shortest row length.

Stage 3 for ( i = 0; i < All_Rows; i++ )
            if the total modules in row i is more
            than the maximum number allowed, then
                move the smallest cells to the row j
                having the least cells.
                if the above move is not suitable, then
                    find out the row k which having the
                    least number of cells, and swap the
                    largest width cell in the row i with
                    the smallest cell in the row k;
                    i = i - 1

Stage 4 for ( i = 0; i < All_Rows; i++ )
            Shift the modules to remove overlaps
        Update circuit information
```

Figure 3.7: Legalize_Circuit() Procedure

effectively. Section 3.2 and 3.3 will present implementations of several advanced meta-heuristic algorithms for standard cell placement.

3.1.3 Experimental Results

The algorithms were implemented based on the C programming language, and compiled using a GNU C compiler. To study the implemented algorithms, a large number of experiments are conducted on Sun Solaris operating system.

Table 3.1 lists the general information of the benchmark circuits that were used in the experiments in this thesis. These circuits are the standard MCNC '91 Benchmarks used for standard cell placement [Kozm91]. The suite includes ten standard cell circuits ranging from 125 to 25114 cells. According to the size of the circuits, benchmarks are grouped into small, medium and large classes. The second column of the table shows the name of the circuit. The third and the fourth columns show the number of cells and the number of I/O pads that connect the circuit to the outside world. The number of nets are presented in the sixth column. The total number of pins of all modules are summarized in the seventh column. The last column provides the number of rows where the cells are to be placed. Figure 3.8 illustrates a standard cell circuit.

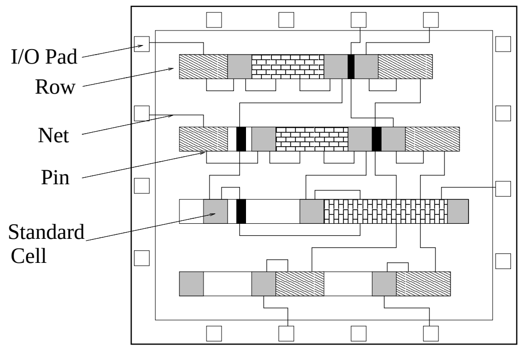


Figure 3.8: A MCNC Standard Cell Circuit

Performance evaluations of the developed algorithms are based on total wire

Group	Circuit	Cells	IO Pads	Total modules*	Nets	Pins	Rows**
Small	Fract	125	24	149	147	876	6
	Prim1d	752	81	833	876	5614	16
	Struct	1888	64	1952	1920	10814	21
Medium	Ind1	2271	814	3085	2478	19186	15
	Prim2	3014	107	3121	3136	22371	28
	Bio	6417	97	6514	5742	41886	46
Large	Ind2	12142	495	12637	13419	95818	72
	Ind3	15059	374	15433	21938	136084	54
	avq.s	21854	64	21918	22124	152334	80
	avq.l	25114	64	25178	25384	165374	86

Note:
* $TotalModules = Cells + IOPads$
** Rows: The number of rows is specified by a parameter at run time.

Table 3.1: MCNC Benchmarks Used in the Experiments

length obtained and CPU time consumed to achieve final results. Since the initial placement solutions are different, using only the wire length of final solutions may not be sufficient. Therefore, the percentage of improvement is also used to measured the search quality. Most experimental results in this chapter are based on initial random placement solutions. Each test was conducted 20 times and the average values were recorded. Note, the "WL" in the tables of this chapter stands for Wire Length, and "%Imp" refers to percentage of improvement.

3.1.3.1 Comparison of the Two Strategies

The first set of experiments were conducted to compare Strategy-I (minimizes total wire length, by generating only legal solutions) and Strategy-II (minimizes cost function (3.1), allowing infeasible solutions with penalty introduced).

Table 3.2 presents the placement results based on the two strategies. In the experiments, the neighbourhood size is a value proportional to the size of the circuits, that is, $Neighbours = N_{cells}/20$ (N_{cells} is the number of modules of a circuit). It is evident that Strategy-I is more CPU-demanding than Strategy-II. For example circuit *ind3*, the time required by Strategy-I was 46 times longer than that based on Strategy-II. However, solution qualities based on Strategy-I were superior.

Circuit	Strategy-I			Strategy-II		
	WL	%Imp	Time	WL	%Imp	Time
fract	83049	3.58%	0.06	84006	2.47%	0.03
prim1	2.38×10^6	7.62%	0.33	2.49×10^6	3.29%	0.13
struct	1.74×10^6	29.34%	8.20	1.97×10^6	20.22%	1.00
ind1	4.02×10^6	34.94%	37.50	4.40×10^6	28.78%	2.15
prim2	1.22×10^7	26.80%	74.60	1.34×10^7	19.13%	1.55
bio	7.69×10^6	41.44%	503.63	8.26×10^6	37.07%	40.49
ind2	5.10×10^7	45.24%	5093.67	5.65×10^7	39.39%	214.62
ind3	1.56×10^8	44.53%	11302.80	1.68×10^8	40.33%	245.42
avq.s	-	-	very long	4.20×10^7	52.43%	9711.05
avq.l	-	-	very long	4.63×10^7	54.17%	12679.85
Note: The neighbourhood size is $N_{cells}/20$.						

Table 3.2: Comparison of the Two Placement Strategies (part I)

To further examine the search capability of the two strategies, a second set of experiments were conducted by varying the neighbourhood size (the results are very interesting). When the neighbours size was small (*e.g.* less than 200), Strategy-I produced better solutions than Strategy-II (see Table 3.3). However, when the size increased to 200, Strategy-II performed better. The data is plotted in Figures 3.9 and 3.10, where the dashed lines denote the improvement and the computation time of Strategy-I over varied neighbourhood sizes, and the solid lines denote those of Strategy-II. The results clearly indicate that (i) Strategy-II can yield better solutions than Strategy-I if the neighbour size is not too small; and (ii) Strategy-I requires much more computation time than Strategy-II. Similar results were obtained with the rest of the benchmarks.

Circuit	Neighbors	Strategy-I		Strategy-II	
		Improvement	Time	Improvement	Time
struct	20	1.12%	0.89	1.85%	0.8
	40	9.20%	1.66	5.17%	0.82
	80	23.56%	5.37	21.30%	1.01
	100	30.27%	8.95	25.95%	1.12
	200	39.55%	23.42	40.77%	1.77
	500	48.92%	80.85	52.63%	5.04
	1000	54.91%	220.37	59.35%	12.87
	2000	58.57%	532.32	66.17%	32.15

Table 3.3: Comparison of the Two Placement Strategies (part II)

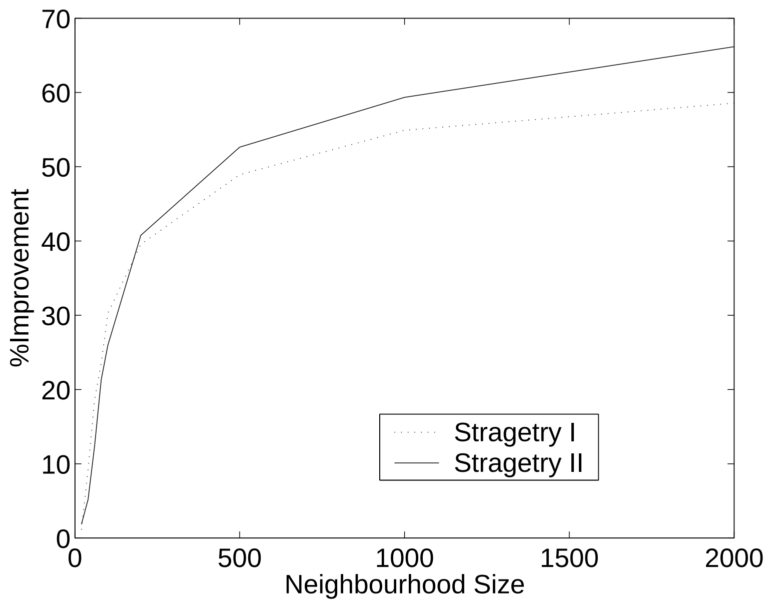


Figure 3.9: Strategy-I *vs.* Strategy-II (%Improvement)

Since Strategy-II can produce better results in less CPU time, the Meta-heuristics developed later such as Tabu Search and Simulated Annealing will be based on this

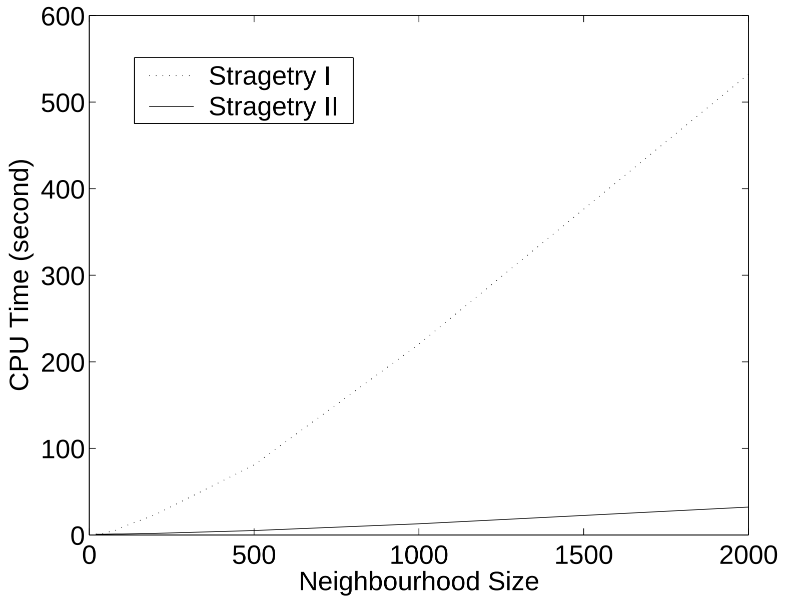


Figure 3.10: Strategy-I *vs.* Strategy-II (CPU Time)

strategy.

3.1.3.2 Legalization Procedure

Since a legalization procedure is required for Strategy-II, a set of tests were conducted to experiment how the *Legalize_Circuit()* function affects the final solutions, as Table 3.4 and 3.4 shown.

In the experiments, the neighbourhood size of Local Search was determined by circuit size (i.e. $Neighbours = N_{cells}/30$). X and Y are the wire length in the x and y direction, respectively. The initial solutions are shown in the second and the third columns of Table 3.4, and the wire length obtained by Local Search are shown in the fourth and the fifth columns. The last column displays the percentage of improvement. On the other hand, the wire length after the legalization stage are

	Initial Placement		After LS		% Imp by
	X	Y	X	Y	LS
fract	44257	45356	44058	45356	0.2
prim1	2.00×10^6	5.72×10^5	1.96×10^6	5.72×10^5	1.6
struct	2.00×10^6	5.30×10^5	1.25×10^6	5.30×10^5	29.4
ind1	5.36×10^6	8.10×10^5	2.44×10^6	8.10×10^5	47.3
prim2	1.36×10^7	3.03×10^6	8.76×10^6	3.03×10^6	29.0
bio	1.16×10^7	1.55×10^6	5.30×10^6	1.55×10^6	47.8
ind2	7.93×10^7	1.44×10^7	3.12×10^7	1.44×10^7	51.3
ind3	2.52×10^8	2.97×10^7	7.87×10^7	2.97×10^7	61.6
avq.s	8.20×10^7	6.41×10^6	2.57×10^7	6.41×10^6	63.6
avq.l	9.45×10^7	6.59×10^6	3.08×10^7	6.59×10^6	63.0

Table 3.4: Results *before* and *after* the Legalizing Procedure

	After Legalization		% Imp by	% Imp
	X	Y	*Legalizing*	(final)
fract	43465	44776	1.3	1.5
prim1	1.97×10^6	5.72×10^5	-0.4	1.2
struct	1.37×10^6	5.69×10^5	-6.3	23.1
ind1	3.06×10^6	9.38×10^5	-12.1	35.2
prim2	9.79×10^6	3.40×10^6	-8.5	20.5
bio	6.10×10^6	2.01×10^6	-9.6	38.2
ind2	3.98×10^7	1.47×10^7	-9.5	41.8
ind3	1.29×10^8	3.53×10^7	-19.9	41.7
avq.s	3.31×10^7	9.26×10^6	-11.6	52.0
avq.l	3.93×10^7	1.03×10^7	-12.1	50.9

Table 3.5: Results *before* and *after* the Legalizing Procedure (Continue)

shown in Table 3.5. Results indicate that the *Legalize_Circuit()* function tends to deteriorate the total wire length. In addition, it can be seen that the larger the improvement obtained by the previous stages (iterative improvement algorithms), the more deterioration by the *Legalize_Circuit()* function. For the *bio* circuit, although Local Search decreased the total wire length by 47.8%, the legalization increased the wire length by 9.6% and yielded an overall improvement of 38.2%. Figure 3.11 illustrates the improvements achieved by Local Search algorithm and the overall improvement following the legalization step.

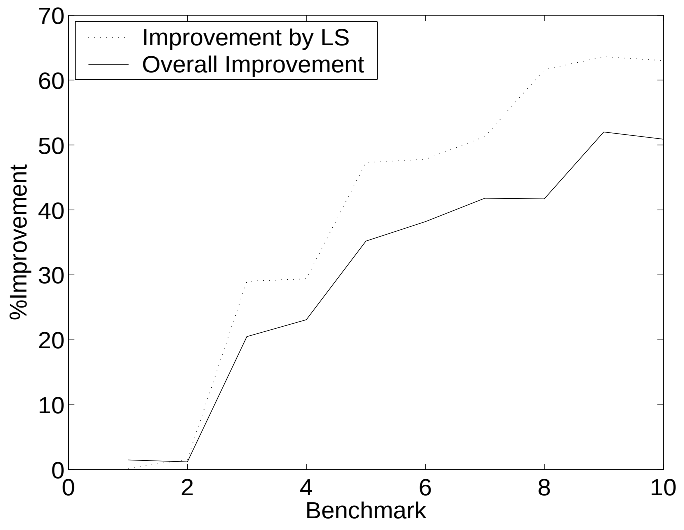


Figure 3.11: Effect of Legalization

3.1.3.3 The MOVE/SWAP Ratio

When generating a neighbourhood solution, Local Search randomly selects one of the two types of module interchanges ("cell displacement" and "cell exchange") with a probability determined by the MOVE/SWAP ratio. Since this ratio affects search quality, several experiments were conducted in an attempt to find out the most suitable value.

With a ratio ranging from 1 to 10, a small circuit (*fract*), a medium circuit (*struct*) and a large circuit (*ind2*) were tested (see Table 3.6). However, the results are not conclusive since no particular value is found to be ideal for all the

Ratio	fract		struct		ind2	
	Improvement	Time	Improvement	Time	Improvement	Time
1	11.65%	0.12	10.16%	1.76	9.81%	113.90
2	41.28%	0.10	31.10%	1.49	26.25%	97.82
3	37.09%	0.10	31.56%	1.53	25.57%	95.53
4	38.25%	0.14	27.12%	1.38	23.97%	88.60
5	34.78%	0.09	32.16%	1.56	26.41%	93.31
6	34.11%	0.07	31.29%	1.52	28.44%	97.76
7	31.69%	0.09	32.48%	1.60	27.01%	95.83
8	36.45%	0.10	26.39%	1.38	25.31%	93.64
9	40.06%	0.09	30.02%	1.53	28.24%	96.06
10	37.47%	0.09	30.31%	1.47	27.80%	100.21

Table 3.6: MOVE/SWAP Ratio

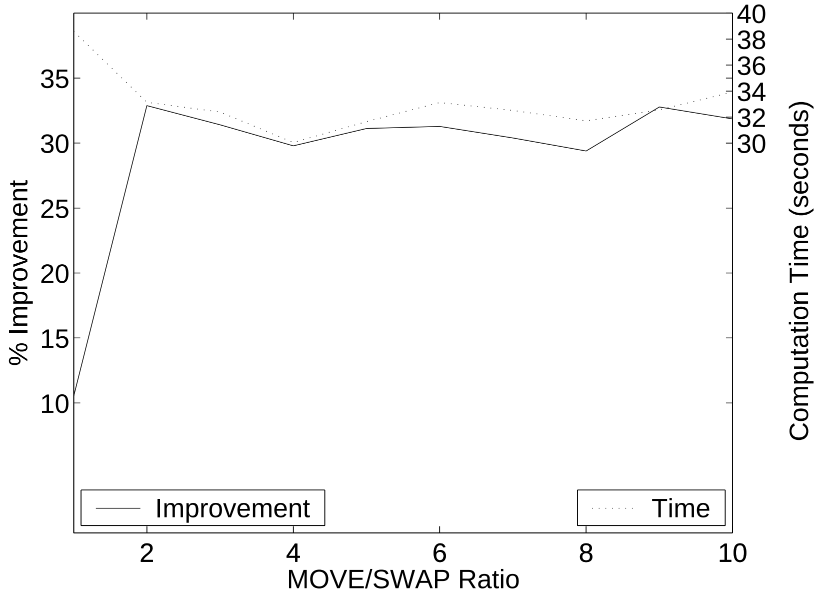


Figure 3.12: MOVE/SWAP Ratio: Circuit *struct*

benchmarks. The normalized improvement and computation time of the circuits versus the MOVE/SWAP ratio is plotted in Figure 3.12, which indicates that ratios ranging from 2 to 9 obtained adequate results in less CPU time. Empirically, the current implementation uses 5 as the default MOVE/SWAP ratio.

3.1.3.4 Size of Neighbourhood

Neighbourhood size is a crucial parameter to Local Search. Results in Table 3.7 show the effect of neighbourhood size on search quality and CPU time. Since a candidate solution is created either by moving a cell to a random position or by exchanging two random cells, the number of potential neighbours of a current solution may be huge for a medium/large circuit. Thus, usually only a portion of the potential neighbours are examined. In the experiments, the size ranges from 20 to 2000. From Table 3.7 we can conclude that search quality of Local Search

Circuit		Initial	Neighbourhood Size								
			20	40	60	80	100	200	500	1000	2000
fract	Interchange	86133	38	84	122	112	173	179	294	345	448
	W.L.		71947	61490	56805	55194	53402	49122	45030	40043	37632
	CPU Time		0.04	0.06	0.08	0.08	0.10	0.16	0.50	1.13	2.86
struct	Interchange	2.47	17	83	198	372	468	713	1233	1761	2290
	W.L.($\times 10^6$)		2.42	2.34	2.16	1.94	1.83	1.46	1.17	1.00	0.83
	CPU Time		0.80	0.82	0.90	1.01	1.12	1.77	5.04	12.87	32.15
ind2	Interchange	9.32	16	114	301	1009	1171	3306	6791	9721	12609
	W.L.($\times 10^7$)		9.26	9.13	8.88	7.92	7.39	6.93	5.61	5.03	4.59
	CPU Time		59.06	60.77	61.60	64.09	66.39	92.70	221.42	508.83	1280.21

Note:
Interchange: the number of moves attempted;
W.L.: total wire length.

Table 3.7: Size of Neighbourhood

highly depends on neighbourhood size. When a larger size is used, better search results are obtained. Figure 3.13 plots the experimental results for circuits *struct* and *ind2*.

In the previous experiments (see Tables 3.2, 3.4 and 3.5), neighbours size was set proportional to the circuit size, i.e $N_{cells}/20$ or $N_{cells}/30$. However, results indicate that these sizes are too small for small circuits and excessive for large circuits

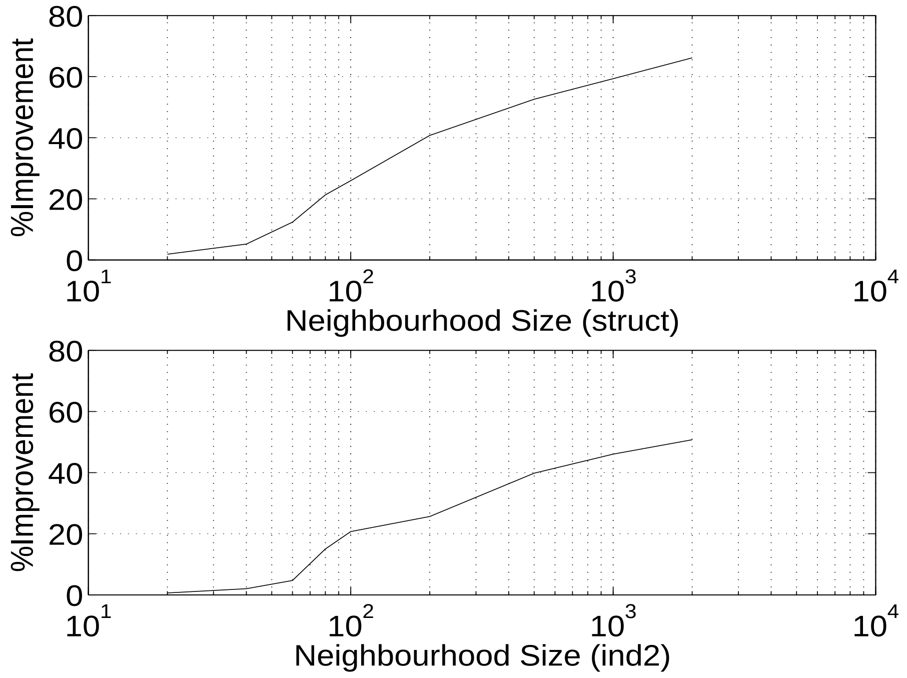


Figure 3.13: Improvement *vs* Neighbourhood Size

in terms of search quality and CPU time. Empirically, we found that a size of $50 \log_{10} N_{cells}$ produced similar quality of solutions with affordable CPU time for most circuits. Therefore, $50 \log_{10} N_{cells}$ is used as the default neighbourhood size in current Local Search implementation (results with this setup can be found in Table 3.10).

3.1.3.5 Overlap Penalty and Weight W_2

In Section 3.1.2, Equation (3.3) and (3.4) were considered for the second term C_2 (overlap penalty) of the cost function used by Strategy-II. In this section, experiments were conducted in an attempt to find out: (i) which equation is the most suitable for C_2 and (ii) how weight W_2 affects search quality. Results are displayed in Table 3.8 and 3.9.

Circuit	$W_2 \sum_{i \neq j} [O_{i,j}]$		$W_2 \sum_{i \neq j} [O^2_{i,j}]$	
	Improvement	Time	Improvement	Time
fract	34.78%	0.09	36.97%	0.12
struct	32.16%	1.56	23.59%	1.25
ind2	26.41%	93.31	13.40%	71.37

Table 3.8: Two Overlap Penalty Functions

The overlap penalty based on Equation (3.3) tends to penalize on the size of overlaps linearly while the quadratic term in Equation (3.4) applies more penalty on large overlaps than small ones. Results obtained in Table 3.8 indicate that the first function yields better improvement and therefore Equation (3.3) is chosen to represent overlap penalty.

	struct			ind2		
W_2	Interchange	%Imp	Time	Interchange	%Imp	Time
1	634	30.98%	1.52	3101	23.64%	102.6
2	643	30.90%	1.53	3212	24.17%	95.57
3	618	30.38%	1.50	3254	25.1%	90.24
4	595	29.92%	1.47	3268	26.2%	89.21
5	575	29.11%	1.45	3244	25.40%	88.84
6	568	29.12%	1.45	3126	24.31%	83.69
7	631	31.07%	1.53	3098	22.63%	80.54
8	573	29.01%	1.45	3052	23.75%	81.23
9	549	28.90%	1.43	3012	23.88%	79.96
10	528	27.97%	1.39	2886	20.46%	75.85
25	519	28.16%	1.38	2537	18.14%	70.63
50	345	21.02%	1.18	1830	13.99%	62.74
100	232	14.13%	1.0	986	10.52%	55.63

Table 3.9: Overlap Penalty Weight W_2

Table 3.9 presents results obtained for the LS based on different W_2 values.

A weight value in the range 1 to 9 yields acceptable solution quality while a large value tends to deteriorate solution quality. For circuit *struct*, a weight of 7 obtained 31.07% improvement and increasing the weight to 100 yielded 14.13% improvement.

3.1.3.6 Overall Performance of Local Search

Table 3.10 summarizes the experimental results for the implemented Local Search algorithm. The default neighbourhood size ($50 \log_{10} N_{cells}$) was used. An average improvement of 27.68% was obtained based on Strategy-II implementation.

Circuit	Wire length		Improvement	Interchanges	Time
	Initial	Final			
fract	86133	60250	30.05%	117	0.09
prim1	2.57×10^6	2.00×10^6	22.31%	293	0.37
struct	2.47×10^6	1.72×10^6	30.18%	602	1.45
ind1	6.17×10^6	4.08×10^6	33.90%	1160	4.17
prim2	1.66×10^7	1.27×10^7	23.60%	1196	2.79
bio	1.31×10^7	0.96×10^7	26.65%	1526	18.52
ind2	9.32×10^7	6.95×10^7	25.40%	3244	88.84
ind3	2.82×10^8	2.17×10^8	23.04%	4768	42.46
avq.s	8.84×10^7	6.28×10^7	28.96%	5814	797.78
avq.l	1.01×10^8	0.73×10^8	27.71%	5575	724.12
Average Improvement			27.68%		
Note:	The neighbourhood size is defined as $50 \log_{10} N_{cells}$.				

Table 3.10: Overall Performance of Local Search

3.2 A Tabu Search Implementation

The Tabu Search [Glov93] meta-heuristic is an effective and efficient technique that have been successfully applied to many applications. This section presents our implementation of Tabu Search method to the standard cell placement problem. Results obtained indicate that for most benchmarks, Tabu Search produces better solution quality than hill-climbing Local Search algorithms.

3.2.1 Algorithm Outline

An outline of the Tabu Search implementation is illustrated in Figure 3.14. For standard cell placement, the cost function of the Tabu Search algorithm is based on Equation 3.1. The input of Tabu Search is an initial placement solution. This solution can be either randomly generated or constructed based on a certain constructive technique. Based on a neighbour searching method similar to that described previously for LS, the TS algorithm accepts both improving and deteriorating moves as long as they are the best candidates in any iteration. Moreover, it incorporates a short-term memory of moves obtained from past history to guide the search such that it avoids revisiting solutions recently explored. Aspiration is an additional feature added to further explore the solution space. Two different criteria can be used to terminate the search: (i) when a specified maximum number of iterations are reached, (ii) when the search performs a specified maximum number of consecutive non-improved moves. Since Tabu Search accepts moves that tend to deteriorate the cost function, a current solution does not necessarily have to be an improving move. Thus the best-so-far solution is always updated whenever a better one is found. At the end of Tabu Search, a legalization procedure is required to convert the best-so-far solution into a valid placement solution.

In the next section, several implementation issues are discussed, including the neighbourhood definition, the tabu memory, aspiration, the stopping criteria and

```
                        Tabu Search Algorithm
Input:
The net list of the circuit

Initialization:
Choose an initial placement solution
Accumulated_Imprv := 0
Initialize Tabulist
Initialize Recoverlist

Mainloop:
while (the stopping criteria is not met)
    BestImprovement := $-\infty$
    for( i = 0; i < NeighbourhoodSize; i++ )
        CurrentNeighbour := Generate_an_Interchange()
        improvement := Evaluate_the_Interchange()
        if CurrentNeighbour is tabued then
            if ((improvement + Accumulated_Imprv) $\leq 0$ ) then
                Comment: CurrentNeighbour does not meet Asp criteria,
                                this move is discarded.
                continue
            else Override the tabu status
        if ( improvement > BestImprovement ) then
            BestImprovement := improvement
            BestNeighbour := CurrentNeighbour
    end for
    Accumulated_Imprv := Accumulated_Imprv + BestImprovement
    if (Accumulated_Imprv $\geq 0$) then
        Comment: It is the best solution so far
        Flush Recoverlist
        Accumulated_Imprv := 0
    else
        Add the move to Recoverlist
    Perform the interchange of BestNeighbour
    Update Tabulist
    CurrentSolution := BestNeighbour
end while
if (Accumulated_Imprv $< 0$) then
    Comment: CurrentSolution is not the best solution so far
    Roll back to best_so_far solution from CurrentSolution with Recoverlist
Legalize_Circuit()

Output:
CurrentSolution
```

Figure 3.14: Tabu Search Algorithm for Standard Cell Placement

the roll back function.

3.2.2 Definition of Neighbourhood

The neighbourhood $N(s)$ of a solution s is defined as the set of solutions s' that can be obtained from s by either randomly displacing a cell to a new location or randomly exchanging the locations of any two cells. Compared to the neighbourhood definitions in [Song92] and [Sait01], this implementation tends to search within a much larger neighbourhood space. In [Song92], a neighbour was created only by exchanging two connected cells, and [Sait01] allowed exchanging non-adjacent cells however “cell displacement” method was not used. Since swapping two adjacent cells will not change the half-perimeter of the bounding box of the net that connects these two cells, a cell exchange should not be restricted to the pairs of connected cells. In addition, with “cell displacement, it is more efficient for a cell to move to a desired location since overlaps are allowed in Strategy-II.

In the current implementation, users can determine the neighbourhood size in run-time, otherwise a default value that depends on circuit size is used.

3.2.3 Tabu Memory

Tabu memory is a crucial component of the Tabu Search approach. Simply accepting bad moves cannot prevent the search from cycling. A tabu list is therefore used to record some recently visited solutions and prevent the search from revisiting them. In any iteration k, the algorithm generates a set of neighbourhood moves $N(s)$ by randomly choosing a cell with a new location (cell displacement) or a pair of cells (cell exchange). Instead of evaluating all solutions of $N(s)$, the procedure evaluates only the candidates of $N(s,k) = N(s) - T(k)$, where $T(k)$ is the set of solutions in the tabu list. Each time a new move is generated, the solution will be looked up in the tabu list; if a move is tabu (and does not meet the aspiration criteria), this placement configuration is rejected.

A successful implementation of Tabu Search involves an appropriate definition of tabu memory. A simple tabu list in which each tabu element records all modules of a solution may work well for small circuits. However, for a large circuit, the operation of such a tabu list may be prohibitive. Therefore, many applications of Tabu Search in the literature store attributes of solutions. The current implementation uses *from-attributes* of solutions in the tabu memory. To identify a practical and effective transforming attribute, the following definitions of a tabu move are considered:

1. Each tabu element records the cell number, the row number and the cell location of each affected cell of a tabu move. Using this attribute will hardly forbid any moves since the location of a cell is represented by a floating point number and the possibility of randomly generating two identical locations for a cell is small.

2. Each entry in the tabu list records the cell number, the row number and the index of each affected cell. An index means the position order of a cell in the row. This method requires extra computational effort to keep updating the index of each module, which could be expensive.

3. Each tabu element records the cell number and the row number of each affected cell. This type of tabu memory is employed in the implementation

since it is the most restrictive.

Tabu list management is concerned with the determination of size of the tabu list and means to update it. The recency based tabu memory avoids cycles of length less than or equal to the size of tabu list. If the size is too small, cycling may occur frequently and if the size is too large, it may forbid too many moves and restrict search exploration. A large tabu list size may also require more time to search. In our implementation, empirically large circuits should have a larger size of tabu list since they usually have a longer search trajectory and may have to prevent a longer cycle. In the implementation, the default size of the tabu list is a function of the number of modules in the circuit: $|T| = MAX(5, \alpha \times \log_{10}(modules))$. Value of α ranges from 2 to 7. This formula produces acceptable results for the tested benchmarks. The tabu size can also be set dynamically during the search as follows: (i) Increase the size by 1 when a repetition of a solution occurs since the last change; (ii) Decrease the size by 1 when a cycling is detected and its length is greater than the moving average of cycling length[1]. This strategy is implemented according to Glover's observation: A small tabu size is preferable for exploring and a larger tabu size is preferable for breaking free from the vicinity of a local optima [Glov93].

Tabu list is a first-in-first-out queue of length $|T|$, where elements form a circular queue. After a new solution is inserted into the end of the list, the oldest tabu element entry is dropped. Figure 3.15(a) and (b) illustrate the tabu list before and after this change. The arrangement of tabu memory as a circular queue makes the changing of tabu size very easy.

[1]Whenever a cycling is detected, the moving average of cycling length is updated by: $moving\_average = 0.1 \times this\_cycle\_length + 0.9 \times moving\_average_{last}$

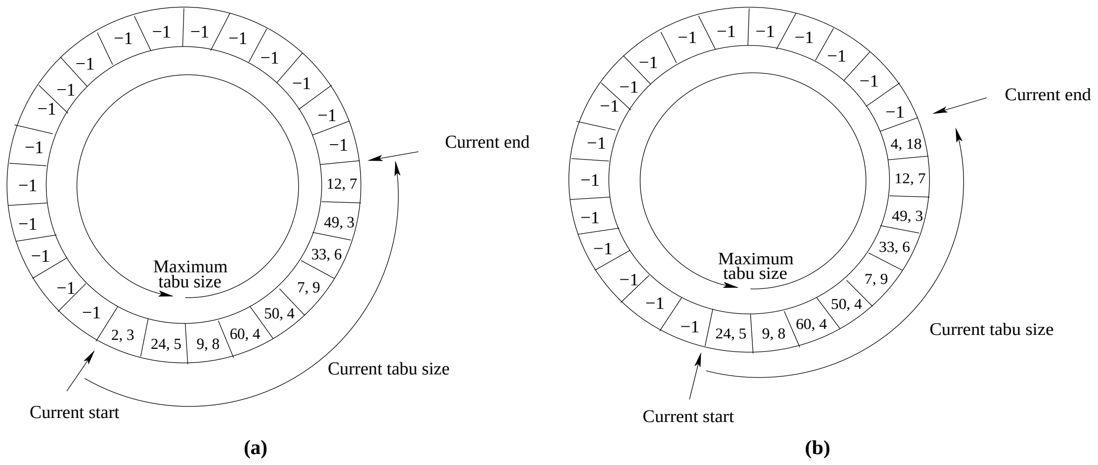


Figure 3.15: The Circular Queue Structure of Tabu List

3.2.4 Aspiration

Aspiration is a mechanism within Tabu Search that temporarily overrides the tabu status of moves to give more flexibility to search the solution space.

Aspiration criteria defines when the algorithm should release a solution from its tabu status. One possible criteria that can be used is: when a move leads to a new solution which is better than the best found so far, the tabu status of this solution is dropped. Each time a new solution is generated, it will be looked up in the tabu list; if the move is tabu, it will be compared with the current aspiration level which is the cost value of the best found solution so far. If the tabu move is better than the aspiration level, its tabu status is dropped or else it is rejected. In cooperation with short term memory, aspiration plays a crucial role in achieving good solutions.

3.2.5 Stopping Criteria

In this implementation, users can specify one of the following two criteria to terminate the search:

1. The search stops when the total number of iterations performed is greater than a specified maximum number of iterations.

2. The search stops when the number of iterations performed since the last best solution exceeds a specified maximum number of iterations.

Maximum	Circuit struct		
iterations	Interchanges	Improvement	Time
1000	1000	38.62%	2.03
2000	2000	49.12%	3.28
3000	3000	54.20%	4.54
4000	4000	56.41%	5.85
5000	5000	58.85%	7.05
10000	10000	61.28%	13.52
15000	15000	62.77%	19.70
20000	20000	63.03%	25.98
30000	30000	63.75%	38.76

Table 3.11: Stopping Criteria: Maximum Iterations

Results in Table 3.11 and 3.12 indicate that the first criteria has more control of desired computation effort while the second criteria has more control of expected search quality.

3.2.6 Roll Back to the Best Solution

Since the algorithm accepts bad moves, the current solution does not necessarily represent the best solution obtained so far and therefore the *best-so-far* solution is

Maximum moves	Circuit struct		
	Interchanges	Improvement	Time
10	2651	52.72%	4.14
20	3636	56.31%	5.34
40	4887	58.16%	6.97
60	6663	60.50%	3.17
80	7112	60.47%	9.74
100	8493	60.88%	11.54
200	16058	63.06%	21.46

Table 3.12: Stopping Criteria: Maximum Consecutive Non-improved Moves

legalized as a final output. In the implementation, a data structure *Recover_List* is used to record the moves since the last change of *best-so-far*. At the end of the iterative improvement procedure, the best solution is reversely recovered from the current solution according to the information in the *Recover_List*.

3.2.7 Experimental Results

3.2.7.1 Overall Performance of Tabu Search

Table 3.13 summarizes the default parameters of the Tabu Search implementation. The neighbourhood size is set to a similar value used by Local Search.

With using default parameters, a set of experiments are conducted with various benchmarks (see Table 3.14). It can be concluded from the results that Tabu Search can produce better solution quality than Local Search.

Circuit	Default Parameters		
	Neighbourhood size ($50\log_{10} N_{cells}$)	Tabu size ($7\log_{10} N_{cells}$)	Stopping criteria $*$ ($10\log_{10} N_{cells}$)
fract	104	14	20
prim1	143	20	28
struct	163	22	32
ind1	167	23	33
prim2	173	24	34
bio	190	26	38
ind2	204	28	40
ind3	208	29	41
avq.s	216	30	43
avq.l	219	30	43
$*$ Stopping criteria: Maximum consecutive non-improved moves.			

Table 3.13: Default Parameters Used by Tabu Search

Circuit	Wire length		Improvement	Interchanges	CPU Time
	Initial	Final			
fract	86133	43368	49.65%	1841	0.66
prim1	2.57×10^6	1.46×10^6	43.27%	3425	2.98
struct	2.47×10^6	1.05×10^6	57.50%	4497	5.90
ind1	6.17×10^6	2.83×10^6	54.10%	9342	30.51
prim2	1.66×10^7	0.83×10^7	49.47%	18391	35.44
bio	1.31×10^7	0.58×10^7	55.32%	14095	158.54
ind2	9.32×10^7	4.59×10^7	50.73%	33220	372.97
ind3	2.82×10^8	1.50×10^8	46.92%	36843	239.08
avq.s	8.84×10^7	3.65×10^7	58.75%	48270	6191.17
avq.l	1.01×10^8	4.22×10^8	58.31%	50403	5660.50
Average Improvement			52.40%		

Table 3.14: Performance of the Tabu Search Technique

3.3 A Simulated Annealing Implementation

Simulated Annealing (SA) is a powerful technique that has been successfully applied to solve many large combinatorial problems. This section reports our implementation of Simulated Annealing algorithm to the standard cell placement problem.

3.3.1 Algorithm Outline

In the current implementation, Simulated Annealing employs the same cost function represented by Equation (3.1), except that the weights are dynamically altered according to the change in temperature. Again, a legalization procedure at the end of the algorithm is required.

The pseudo-cede of the implemented algorithm is given in Figure 3.16. The variable "T" is an important control parameter used by the algorithm. Initially, annealing starts with a high initial temperature "T_0". As the algorithm proceeds, the temperature decreases according to the cooling schedule. At each temperature, a number of neighbourhood solutions are generated and evaluated. Moves are always accepted if they reduce the cost of the objective function. However, when a move tends to deteriorate the cost then it is accepted with a probability. When the control parameter "T" is high, deteriorating moves are accepted with high probability, but as temperature cools down the probability decreases and these moves are rejected. The algorithm is terminated when the temperature reaches the final temperature "T_0". After all iterations are completed, the best solution is reversely recovered from the last move according to the data structure *RecoverList*.

3.3.2 Annealing Schedule

Practical Simulated Annealing implementations tend to use heuristic based schedules rather than the original formulation [Aart90] which leads to global convergence.

```
Simulated Annealing Algorithm
Input:
The net list of the circuit

Initialization:
Choose an initial placement solution $s := s_0$
Find $T_0$ according to $Initial\_Acceptance\_Probability$
$T := T_0$
Initialize RecoverList

Mainloop:
while (the stopping criteria is not met)
    while (not yet in equilibrium)
        $s' := Generate\_an\_Interchange()$
        $improvement := Evaluate\_the\_Interchange()$
        if $improvement \geq 0$ then
            Perform the interchange $s := s'$
        else
            if $(e^{\frac{improvement}{T}} \geq random(0,1))$ then
                Perform the interchange $s := s'$
            else
                continue
        $Accumulated\_Imprv := Accumulated\_Imprv + improvment$
        if (Accumulated_Imprv $\geq$ 0) then
            Flush RecoverList
            $Accumulated\_Imprv := 0$
        else
            Add the move to RecoverList
    end while
    Update weights
    Update temperature T
    Update NonImpv_Counter
end while
if (Accumulated_Imprv $<$ 0) then
    Roll back to $s*$ from $s$ with RecoverList
Legalize_Circuit()

Output:
CurrentSolution
```

Figure 3.16: Simulated Annealing Algorithm for Standard Cell Placement

The annealing schedule is determined by four factors:

1. The **initial temperature** T_0. Theoretically, the initial value of T_0 is set so that virtually all transitions are accepted. In the implementation, users can empirically specify T_0 to be set to very large value of T_0 such that $e^{\frac{-\Delta cost}{T_0}} \approx 1$. However, such an initial temperature may be difficult to determine. A low T_0 may cause the initial acceptance probability $e^{\frac{-\Delta C}{T}}$ to be much less than 1 and might affect the final placement solution. With an extremely high T_0, the search may accept too many random moves at the beginning, wasting CPU time. Alternatively, users can determine T_0 by specifying the parameter of initial acceptance probability Pc_{init}. The algorithm performs a number of random walks from s_0 at the initialization stage and calculates the average ΔC for these moves. Then the initial high temperature can be determined with the formula $e^{\frac{-\Delta C_{avg}}{T_0}} \approx Pc\_{init}$.

2. **The size of inner-loop** is the equilibrium detection condition. At each temperature, enough moves should be attempted. In the implementation, the inner-loop size is set according to the size of the circuit. The default value used in this implementation is $50 \log_{10} N_{cells}$.

3. **The temperature decrement rate** (α) is usually fixed and set to a value close to 1. Typical values lie between 0.8 to 0.99 in any implementation.

4. **The termination conditions**. Annealing stops either when the temperature reaches a very low T_{final}, or the cost remains unchanged for the last n consecutive temperatures.

3.3.3 Dynamically Changing Weights

The cost Equation (3.1) can be re-written as:

$$C = (W_H \times X + W_V \times Y) + (W_2 \times OL) + (W_3 \times RL) + (W_4 \times RM) \qquad (3.8)$$

where X and Y denote the total wire length of all nets in the x and y direction; OL represents the total overlap of all cells; RL denotes the sum of undershoot/overshoot length in all rows; and RM is the total overcrowded penalty in all rows. Since the legalization function at the final stage may increase the X but the Y will not be affected, usually we set the value of W_H to 1 and set a slightly larger value for W_V to give higher priority to the wire length in the vertical direction. A fixed value of W_2 specifies the weight of modules overlap penalty. Weights W_3 and W_4 decide how much penalty is added to the layouts with the illegal row length and overcrowded row, respectively.

The weights W_2, W_3 and W_4 are usually small at the beginning, giving more flexibility to the search. As the temperature cools down the weights increase slightly to apply more penalty to illegal moves in an attempt to produce a legal final solution.

3.3.4 The Spanning Windows

Usually, the cells and the destinations are randomly selected. However, a spanning window is also implemented and can be used to restrict the moving scope of the cells. This attempts to imitate the annealing process of solids: when the temperature is high in the liquid phase, particles of the solid have more energy and arrange themselves in a large scope; as temperature decreases to the freezing point, the energy of the system is minimal and the particles arrange themselves in the ground state of the solid. Large-distance moves of a cell usually yield large values of $|\Delta C|$, whether better or worse. Therefore, these moves should be rejected at low temperatures. This is achieved by applying a spanning window to the *Generate_an_Interchange* function to limit a cell's moving distance.

3.3.5 Experimental Results

3.3.5.1 Effect of the Parameters on Solution Quality and CPU Time

The temperature decrement rate (α) determines the next temperature to be used by the algorithm. Table 3.15 and 3.16 display experiments conducted with α ranging from 0.5 to 0.99. The stopping criteria was based on $T_{final} = 0.1$. Results indicate

	struct			
α	Interchange	Final wire length	Improvement	Time
0.50	150	2.41×10^6	2.49%	0.82
0.70	288	2.36×10^6	4.62%	0.83
0.90	812	2.15×10^6	12.97%	0.87
0.95	1276	1.98×10^6	19.91%	0.90
0.96	1686	1.89×10^6	23.26%	0.97
0.97	2120	1.85×10^6	25.13%	1.02
0.98	2391	1.70×10^6	31.11%	1.12
0.99	4408	1.47×10^6	40.27%	1.40
Initial wire length: 2.47×10^6				
$*$ Parameters:				
Probability of initial move: 0.95				
Final temperature: 0.1				

Table 3.15: Effect of the Temperature Decrement Rate α: *struct*

that a small temperature decrement rate (e.g. less than 0.9) yielded poor placement

α	ind2			
	Interchange	Final wire length	Improvement	Time
0.50	391	8.87×10^7	1.5%	38.0
0.70	910	8.73×10^7	3.1%	41.9
0.90	2209	8.38×10^7	6.9%	42.5
0.95	4246	7.85×10^7	12.8%	43.6
0.96	5125	7.67×10^7	14.8%	44.7
0.97	6408	7.33×10^7	18.5%	45.7
0.98	7207	7.29×10^7	19.0%	46.7
0.99	14088	6.31×10^7	29.9%	56.5
Initial wire length: 9.32×10^7 $*$ Parameters: Probability of initial move: 0.95 Final temperature: 0.1				

Table 3.16: Effect of the Temperature Decrement Rate α: *ind2*

solutions. This is attributed to the rapid temperature cooling and therefore a small number of iterations being executed. The values of α ranging from 0.98 to 0.99 are empirically appropriate for the implementation.

The initial temperature T_0 of the algorithm should be large enough to allow virtually all new configurations to be accepted. T_0 can be determined by a user. Alternately, a parameter of initial acceptance probability Pc_{init} can be specified to determine T_0. Table 3.17 and 3.18 displays the results of a set of experiments where Pc_{init} ranged from 0.55 to 0.99. For the circuit *struct*, $Pc\_{init} = 0.99$ yielded the best solution among the five Pc_{init} values however its CPU time is the longest. On the other hand, when $Pc\_{init} = 0.90$, the algorithm produced a result close to the best one but in a much smaller CPU time. The experiment with circuit *ind2* displays similar results. In the current implementation, the initial acceptance probability Pc_{init} is 0.90 by default.

$Pc_{\_init}$	struct				
	T_0	Interchange	Wire length	Improvement	Time
0.50	6932363	1651	1.67×10^6	21.3%	1.1
0.70	8663310	1941	1.62×10^6	25.0%	1.1
0.90	8329563	2083	1.59×10^6	26.1%	1.2
0.95	9187583	1973	1.61×10^6	25.6%	1.2
0.99	3292007	4278	1.55×10^6	27.9%	2.8
Initial wire length: 2.47×10^6					
$*$ Parameters:					
Alpha: 0.98					
Final temperature: 0.1					

Table 3.17: Effect of Initial Acceptance Probability $Pc_{\_init}$: *struct*

$Pc_{\_init}$	ind2				
	T_0	Interchange	Wire length	Improvement	Time
0.50	78851316	3548	7.97×10^7	14.5%	43.2
0.70	79181012	3774	7.92×10^7	15.1%	43.4
0.90	83955901	5214	7.71×10^7	17.3%	44.3
0.95	81956050	4610	7.76×10^7	16.8%	44.2
0.99	86070492	7454	7.64×10^7	18.2%	42.1
Initial wire length: 9.32×10^7					
$*$ Parameters:					
Alpha: 0.98					
Final temperature: 0.1					

Table 3.18: Effect of Initial Acceptance Probability $Pc_{\_init}$: *ind2*

Originally with homogeneous Simulated Annealing algorithms, an infinite number of transitions is generated for each temperature. In practice, this value should be set as a function of the number of modules in the circuit. Table 3.19 shows results obtained using inner-loop sizes ranging from 60 to 2000. Results indicate that a small inner-loop size yields poor improvement since the search was prevented to reach its equilibrium at each temperature (i.e *meta-stable* structures). With a

Inner-loop size	Interchange	Final wire length	Improvement	Time
60	4047	1.56×10^6	36.73%	1.29
80	3422	1.41×10^6	43.00%	1.51
100	3866	1.33×10^6	46.22%	1.71
120	4356	1.29×10^6	47.93%	1.84
150	4611	1.21×10^6	50.82%	2.15
200	5317	1.15×10^6	53.28%	2.57
300	6610	0.98×10^6	59.98%	4.50
400	7826	0.93×10^6	62.32%	5.98
500	8659	0.89×10^6	63.85%	7.89
600	9409	0.85×10^6	65.55%	10.65
800	11128	0.79×10^6	67.83%	15.97
1000	13459	0.76×10^6	68.94%	26.61
2000	35858	0.66×10^6	73.14%	165.39
Initial wire length: 2.47×10^6				
$*$ Parameters:				
Alpha: 0.99				
Probability of initial move: 0.90				
Stopping criteria: consecutive non-improved temperatures is 3.				

Table 3.19: Effect of the Inner-loop Size

lager size, the SA produced better results but the computation time grew exponentially. For circuit *struct*, a size of 1000 produced results 1% better than those of the size of 800, but it required a 67% longer computation time. In the current SA implementation the inner-loop size is set to $50 \log_{10} N_{cells}$ (further results can be found in Table 3.22 in Section 3.3.5.3).

3.3.5.2 Stopping Criteria

Two stopping criteria can be used to terminate the search: (i) the final temperature T_{final} set by the user, and (ii) the maximum number of consecutive non-improved temperatures. Table 3.20 and 3.21 display the results of the experiments conducted using different stopping criteria.

Final Temperature	Interchange	Improvement	CPU Time
0.50	3489	42.60%	1.44
0.25	3450	43.79%	1.48
0.10	3997	45.17%	1.67
0.050	4221	46.15%	1.74
0.025	4213	47.03%	1.78
0.010	4595	49.68%	1.99
0.005	5092	51.12%	2.16
0.001	4850	51.31%	2.22
0.0001	5217	53.78%	2.56

Table 3.20: Stopping Criteria: Final Temperature

Non-improved Temperatures	Interchange	Improvement	CPU Time
1	4006	46.99%	1.62
2	4548	51.81%	2.02
3	5403	55.31%	2.48
4	6088	58.10%	3.13
5	6396	60.01%	3.79
6	6661	60.61%	4.04
7	6959	61.37%	4.55
8	7550	64.21%	5.39
9	7604	65.97%	6.03
10	8018	66.53%	6.81

Table 3.21: Stopping Criteria: Non-improved Temperatures

3.3.5.3 Overall Performance of Simulated Annealing

Table 3.22 displays the placement results based on the Simulated Annealing algorithm. Various benchmarks were tested and a good quality of solutions were

Circuit	Wire length		Improvement	Interchange	Time
	Initial	Final			
fract	86133	36159	58.02%	5207	1.23
prim1	2.57×10^6	1.24×10^6	52.02%	15388	11.04
struct	2.47×10^6	0.77×10^6	68.71%	10436	13.07
ind1	6.17×10^6	2.33×10^6	62.28%	22170	71.64
prim2	1.66×10^7	0.66×10^7	60.01%	49500	111.06
bio	1.31×10^7	0.46×10^7	64.34%	29315	301.31
ind2	9.32×10^7	3.96×10^7	57.56%	70277	635.50
ind3	2.82×10^8	1.22×10^8	56.72%	77883	393.90
avq.s	8.84×10^7	3.09×10^7	65.02%	86056	6204.68
avq.l	1.01×10^8	0.35×10^8	64.46%	91572	5708.38
Alpha: 0.99 Pc_{init}: 0.90 Inner-loop size: $50 \log_{10} N_{cells}$ Stopping criteria: consecutive non-improved temperatures is 6.					

Table 3.22: Overall Performance of Simulated Annealing

obtained as expected. The results were based on the following annealing schedule: T_0 was determined by Pc_{init} where Pc_{init} is set to 0.90, α was set to 0.99, and the stopping criteria was six non-improved consecutive temperatures.

3.4 A Genetic Algorithm Implementation

Figure 3.17 displays the flowchart of a steady-state Genetic Algorithm implementation carried out by [Yang03] previously. The algorithm first chooses an initial population, then initiates evolution by applying stochastic operators such as selection, crossover and mutation to the individuals iteratively. In the implementation, only two individuals are randomly selected to perform mating in each generation and create two children in an attempt to replace two worse individuals. Unlike

```
GA for Placement
1. set popsize, max_gen;
   crossover_rate, mutation_rate, selection method;
2. Choose initial population randomly
3. **While** Not Done
     **For** (i=1 to popsize/2)
        Select_parents(mate1,mate2);
        if (random(0,1) $\leq$ crossover_rate)
           child = Do_Crossover(mate1,mate2);
        if (random(0,1) $\leq$ mutation_rate)
           Mutation(offspring) and evaluate offspring;
     **End For**
        Replacement();
        gen = gen + 1 ;
   **End While**
4. Return best placement in current population.
```

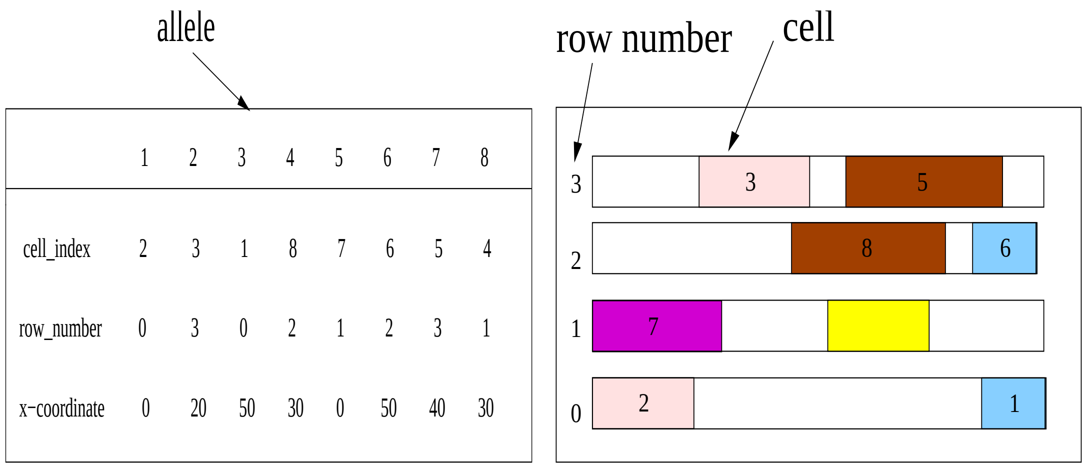
Figure 3.17: A Genetic Placement Algorithm [Yang03]

generational GAs where the entire population is replaced, steady-state GAs are population overlapped.

Scoring Function

Each individual (string) in the population encodes a feasible solution to the standard-cell placement problem. This string is represented by a set of alleles, as shown in Figure 3.18. The number of alleles is equal to the number of cells. Each allele indicates the cell index, the X- coordinates, and the row number of the cell. Figure 3.18(a) illustrates the string encoding of the standard-cell placement given in Figure 3.18(b). The fitness value of an individual is evaluated by a fitness function F that estimates the total half-perimeter wire length:

$$F = \frac{1}{\sum_{i=1}^{n} HPWL_i} \tag{3.9}$$



	1	2	3	4	5	6	7	8
cell_index	2	3	1	8	7	6	5	4
row_number	0	3	0	2	1	2	3	1
x-coordinate	0	20	50	30	0	50	40	30

(a) String Encoding (b) Placement

Figure 3.18: String Encoding

where $HPWL_i$ is the estimated wire-length of net i, and n is the number of nets. Cell overlaps are removed and row lengths are adjusted before evaluating the chromosome.

Selection Function

The selection function employs a Binary Tournament method. To select an individual as a candidate parent, two individuals are chosen randomly and the one with higher fitness value is picked for mating.

Crossover Operator

Once two chromosomes are selected, a crossover operator is used to generate two offspring. [Yang03] employed a crossover operator called Order crossover in the implementation. Figure 3.19(a) shows a one-point order crossover operator where each pair of parents generates two children with a probability equal to the crossover rate. The operator first copies the array segment to the left point from one parent to one offspring. Then it fills the remaining part of the offspring by going through the other parent, from the beginning to the end and taking those elements that were left out, in order. The default crossover operator of the implementation is a two-points order crossover, which is similar to one-point order crossover, as Figure 3.19(b) illustrates.

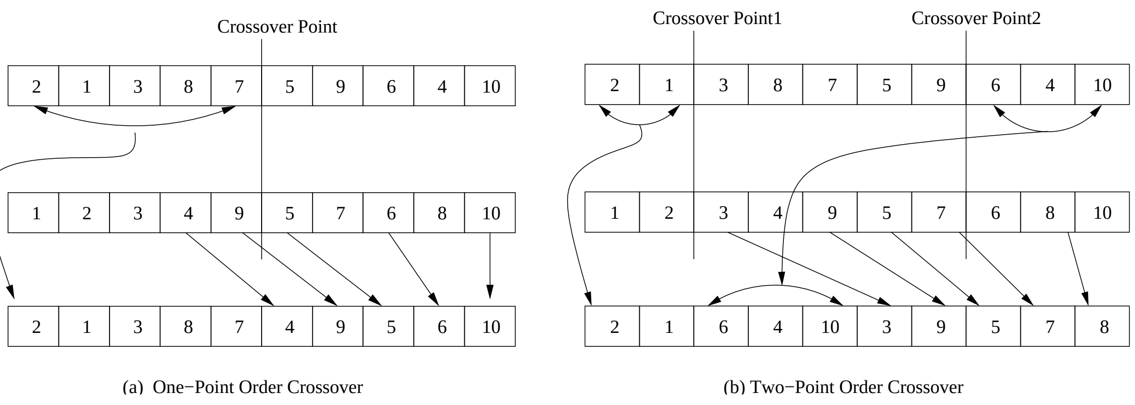


Figure 3.19: One-Point and two-Point Order Crossover

Mutation Operator

Following crossover, each offspring is mutated with a probability equal to the mutation rate. In the implementation, the mutation operator alters an individual by randomly selecting a pair of cells following an interchanging of their cell numbers, x-coordinates and row numbers. Figure 3.20 illustrates the mutation process. The random nature of mutation operation allows for a broader exploration of the

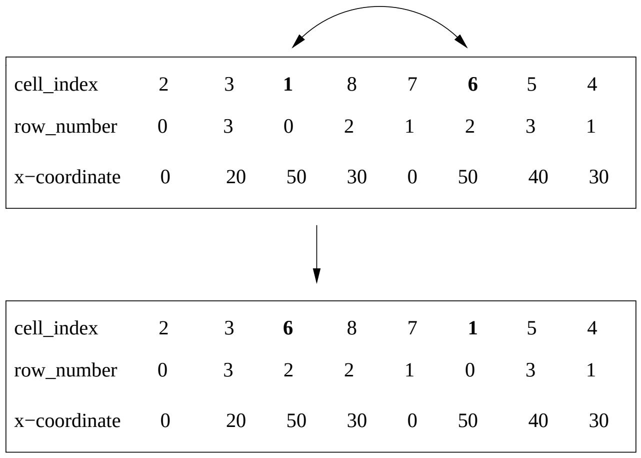

cell_index 2 3 1 8 7 6 5 4
row_number 0 3 0 2 1 2 3 1
x−coordinate 0 20 50 30 0 50 40 30
cell_index 2 3 6 8 7 1 5 4
row_number 0 3 2 2 1 0 3 1
x−coordinate 0 20 50 30 0 50 40 30

Figure 3.20: Mutation Operator

solution space.

Replacement Function

As the new offspring are mutated, replacement is then performed to form a next generation by using an elitism method. Two children are compared with the two worst parents and the two fittest individuals are kept to the next generation. The approach guarantees that the best individual in the current generation will appear in subsequent generations [Mitc96, Grew95], protecting the search from regression.

3.4.1 Experimental Results

Table 3.23 summarizes the experimental results of the Genetic Algorithm, where the generation size is set to 100, the population size is set to 24, and the initial solutions are injected with 30% of good solutions based on the Cluster-Seed method in [Yang03]. Results obtained indicate that the Genetic Algorithm produce good

	Wire length	Improvement	CPU Time
Fract	62338	27.63%	1.7
Prim1	1.81×10^6	29.75%	15.4
Struct	9.3×10^5	62.14%	55
Ind1	4.55×10^6	26.37%	86.3
Prim2	1.05×10^7	36.48%	130.8
Bio	7.2×10^7	45.05%	537.4
Ind2	6.41×10^7	31.23%	2223.5
Ind3	1.59×10^8	43.71%	3530.3
avq.s	3.89×10^7	56.01%	7535.5
avq.l	5.1×10^7	49.57%	9661.3
Average		40.79%	
∗ Parameters: Generations: 100; Population size: 24; Others: 30% Injection Initial Solution			

Table 3.23: Performance of Genetic Algorithm

quality solutions but is a notorious consumer of CPU time.

3.5 LS, TS, SA and GA: A Comparison

The performance of the Local Search algorithm was compared to (i) Tabu Search (ii) Simulated Annealing algorithm and (iii) Genetic Algorithm based on 20 runs. The tests were carried out on all MCNC benchmark circuits, using only the default run-time parameters. Experimental results are displayed in Table 3.24 and 3.25.

Table 3.24 compares the algorithms in terms of CPU time, and Figure 3.21 illustrates these performance results. Since TS and SA are based on the same local search algorithm, their searching activities (the number of moves attempted) are also plotted in Figure 3.22. Results show that the Local Search algorithm accepts

Circuit	**LS**	**TS**	**SA**	**GA**
fract	0.09	0.66	1.23	1.7
prim1	0.37	2.98	11.04	15
struct	1.45	5.90	13.07	55
ind1	4.17	30.51	71.64	86
prim2	2.79	35.44	111.06	130
bio	18.52	158.54	301.31	537
ind2	88.84	372.97	635.50	2223
ind3	42.46	239.08	393.90	3530
avq.s	797.78	6191.17	6204.56	7535
avq.l	724.12	5660.50	5708.38	9661

Table 3.24: Comparison of LS, TS, SA and GA (CPU Time)

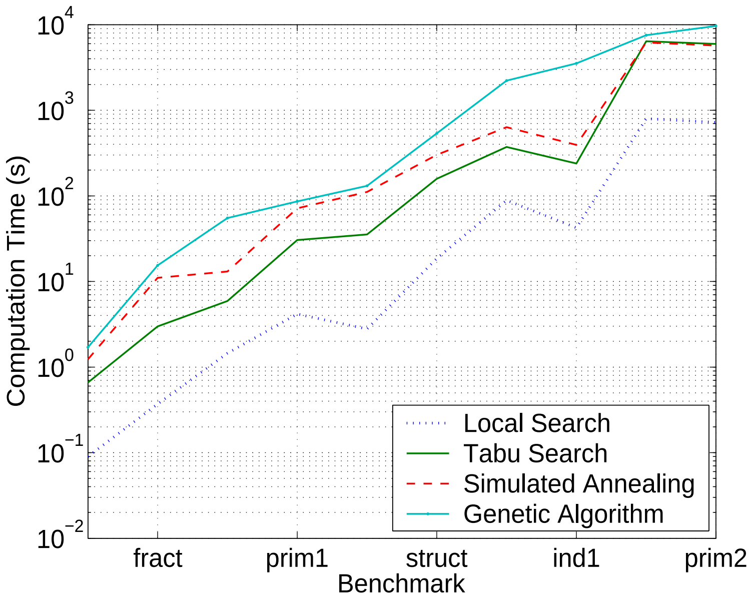


Figure 3.21: Comparison of LS, TS, SA and GA (CPU Time)

moves much less than those of the Tabu Search and Simulated Annealing, since the LS gets trapped in local minima while the meta-heuristics continue to search landscape. That is, advanced meta-heuristics required longer computation time

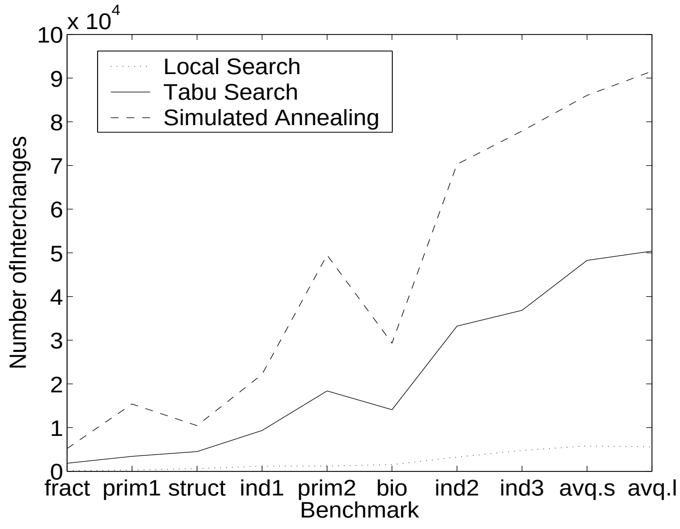


Figure 3.22: Comparison of LS, TS and SA (Move Attempted)

than a simple heuristic. The Tabu Search generally performs productive moves and therefore spends less time than the Simulated Annealing as seen in Figures 3.21 and 3.22. On the other hand, GA consumed the largest amount of CPU time to solve the problem in the experiments.

Table 3.25 summarizes the percentage of improvement obtained by these heuristic algorithms. It is clear from Table 3.25 that the improvement obtained by LS was inferior and the meta-heuristics yielded superior solution quality.

The Local Search algorithm is a basic heuristic which has neither intelligence nor knowledge to guide the search. It blindly searches for the best solution among the neighbours of a current solution and performs the moves until no further improvement can be obtained. The search always gets trapped in local minima, therefore results expected are poor. With a Local Search algorithm, the final solutions are dependent on the initial placements. In addition, a large neighbourhood size may

Circuit	**LS**	**TS**	**SA**	**GA**
fract	35.05%	49.65%	58.02%	27.63%
prim1	22.31%	43.27%	52.02%	29.75%
struct	30.18%	57.50%	68.71%	62.14%
ind1	33.90%	54.10%	62.28%	26.37%
prim2	23.60%	49.47%	60.01%	36.48%
bio	26.65%	55.32%	64.34%	45.05%
ind2	25.40%	50.73%	57.56%	31.23%
ind3	23.04%	46.92%	56.72%	43.71%
avq.s	28.96%	58.75%	65.02%	56.01%
avq.l	27.71%	58.31%	64.46%	49.57%
Average	27.68%	52.40%	60.91%	40.79%

Table 3.25: Comparison of LS, TS, SA and GA (%Improvement)

help to yield a better final result, but an exhaustive neighbourhood examination usually is not affordable. Although there are some possible tricks to help the algorithm work better than a standard Local Search routine (e.g. multiple random starting points), it is difficult for a Local Search heuristic to avoid getting trapped in local optima.

The Tabu Search algorithm on the other hand is meta-heuristic approach. It makes use of searching information from past history, complements prohibition-based techniques to a basic neighbourhood search heuristic, and guides the search to move beyond local optimality. A Local Search quits at the first local minima encountered while Tabu Search accepts even deteriorating moves and thus continues to search in various regions. Results obtained show that, with intelligent and systematic use of the past search history, Tabu Search is capable of producing good results.

Simulated Annealing is a type of probabilistic hill-climbing meta-heuristic ap-

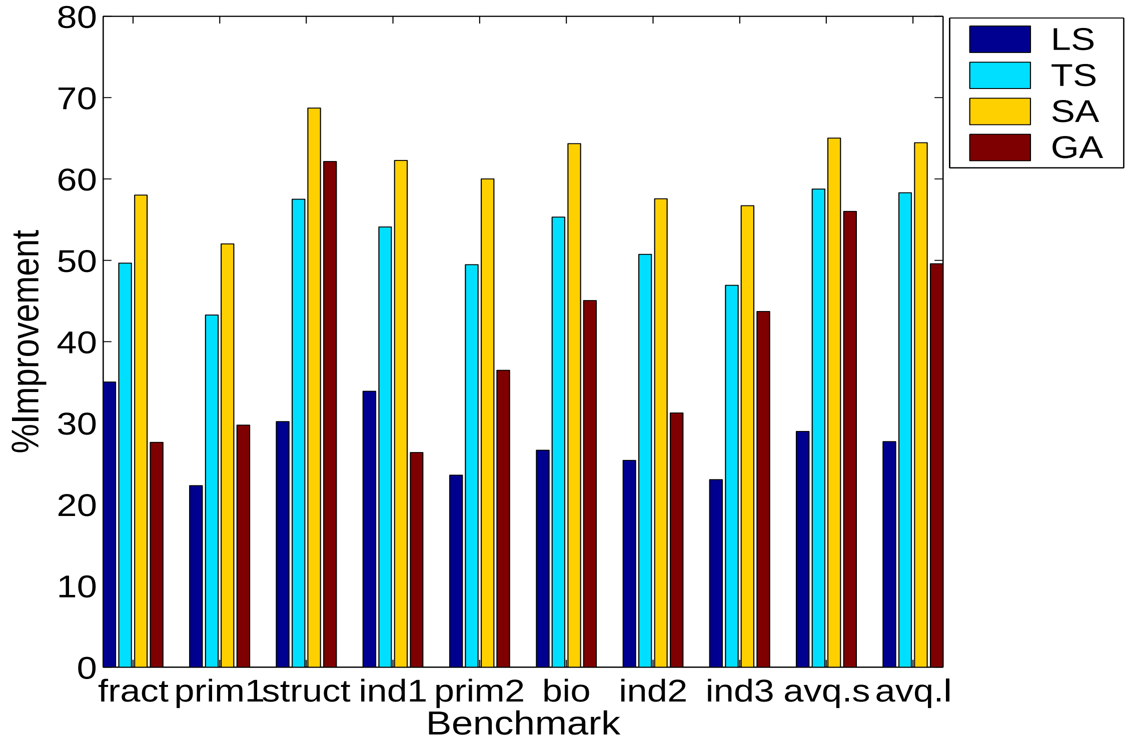


Figure 3.23: Comparison of LS, TS, SA and GA (%Improvement)

proach that guides the search by accepting bad moves with a certain probability. Experimental results indicate that SA is a powerful optimization technique that can produce very good quality solutions. However, one disadvantage of this approach is that it generally demands longer computation time.

Local Search, Tabu Search and Simulated Annealing algorithms are single-point heuristics, which means that they operate on only a single solution at a time. This in place hinders the capability of these algorithms to explore the solution space effectively. Genetic Algorithm maintains a large population of solutions optimized simultaneously during each iteration. This is extremely helpful in finding good solutions when the landscape is irregular. It is evident that GA can produce good quality solutions comparable to other advanced meta-heuristics. However, GA generally requires relatively large amount of CPU time to solve problems. In the current implementation, there are two issues causing the long CPU time of GA: (i) GA operates a number of solutions in each generation while LS, TS and SA work on one solution at a time; and (ii) After mating of two parents, a repairing process is required, which is timing consuming.

3.6 Summary

In standard cell placement, moving a cell to a different location usually requires shifting modules, which is time consuming. The alternative strategy of allowing an illegal layout during the search was implemented for VLSI placement problem. A comparison between the two placement strategies shows that the second strategy displays great advantages in terms of computation time and solution quality. A legalization procedure is required at the end of this method, which unfortunately decreases part of the improvement obtained. Based on this approach, a Local Search heuristic and several meta-heuristics such as Tabu Search and Simulated Annealing were implemented in the iterative optimization stage for placement. Experimental results display that the meta-heuristic algorithms effectively reduced the total wire length and yielded good quality of placement solutions.

The fast-paced nature of VLSI design demands efficient CAD tools. Parallelism is an effective approach to achieve faster performance. Genetic Algorithms are parallel in nature and can be efficiently parallelized by a distributed processing system. In the next chapter, the parallel Island-based GA implementation on a cluster of networked workstations will be presented[2].

[2]The Local Search, Tabu Search and Simulated Annealing are inherently serial algorithms. Although many studies implemented these algorithms in parallel on loosely-coupled systems, their parallel efficiency distributed processing system is not promising.

Chapter 4

A Parallel GA Implementation

VLSI standard cell placement is an NP-hard problem [Chan99]. Various heuristic optimization techniques have been proposed and applied to this problem in the past. Genetic Algorithms have proven to be able to produce high-quality placement solutions for many combinatorial optimization problems including standard-cell circuits and competitive with other sophisticated algorithms such as Simulated Annealing, Tabu Search and force-directed algorithms [Klin90]. However, since the runtime of Genetic Algorithms is relatively longer than other algorithms, they are becoming less competitive in the real world [Kiln89, Shah90b]. Therefore, researchers are seeking parallel GA implementation for better performance.

Island-based GAs are coarse-grain parallel models that can be easily mapped to existing distributed parallel systems similar to the cluster of networked Sun workstations in Intelligent System Lab or the high performance SHARCNET computing facility [SHAR04]. Since a fairly good speedup can be easily achieved [CP00], the Island-based GA is becoming the most popular parallel method. In this method, the population is divided into subpopulations (also known as demes) and distributed among multiprocessors. These semi-isolated subpopulations are executed as normal Genetic Algorithms, except that they would interchange a few strings among them occasionally. By introducing migration, parallel island models have often been reported to display better search quality than that of a serial GA [Whit97, CP00].

Migration is very important to the search quality and parallel efficiency. Usually considered to be a good migration scheme, Delay-migration schemes are techniques where migration occurs when the demes approach convergence. However, in standard-cell placement, this approach is not suitable since large circuits require an extremely long time to converge. We propose a practical migration scheme for placement, and describe a successful synchronous implementation. Besides the migration scheme, our contribution also includes an implementation of an asynchronous model for placement that can achieve super-linear speedup.

The methods of migration schemes determined by convergence are very popular. [Mune93] proposed algorithms where migration starts only near subpopulations convergence. In [Brau90], migration occurs after the subpopulations completely converge. [Brau90] claims that if migration is introduced too early before search convergence, good original schemata in the demes may be destroyed by incoming migrants and thus the demes may lose their diversity.

However, large circuits require an extremely long time to converge. Our experimental work shows that for medium circuits, convergence takes several hours even if we parallelize the GA using 7 processors. Since using convergence to trigger migration is not practical for placement, we employ a better migration scheme in the implementation of the synchronous island-based GA. In addition, an asynchronous model that can achieve super-linear speedup is implemented and discussed.

4.1 Synchronous Island-based GA

In synchronous island-based GA, the total population is divided equally into several subpopulations, and each processor is assigned a single subpopulation to form an island. As depicted in Figure 4.1, the algorithm in each island is simply a regular serial steady-state GA that includes a migration process (i.e each algorithm includes an extra phase of periodically exchanging individuals). More specifically, for every certain number of generations (known as migration interval), all islands exchange a candidate of good individuals (migrants), based on the communication topology. For each island, if an incoming migrant is better than the worst existing individual, this migrant is injected into the subpopulation. Migration is performed simultaneously by all islands within a certain interval.

The mechanism to achieve the functionality of island models is complex, since many parameters affect the quality of search as well as efficiency of parallelism. Besides the basic GA parameters, other parameters are involved such as deme size, migrant selection method, migration interval, migration rate, communication topology, and replacement rules.

4.1.1 The Deme Size and the Number of Islands

For a specific population size, the deme (subpopulation) size is determined by the number of processors used. The more processors utilized within the distributed system, the smaller the size of subpopulations, and the shorter the computation

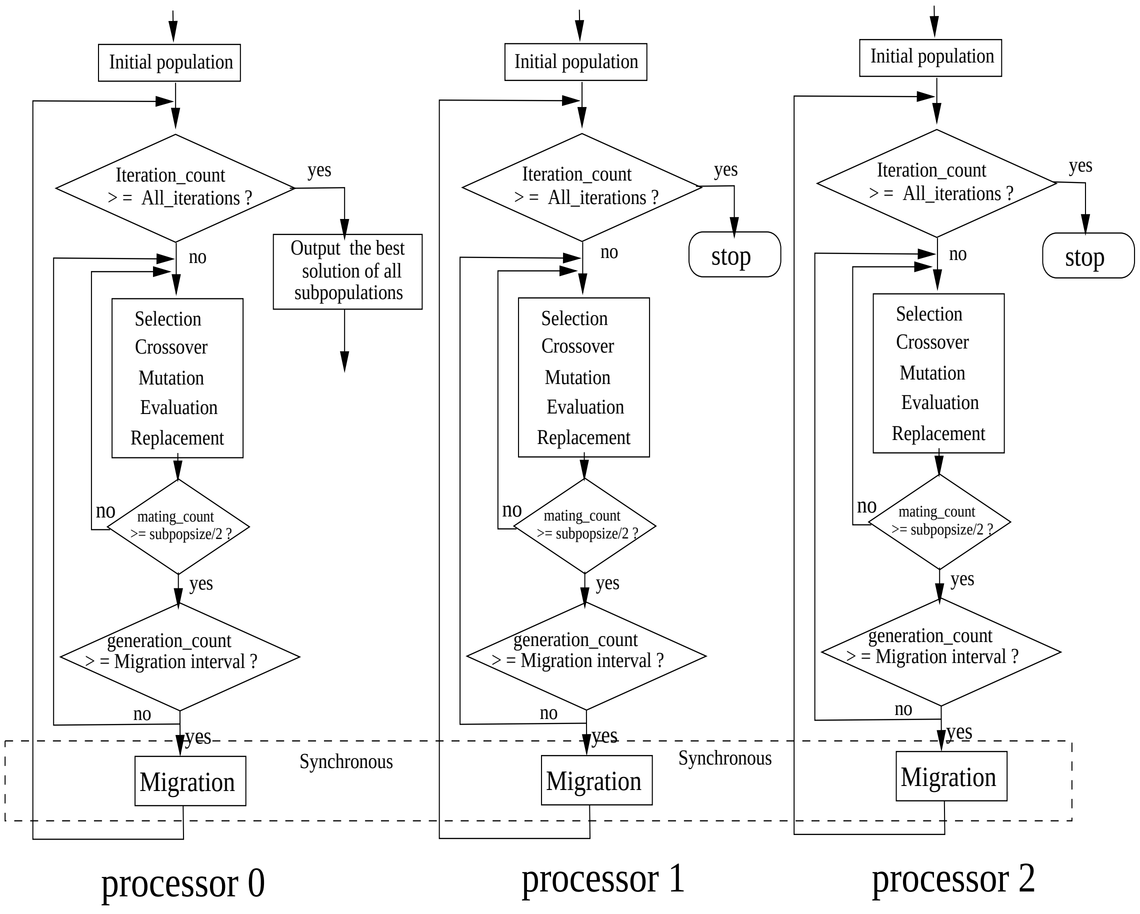


Figure 4.1: A Synchronous Model with 3 Processors

time. However, this is not always the case, since the execution of a parallel GA includes not only computation time but also synchronization and communication. As more processors are used, the average available bandwidth for each processor decreases, but more communication is required for each processor. The latter grows especially fast in a low bandwidth parallel system. On the other hand, a very small subpopulation might prematurely converge, leading to poor solutions. Therefore, when a parallel GA system is scaled up to pursue better performance, the parallel efficiency usually diminishes.

4.1.2 Migration

Migration is the process where subpopulations occasionally exchange some individuals. It is an important mechanism to improve the solution quality (diversify search) and affects the efficiency of the parallelism. If the migration is low, the subalgorithms on different small subpopulations work independently and the final result may be worse than a serial GA. On the other hand if the migration is high, the parallel GA behaves very similarly to a serial GA [Whit97, CP00]. There are several factors that affect the performance of migration, including:

- **Migrants Selection Methods**: Among the chromosomes in an island, high-quality individuals are chosen to be sent out as migrants. Several techniques can be used to select individuals such as picking the best. Other techniques involve selecting the best individuals probabilistically with methods such as Roulette-wheel selection, Stochastic universal selection, or Binary tournament selection. The higher the pressure, the faster the algorithm reaches equilibrium while sacrificing more genetic diversity of migrants [CP99a]. In the current implementation, the outgoing migrants are chosen from a pool of the best individuals, with the restriction that each individual is sent out once from the same processor. In addition, the algorithm restricts an incoming immigrant from being sent out directly but allows such a mechanism for children. In the "To-All" communication scheme, only the best unsent individual of a deme can be sent out in each phase of migration.

- **Migration Interval**: Migration intervals specify the migration plan (how often should migration occur). For standard-cell placement, we use a fixed epoch in our synchronous model, and initiate migration early in the evolution process. This can be attributed to the following: (i) it is simple; (ii) for medium/large circuits, convergence could take too long. Also if a fixed migration interval and migration scheme are used, we could make distributed demes behave like a single panmictic population.

- **Communication Topologies**: The communication topology determines the pattern of communication and connectivity among subpopulations. The ring topology is extremely popular, since it has the longest diameter and demes are more isolated compared to other topologies. With a ring topology, the parallel search may find better solutions than that of a serial GA after some generations [CP99b]. However, since the demes are quite isolated, the small deme size usually produces solutions worse than those of a serial GA with a single population in early generations. To solve this problem, a complete graph topology called "To-All" topology is used. With such a topology, all demes are fully connected. During migration, each island contributes the best individual to the global migrant pool. Since the connection between demes is very strong in this topology, the frequency of migration must be controlled. In addition, the number of migrants must be carefully set, so that the strength of migration is not excessive. As shown in Figure 4.2, only a portion of the best individuals in the pool is merged back into the subpopulations. With this migration scheme, a global super-fit individual can move across the overall

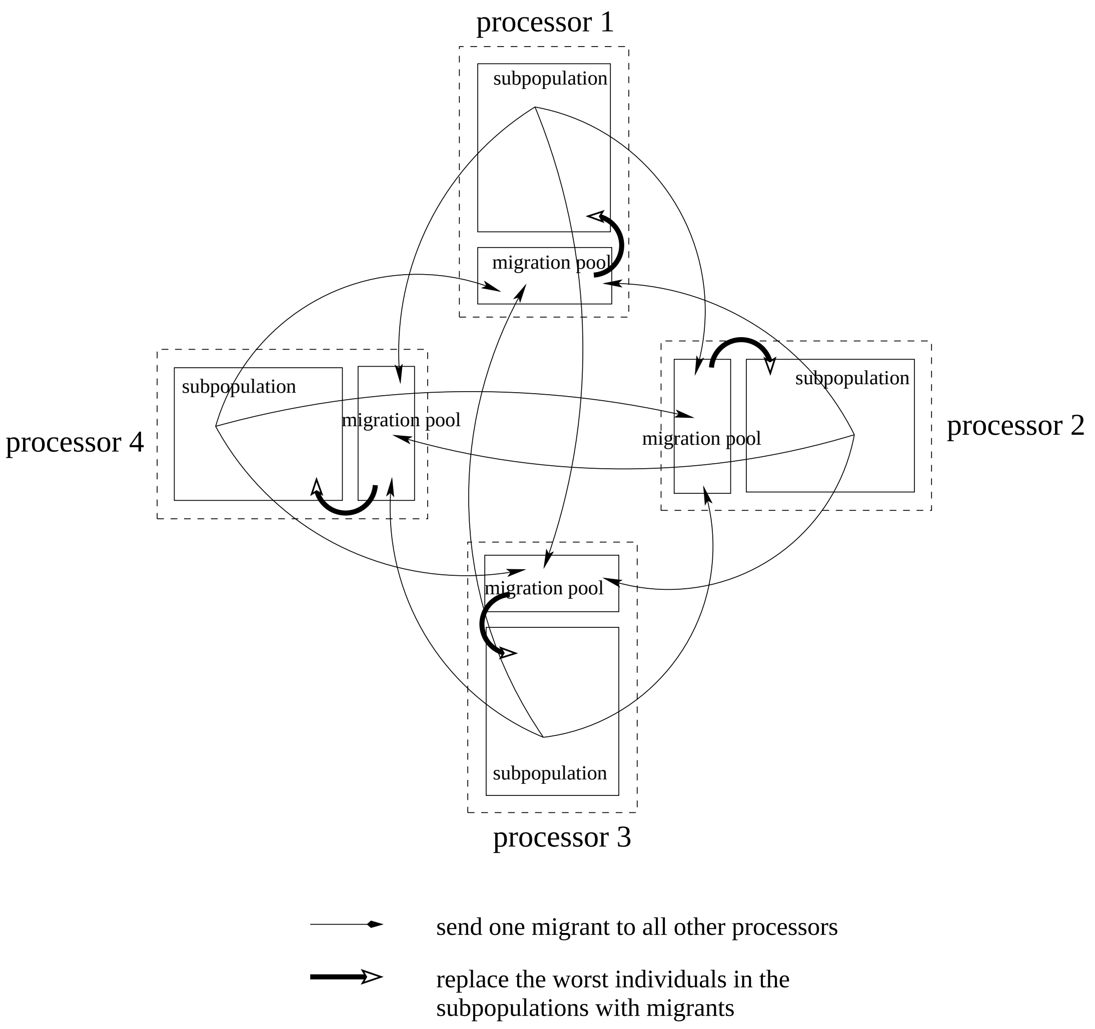


Figure 4.2: The Migration Scheme of the “To-All” Topology

population in every migration interval, while some degree of diversity between the subpopulations is maintained. Accordingly, the synchronous Island-based GA may have the chance to outperform its serial version in a very short time. Also, due to the high connectivity among islands, this synchronous model would search in a trajectory similar to that followed by a serial GA.

- **Migration Rate**: In the implementation, if the ring topology is used, the migration rate is defined as the percentage of the individuals in a subpopulation that need to be sent out as migrants. On the other hand, if the “To-All” topology is used, the migration rate specifies the percentage of migrants in the migration pool that will spread to all subpopulations. A very high migration rate often leads to two problems: (i) high communication costs; and (ii) super-fit migrants may dominate and lead to premature convergence. Search quality usually benefits from an appropriate migration rate.

- **Migrant Replacement Rule**: When an island receives new migrants, the worst individuals are replaced with these incoming migrants. For the “To-All” migration scheme, although each island sends out only one migrant, it may receive more than one incoming individual to replace the worst members in its subpopulation.

The main disadvantage of synchronous models is that the migration of all islands occurs at the same time, and the faster processors have to wait for the slower ones.

4.2 Asynchronous Island-based GA

To further enhance the performance of the Genetic Algorithm, an asynchronous island-based GA is proposed and implemented for standard-cell placement. In the asynchronous model, islands are free to evolve and migrate their individuals without further dealy. As illustrated in Figure 4.3, processors involved are configured as either a master or a slave, where the latter is responsible for holding a subpopulation and the master controls the processing in a centralized manner.

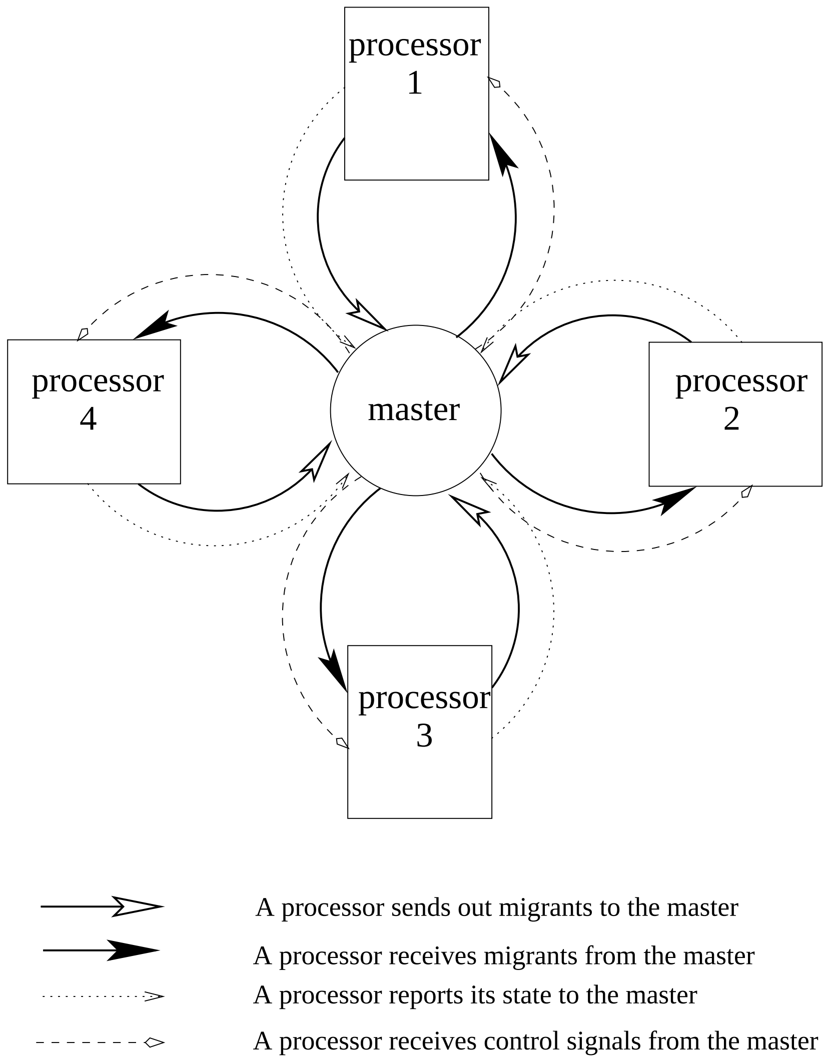

processor 1
processor 4
master
processor 2
processor 3
A processor sends out migrants to the master
A processor receives migrants from the master
A processor reports its state to the master
A processor receives control signals from the master

Figure 4.3: The Migration Scheme of the Asynchronous Model with Ring Topology

4.2.1 The Master

The master has the following responsibilities in the asynchronous model:

1. **Migration controller**: Controls the overall migration activities and dynamic load balancing, including routing migrants, communication topology,

migration plan, and dynamic load balancing. Figure 4.3 illustrates how slaves report their current states to the master periodically and the means by which the master sends control signals to different processors.

2. **High speed communication router**: In the current implementation, there is no direct communication between slaves, and migration is performed through the master. Since the master continuously listens to the communication channels, it is able to respond to the requests from slaves almost immediately. As soon as the master receives migrants from a slave, it buffers these messages, and controls the migration scheme. Incorporating a master in the system eliminates virtually all delay that might occur between processors. The master can easily choose a communication topology. Since the "To-All" topology is synchronous in nature, the asynchronous model uses an alternative topology in the form of a "Ring".

4.2.2 Dynamic Load Balancing

Since subpopulations are free to evolve, some processors might finish their task much earlier than others. This leads to two problems: (i) If a migration occurs between an evolving island and an idle island, the migration becomes meaningless; (ii) Majority of processors have to wait for the slowest processor to finish its processing. Therefore, a load balancing mechanism is needed to remove some load from slower processors and transfer it to faster processors.

In Matthem's algorithm [Mcma98], when a processor completes its work early, it becomes idle and the algorithm redistributes work assigned to other processors to this idle processor. However, a possible problem anticipated in this algorithm is that migration now is between two portions of the same subpopulation instead of two different demes. Therefore, a different strategy for dynamic load balancing is used. Instead of redistributing the work of slower processors and replacing the whole subpopulation of the faster processor, the algorithm periodically detects the evolution speed of various demes, and dynamically changes the sizes of the subpopulations to match the speed.

4.2.3 Distributed Random Migration Plans

In the real world, *islands* are independent and semi-isolated. Since subpopulations are free to evolve in the asynchronous model, extra independence is granted to these subpopulations by (i) applying distributed random migration plans created in each island; and (ii) using a random number of migrants (with some control) for each migration. With such distributed random plans, the subpopulations are allowed to evolve asynchronously. The strategy attempts to simulate the fact that the number of migrants in each migration is not constant in the real world.

4.3 Experimental Results

The serial steady-state GA is capable of producing good placement solutions. Therefore, based on the sequential implementation, the island-based Genetic Algorithm is parallelized using the MPI (Message Passing Interface) v.1.2.5. Currently, the parallel platform utilized is the cluster of networked Sun workstations, which is a distributed memory parallel architecture.

Several parameters affect the performance of an island-based parallel GA, thus a large number of runs were carried out. The benchmark suit used in this chapter is the same as the one introduced in Chapter 3. In the experiment, two to seven islands were used for distributed processing. Each test was conducted 20 times and the average values were recorded. In order to compare the performance and search quality of the island-based GAs with serial GA, the same basic GA parameters were maintained whenever possible. Experiments were conducted with various benchmarks. However, when similar results are obtained from the same setup, plots of a typical circuit (i.e *struct*) is presented.

4.3.1 Measuring Search Quality and Parallel Efficiency

Similar to the experiments conducted in Chapter 3, improvement and execution time are used as criteria to measure the search quality and algorithm efficiency. However, with a parallel algorithm, the overall execution time is not only the CPU time for the task, but also includes the overhead due to communication across the parallel system (i.e synchronization among islands). Speedups and parallel efficiency can be easily obtained by measuring the execution time of the parallel algorithm over its serial version. However, it should be noticed that the parallel efficiency measured accordingly is not sufficient to determine the suitability of a parallel algorithm since communication overhead is not completely determined by the algorithm, while the overhead significantly affects parallel efficiency. Thus, for the same parallel algorithm, the efficiency varies on different parallel architectures, different problem sizes, and even different runs.

4.3.2 Performance: Speedup vs. Processors

The amount of speedup achieved from the distributed computing system is considered the most important measure. The performance of the synchronous model with the “To-All” and “Ring” migration schemes for circuit *struct* are illustrated in Figure 4.4. In this experimental setup, the distributed system is configured

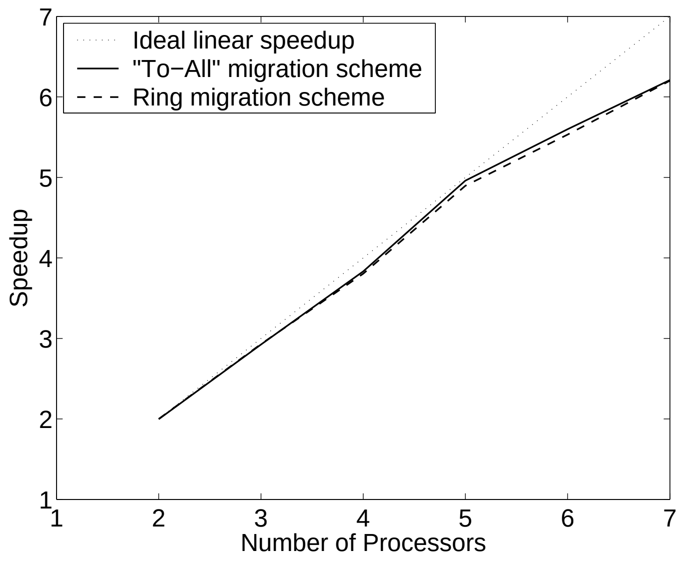


Figure 4.4: Speedup vs. Processors: Synchronous Models

with several processors ranging from two to seven. The parallel algorithms use the same empirical GA parameters of the sequential implementation. Results indicate that both migration schemes obtained near ideal linear speedup, especially when the system had less than six processors. The speedups of the “To-All” topology were slightly better than the Ring’s. As expected, when the number of processors involved increases, the communication overhead cost increases, and a decrease in parallel efficiency is observed.

The asynchronous model obtained better speedup than those obtained by the synchronous models (see Figure 4.5). With asynchronous migration and dynamic

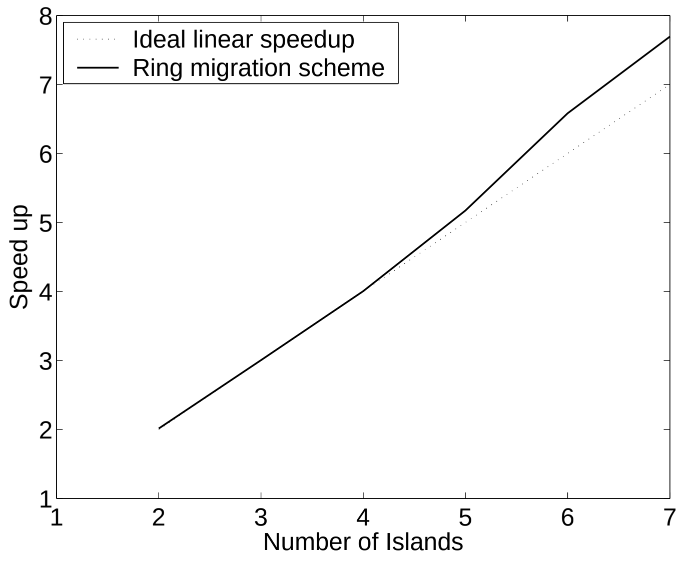


Figure 4.5: Speedup vs. Processors: Asynchronous Model

load balancing, the asynchronous model achieved a super linear speedup (i.e 7 islands obtained a speedup of 7.6). [Alba99] explains why super linear speedup is possible for asynchronous models, and [Andr96] acquired similar results with their implementation, although they were solving a different problem.

4.3.3 Solution Quality

In order to compare the search quality of the proposed “To-All” migration scheme with that of the “Ring” migration scheme, search results of circuit *struct* were plotted for 40,000 generations as seen in Figure 4.6 (a serial GA is also plotted as baseline). In these experiments, seven processors were used. To ensure fairness, the

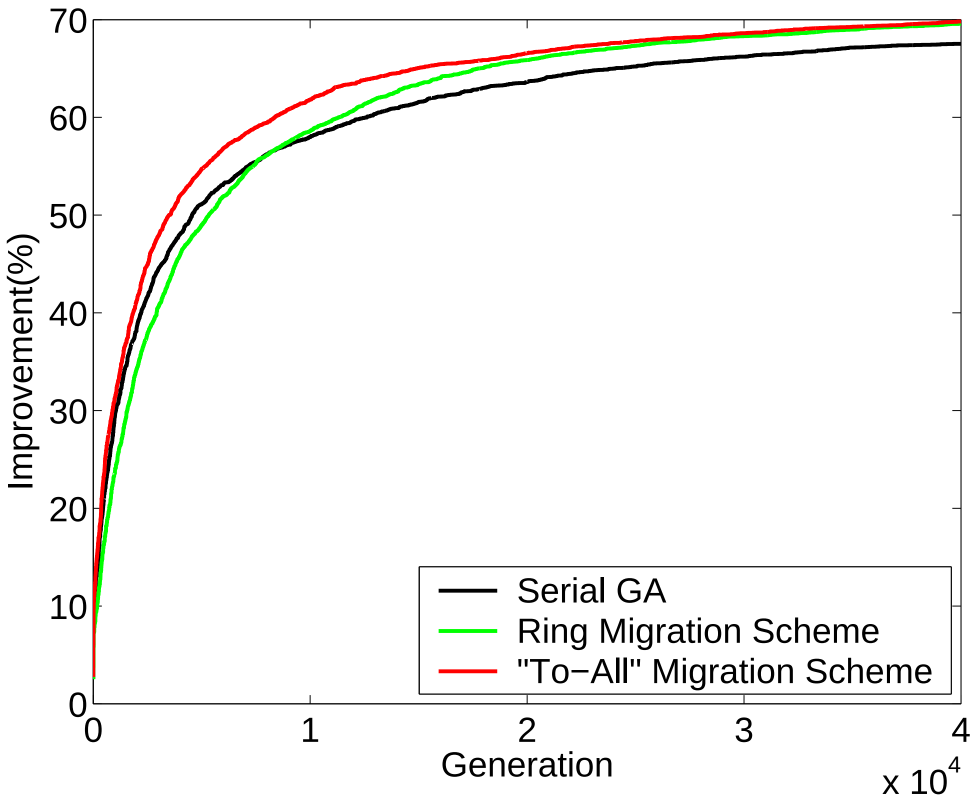


Figure 4.6: Placement Quality: Serial GA and Ring/“To-All” PGAs

same basic GA parameters are maintained.

Similar to other implementations in the literature, the parallel search with ring topology outperformed the serial GA in the medium and later generations ($>$ 8000 generations, in this case). However, in the literature, no implementation with ring topology (as well as ours) was able to surpass its serial version in early generations. Usually, to obtain a better solution than a serial GA, the ring topology parallel GA requires a large enough number of generations to evolve. The proposed “To-All” migration scheme, on the other hand, works better than a serial GA almost all the time. The main advantage of this migration strategy over a “Ring” topology is its capability of producing better results than the serial GA for any number of generation size used. It is important to note that, the execution time for both migration schemes is almost the same.

In this experimental setup, a large number of generations were carried out as seen in Figure 4.6. However, we are especially interested in the search quality of the initial phase of the search, since running a circuit placement with a GA for 40,000 generations is usually not practical for large circuits. Results in Table 4.1 compare

Benchmark	**Speedup**	**Better Quality** of Solutions
ckt1_52.yal	6.38	2.9%
struct.yal	6.25	0.22%
prim2.yal	6.264	0.024%
avq_large.yal	6.063	0.1366%
* 7 processors were used for PGA		

Table 4.1: Search Results in the 100th Generation

the synchronous island-based GA with a serial GA in terms of execution time and search quality, especially their performance in the early generations. The migration scheme used in the experiments is the "To-All" topology. The algorithms ran for only 100 generations. Seven processors were used for the parallel GA, and these experiments involved several circuits. It can been seen from the results that the parallel implementation achieved speedups from 6.06 to 6.38 with seven processors, and the proposed migration scheme yielded improved solutions quality over the serial GA at very early generations (the 100th generation in this case).

4.3.4 Migration Interval

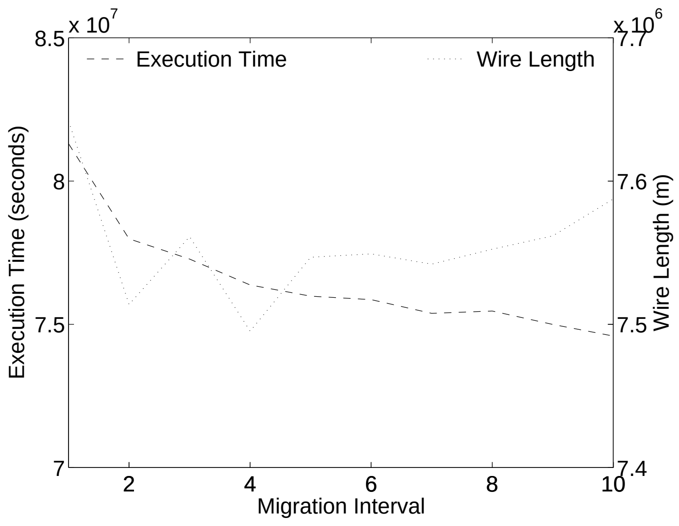
The migration interval as explained earlier indicates the frequency of migration. Figure 4.7 and Figure 4.8 introduce migration results based on synchronous and asynchronous models respectively.

In the synchronous models when migration occurs too frequently, the search may quickly end up with identical subpopulations and converges prematurely. A longer migration interval helps the subpopulation to have more divergence. However, if the interval is too long, migration becomes insufficient, and may lead to worse solutions. Also, the shorter the migration interval, the more communication overhead involved. For example, a migration interval of 4 produces the best result for the "To-All" scheme as shown in Figure 4.7(a) and an interval of 2 is best for the ring scheme as shown in Figure 4.7(b).

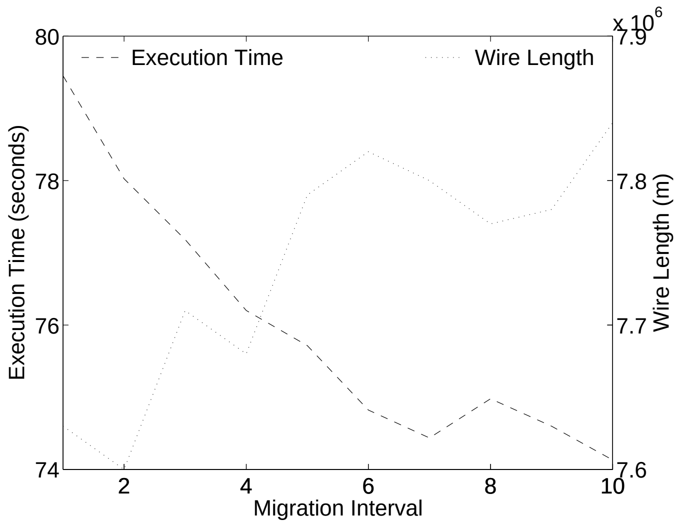
In the asynchronous model, the interval of 10 (as evident from Figure 4.8) produces the best result.

4.3.5 Migration Rate

The migration rate determines the number of chromosomes that need to be exchanged in a certain interval. Experimental results of the synchronous model are illustrated in Figure 4.9. It is evident from Figure 4.9 that the parallel execution time grows in proportion to the migration rate. This is because in the synchronous model, the execution time required by each iteration is the slowest processor's computation time plus the communication (migration) time. The larger the migration rate, the more communication involved.



(a) *"To-All"* migration scheme



(b) *Ring* migration scheme

Figure 4.7: Synchronous Model: Migration Interval

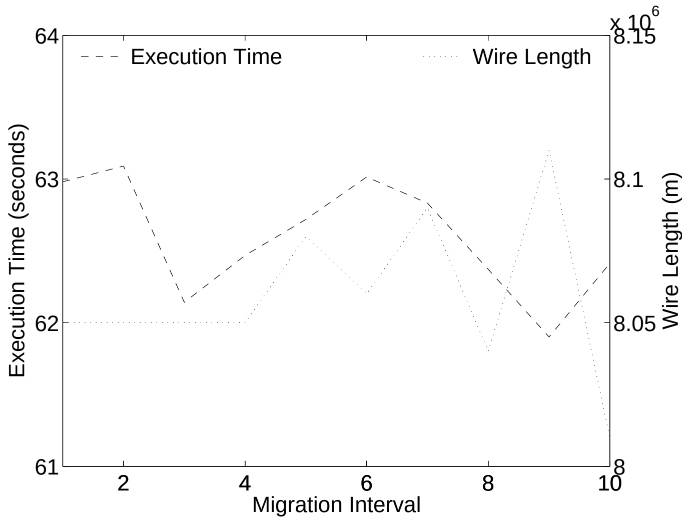


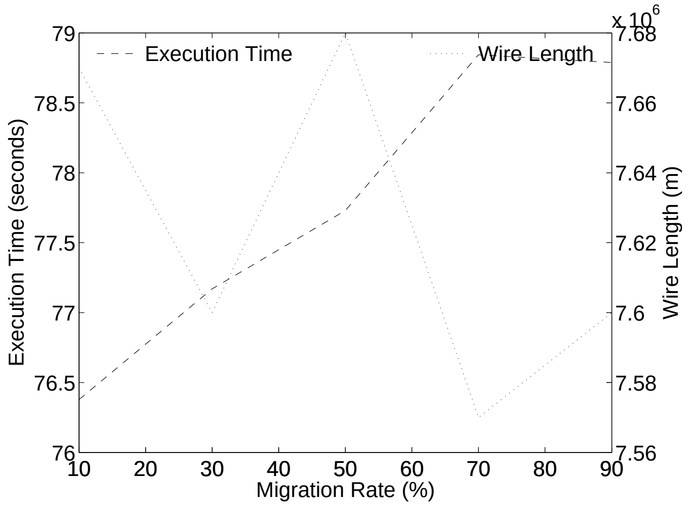
Figure 4.8: Asynchronous Model: Migration Interval

However, in the asynchronous model (Figure 4.10), the migration rate has a minor affect on the overall execution time. This is due to the fact that the computation time conceals most of the communication costs when the computation time is much greater than the communication time.

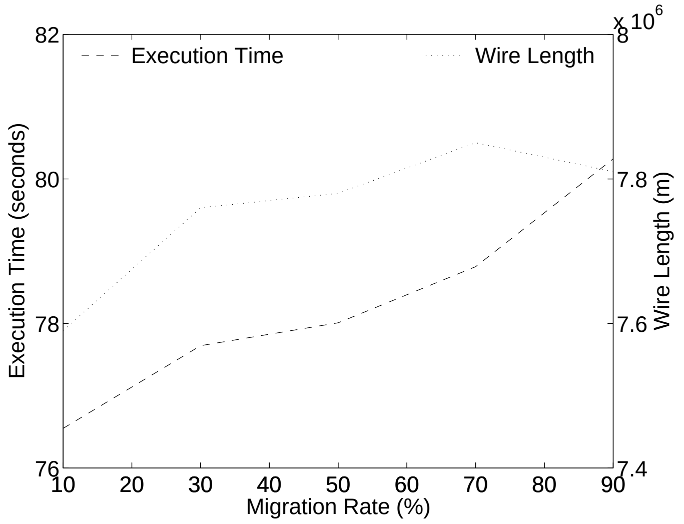
4.3.6 Scalability Analysis

In this section, experiments are conducted to analyze the scalability of the island-based GA algorithm running on a loosely-coupled parallel system.

In island-based GAs, speedups are obtained by dividing the overall population into multiple subpopulations and processing them on multiple processors simultaneously. The more processors applied, the faster the execution expected. However, Figure 4.4 in previous Section 4.3.2 indicates that as more processors are used the speedup grows slower and the parallel efficiency decreases. As more islands are



(a) *"To-All"* migration scheme



(b) *Ring* migration scheme

Figure 4.9: Synchronous Model: Migration Rate

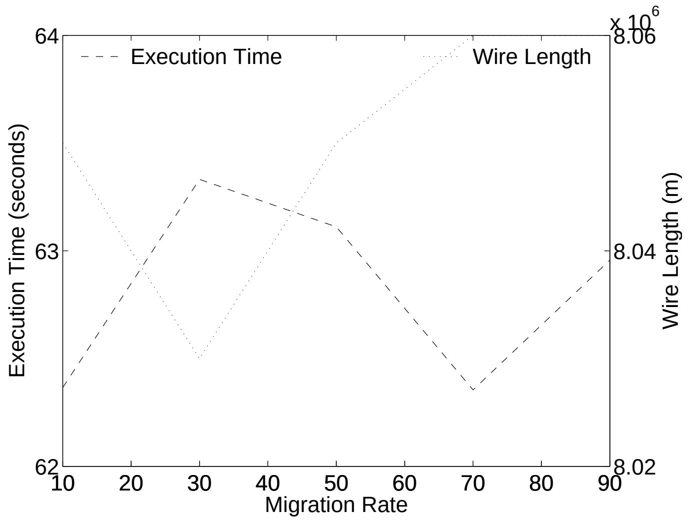
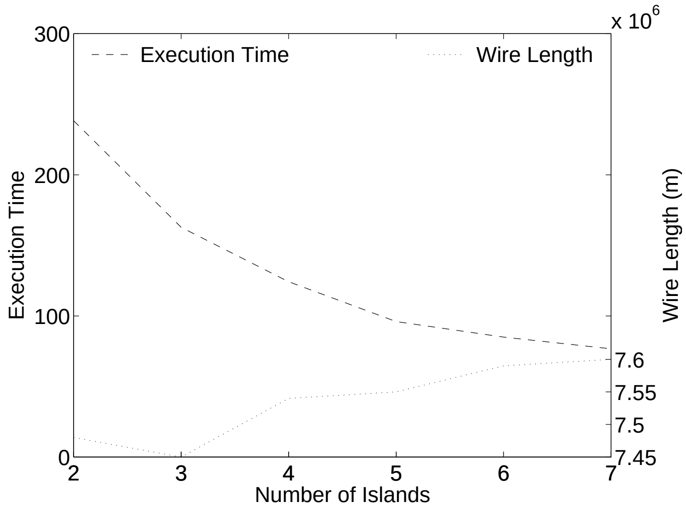


Figure 4.10: Asynchronous Model: Migration Rate

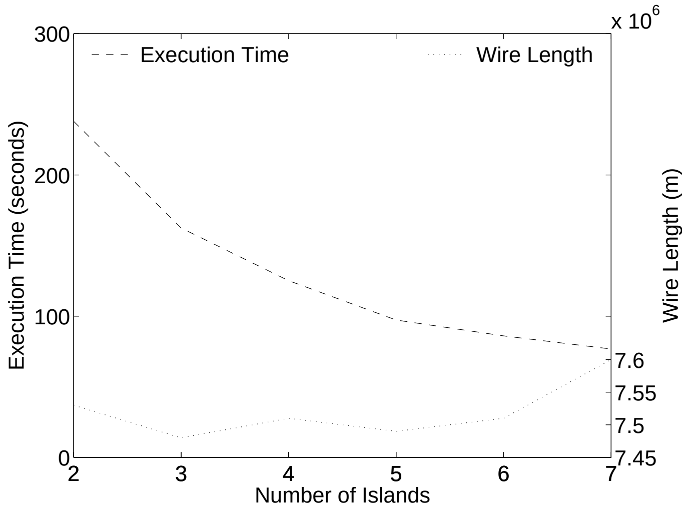
utilized, the average available bandwidth for each island is expected to decrease, but more communication is required for each island. Thus the computation time for each processor decreases, but the communication time for the system increases. The latter increases faster in a low bandwidth parallel system. Similar results are obtained, as Figures 4.11(a) and 4.11(b) illustrate. On the other hand, a small population may result in poor search quality. The figures also show that although migration mechanisms may help to yield solutions slightly better than the serial GA, a very small subpopulation may prematurely converge and lead to poor solutions.

Therefore, the synchronous island-based GA implementation can be scaled up to eight processors to achieve fairly good speedup and produce even better solutions than a serial GA. As more processors are used, the parallel efficiency and the improvement usually diminish as expected.

The asynchronous model has better scalability than the synchronous model in



(a) *"To-All"* migration scheme



(b) *Ring* migration scheme

Figure 4.11: Synchronous Model: Scalability

terms of parallel efficiency, as Figure 4.5 illustrated previously. However as with the synchronous model, the solution quality decreases as the parallel system scales up.

4.4 Summary

In this chapter, two island-based GAs were introduced based on synchronous and asynchronous models. A practical migration scheme of synchronous island-based GA was also presented for the circuit placement problem. Experimental results indicate that the synchronous model can achieve near linear speedup while the asynchronous model can achieve super-linear speedup. In addition, the proposed migration scheme for the synchronous GA yields better quality of solutions than a serial GA from an early stage of evolution. This emphasizes the advantage of the proposed migration scheme for such application as the standard-cell placement problem.

Chapter 5

Preprocessing & Postprocessing Techniques

Thus far we have described the implementation of several iterative improvement techniques that proved effective for solving the placement problem. However, the high computational complexity of placement demands other types of preprocessing/postprocessing techniques to be utilized to further improve solution quality and reduce CPU time.

In order to save computational effort, several preprocessing techniques are proposed. These techniques tend to optimize the position and orientation of rows before the iterative placement process begins, in an attempt to move strongly connected blocks of cells closer. Such a strategy may avoid inefficiently interchanging these cells individually in the main iterative improvement stage. In addition to tuning the rows, extremely long nets in the system may also be eliminated in the preprocess procedure.

In addition, several postprocessing procedures were introduced following iterative improvement to further reduce the total wire length. These postprocessing procedures involve optimizing the position of I/O pads and cell orientation.

5.1 The Preprocessing Procedures

Standard-cell placement is an NP-hard problem. Due to the high computational complexity of the problem, extra preprocessing steps, where possible, are used to save computation time and enhance algorithm efficiency. The preprocessing procedures before iterative improvements involve:

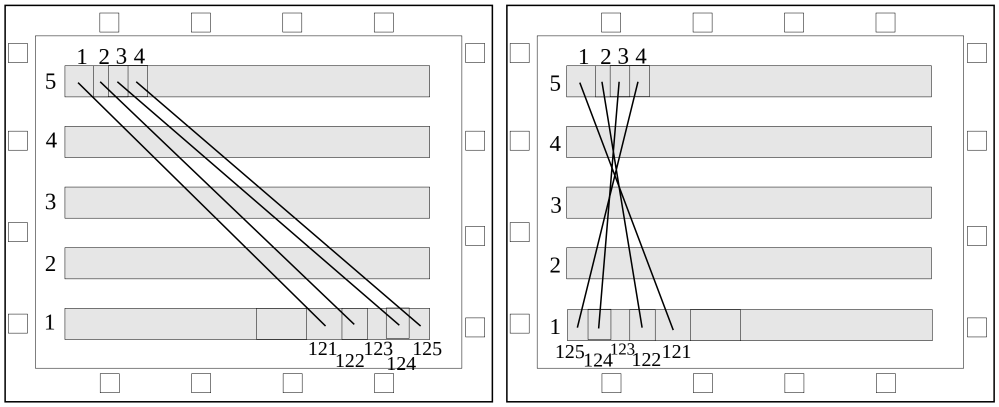
- Eliminating long nets;
- Optimizing row position;
- Optimizing row orientation.

The basic idea behind these algorithms is to apply simple techniques to improve solution quality with a minimum computation effort before iterative optimization.

5.1.1 Optimizing Row Position/Orientation

Initial solutions constructed by the first stage of placement may contain modules that have strong connections to other blocks, but separated from each other. Although iterative improvement algorithms tend to search for more optimized solutions by placing modules with high connectivity closer, achieving this might be quite expensive in terms of CPU time. If strongly connected blocks can be placed together with less effort before the main loop of optimizing positions of individual cells, there could be substantial savings in computation time. Row swapping and optimizing row orientation are proposed as candidates in the preprocessing stage to optimize strongly connected modules together.

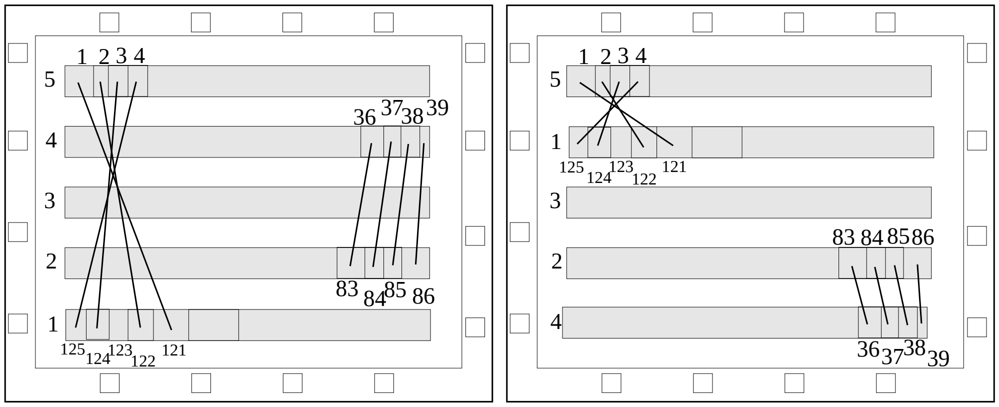
In Figure 5.1(a), the cluster of cells C_1, C_2, C_3 and C_4 have connections to another cluster of cells C_{121}, C_{122}, C_{124} and C_{125}. However the first cluster lies on the left while the other is located on the right. Obviously, changing the orientation of row 1 can be easily performed and will tend to reduce the length of wire length in the x direction, as illustrated by Figure 5.1(b).



(a) Before Changing (b) After Changing

Figure 5.1: Change Orientation of Row 1

Figure 5.2 gives an example of swapping rows. In Figure 5.2(a), row R_1 and row R_5 have a strong connection, and row R_2 is well connected to row R_4. Since row R_1 and row R_5 are far away from each other, and so are row R_2 and row R_4, swapping row R_1 and row R_4 helps to shorten the distance between these strongly connected module blocks, as Figure 5.2(b) demonstrates.



(a) Before Swapping (b) After Swapping

Figure 5.2: Swapping Row 1 with Row 4

Figure 5.3 illustrates the pseudo code for row swapping and optimizing row orientation. Swapping rows affects only the wire length in the y direction while changing row orientation affects only the wire length in the x direction. Therefore, simple data structure can be used to evaluate the quality of neighbours for the Local Search algorithms to reduce the computational cost. In Figure 5.3(a), $Min\_y[i]$ and $Max\_y[i]$ are the bounding box information of net i in the y direction. When evaluating a move, only $Min\_y[i]$ and $Max\_y[i]$ of the affected nets are evaluated, so the computation is simple. Similarly, $Min\_x[i]$ and $Max\_x[i]$ are used in Figure 5.3(b) for the affected nets in the x direction. Thus, the algorithms do not demand a large computational effort.

```
Procedure: Rows Swapping

Choose an initial placement solution
Read Min_y and Max_y for each net

while(1)
    BestImprovement = -1
    for( all neighbours )
        CurrentNeighbour = Generate_A_Swap()
        improvement = Evaluate_the_Swap()
        if ( improvement > BestImprovement ) then
            BestImprovement = improvement
            BestNeighbour = CurrentNeighbour
    end for
    if ( BestImprovement ≥ 0 ) then
        Perform the row swapping of BestNeighbour
        Update circuit information
        Read Min_y and Max_y for affected nets
    else break
end while
```

(a) Optimizing Row Position

```
Procedure: Row Flipping

Choose an initial placement solution
Read Min_x and Max_x for each net

while(1)
    BestImprovement = -1
    for( all neighbours )
        CurrentNeighbour = Generate_A_Change()
        improvement = Evaluate_the_Change()
        if ( improvement > BestImprovement ) then
            BestImprovement = improvement
            BestNeighbour = CurrentNeighbour
    end for
    if ( BestImprovement ≥ 0 ) then
        Perform the row swapping of BestNeighbour
        Update circuit information
        Read Min_x and Max_x for affected nets
    else break
end while
```

(b) Optimizing Row Orientation

Figure 5.3: The Outline of the Algorithms

5.2 The Postprocessing Procedures

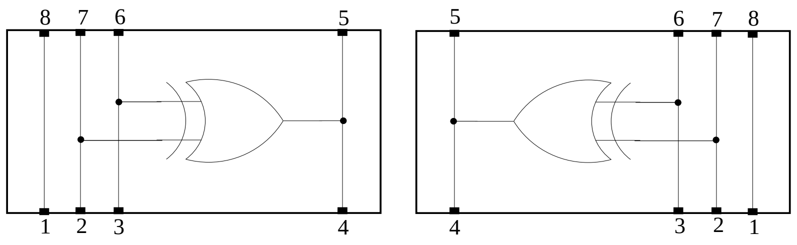
Following the iterative improvement phase, postprocessing procedures can be used to further reduce wire length. In this stage, cell placement has already been performed and all the cells have been assigned to fairly good physical locations within the chip. The only remaining issue involves optimizing the locations of I/O pads and the orientations of cells. Since this skep does not create any overlap or any change of row size, legalization is not required. The postprocessing procedures after iterative improvement include:

- Optimization of I/O pads

- Optimizing cells orientation

5.2.1 Optimizing Cell Orientation

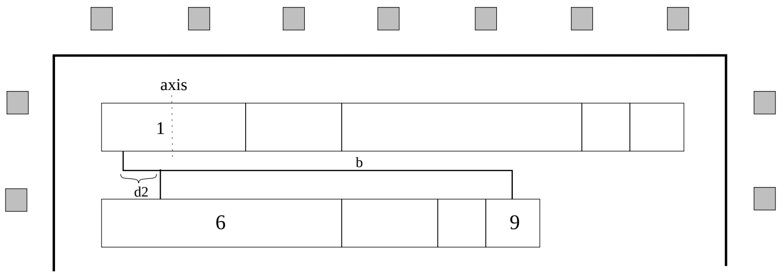
There are many possible orientations that a module may have. A module can be changed to one of eight possible orientations by flipping, rotation or inversion. However, for standard cells which are rectangles of the same height and variable width, there exist only two possible orientations: (a) the original orientation of the cell and (b) the "mirror image" of the given cell, as illustrated in Figure 5.4.



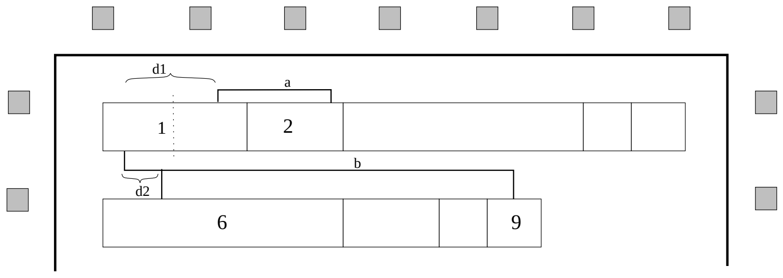
(a) Original Orientation (b) Mirror Image

Figure 5.4: Module Orientation

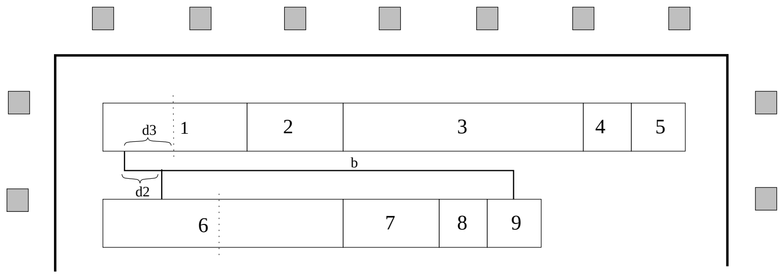
Cell flipping in standard cell placement is an operation that replaces some of the cells with their *mirror images* with respect to a vertical axis while maintaining the flipped cells in the same position slots. In certain situations, flipping the modules in place can reduce the length of some nets. Some examples are illustrated in Figure 5.5, where modules are denoted with numbers and nets are denoted with letters. In Figure 5.5(a), if cell C_1 is flipped with respect to the vertical axis of center of the cell, then net "b" will have a shorter distance; thus the total wire length will be decreased by d_2. However in Figure 5.5(b), net "b" is not lucky:



(a)



(b)



(c)

Figure 5.5: Examples of Optimizing Module Orientation

although flipping cell C_1 decreases the wire length of net "b" by d_2, it also increases net "a" by d_1. If d_1 happens to be greater than d_2, the total cost becomes higher, and this operation is not accepted. Usually in this postprocessing procedure, only the cells on the edges of bounding boxes of nets are examined. However, flipping a cell may affect other cells and cause them to sit on a bounding box. For example in Figure 5.5(c), the pin of cell C_6 was originally not on the bounding box of net "b", but after flipping cell C_1, the situation is reversed. Flipping cell C_1 followed by cell C_6 can yield an improvement of $2 \times d_3$ to the total wire length, as shown in Figure 5.5(c).

Therefore, the main task to be solved is the determination of the optimum orientation of cells by allowing every module to flip left/right so that the total wire length of all nets is minimized. Cheng *et al* [Chen91] claimed that the module orientation problem also turns out to be an NP-hard problem. However, in the implementation, instead of putting a large computation effort to gain a small improvement, a simple algorithm to optimize the module orientation is employed, as seen in Figure 5.6.

```
(1) Initialize Min_Max_Info_() for each net
    Flag := 1

(2) While (Flag)
      Flag := 0
      For each row
          For each cell on the row
              If (the cell is not flipped)
                  Gain := Estimate the Flipping
                  If (Gain > 0)
                      Flag := 1
                      Update Min_Max_Info_() for affected nets
                  End If
              End If
          End For
      End For
    End While

(3) return
```

Figure 5.6: Algorithm Outline of Module Flipping

As mentioned previously in Chapter 3, in the main optimization stage, the location of a net terminal is approximately represented by the center of the cell that the net is connected to. This approximation spares significant computation effort, and has little negative effect on the accuracy of total wire length since the offset of pins on a cell is usually much smaller than the wire length of the net. However, this simplification model is not used in this procedure since flipping cells attempts to squeeze the little improvement by optimizing the locations of these pins.

In the procedure, an alternative simple data structure is used to store the bounding box information for each affected net. Besides the cells on bounding boxes, the cells that are located close to the bounding box are also counted in to deal with the scenario posed by Figure 5.5(c). This information is the so-called *Min_Max_Info_()* in the algorithm outlined in Figure 5.6. For example, for Net_{10} the *Min_Max* information simply stores some cell numbers related to the net's bounding box:

```
Min_x[10] is the most left hand side module that connects to net 10.
Min_x_2[10] is the 2nd very left hand side module that connects to net 10.
Max_x[10] is the most right hand side module that connects to net 10.
Max_x_2[10] is the 2nd very right hand side module that connects to net 10.
```

It is important to note the following: (1) cell flipping does not affect any wire length in the y direction thus the wire length in this direction is not evaluated; (2) the offset of pin terminals is much smaller than the wire length of a net, so no significant wire length decrease should be expected from this procedure; (3) since it is not an iterative procedure and does not require shifting of cells, it is not computationally demanding.

5.3 Experimental Results

Table 5.1 presents results obtained by the *Row_Swapping()* procedure. In the

Circuit	# of Rows	Methods of Initial Solutions	Performed Moves	% Imp. in y direction	Time*
fract	6	Random	4	7.67%	0.27%
		ARP	1	0.76%	0.00%
prim1	16	Random	0	0.00%	0.00%
		ARP	1	0.51%	0.43%
struct	21	Random	12	11.79%	1.30%
		ARP	14	7.47%	2.65%
ind1	15	Random	1	0.20%	0.06%
		ARP	1	0.34%	0.04%
prim2	28	Random	0	0.00%	0.00%
		ARP	2	0.07%	0.01%
bio	46	Random	0	0.00%	0.00%
		ARP	5	0.28%	0.15%
ind2	72	Random	13	3.72%	1.33%
		ARP	17	0.28%	0.91%
ind3	54	Random	1	0.07%	0.38%
		ARP	2	0.06%	0.61%
avq.s	80	Random	0	0.00%	0.00%
		ARP	8	0.20%	0.37%
avq.l	86	Random	0	0.00%	0.00%
		ARP	15	0.18%	0.81%
Average		Random		2.35%	0.33%
		ARP		0.99%	0.60%

* Time: $\frac{Time\ for\ Row\ Swapping}{Time\ for\ Main\ Iterarive\ Improvement\ Loop} \times 100\%$

Table 5.1: Improvement by Optimizing Row Position

experiment, two different initial construction methods are used to provide initial solutions with different quality. The ARP (Attractor-Repeller Approach) [Etaw99] can produce solutions superior to the random constructive method. The table does not record the improvement in the x direction since it is not affected by the algorithm. It is evident from Table 5.1 that although the algorithm is simple, it can improve the solution quality slightly in a short period. This is expected since the algorithm is a basic Local Search and can get trapped easily in a local minima.

Similar experiments are conducted for the *Row_Flipping()* procedure. Initial

Circuit	# of Rows	Methods of Initial Solutions	Performed Moves	% Imp. in x direction	Time*
fract	6	Random	3	6.73%	0.02%
prim1	16	Random	8	4.10%	0.10%
struct	21	Random	11	7.60%	0.15%
ind1	15	Random	5	4.03%	0.08%
prim2	28	Random	12	4.13%	0.35%
bio	46	Random	19	5.75%	0.07%
ind2	72	Random	33	3.96%	3.00%
ind3	54	Random	18	2.72%	2.03%
avq.s	80	Random	33	4.81%	0.05%
avq.l	86	Random	29	2.89%	0.03%
Average				4.67%	0.59%

* Time: $\frac{Time\ for\ Row\ Flipping}{Time\ for\ Main\ Iterarive\ Improvement\ Loop} \times 100\%$

Table 5.2: Improvement by Optimizing Row Orientation

solutions are randomly constructed in the tests. As illustrated in Table 5.2, this procedure optimizes the wire length in the x direction on average by 4.67% with an extra 0.59% CPU time.

Table 5.3 displays the experimental results of the *Flipping_Modules()* procedure (optimizing module orientation). In the experiment, the procedure is executed after some iterative optimization algorithms, such as Local Search algorithm, Tabu Search and Simulated Annealing. Results indicate that, after optimizing the circuits with advanced heuristic algorithms, flipping some of the cells can still produce a small improvement in the x direction. In most cases, the computation time spent by this procedure can be ignored.

The overall performance of the preprocessing and postprocessing procedures are shown in Table 5.4. It is evident from Table 5.4 that the preprocessing optimization step can help obtain better results. When flipping/swapping rows were activated, an average improvement of 4.44% was obtained. On the other hand, the postprocessing procedure was beneficial in improving solution quality on average by 1.32%.

Circuit	Iterative Improvement Algorithms	% Imp.	% Imp. in x direction	Time*
fract	LS	35.99%	2.81%	2.44%
	TS	48.98%	2.94%	0.75%
	SA	56.32%	4.86%	0.00%
prim1	LS	20.93%	1.06%	0.00%
	TS	43.84%	1.93%	0.07%
	SA	49.90%	2.18%	0.12%
struct	LS	22.31%	0.78%	1.17%
	TS	55.01%	1.61%	0.09%
	SA	64.16%	1.92%	0.07%
ind1	LS	34.32%	0.52%	0.37%
	TS	54.31%	0.85%	0.06%
	SA	60.54%	1.08%	0.02%
prim2	LS	23.45%	0.67%	0.71%
	TS	49.89%	1.17%	0.07%
	SA	57.80%	1.42%	0.03%
ind2	LS	25.93%	0.35%	0.42%
	TS	50.11%	0.59%	0.05%
	SA	57.01%	0.67%	0.02%
ind3	LS	22.55%	0.18%	0.68%
	TS	47.61%	0.30%	0.09%
	SA	54.76%	0.35%	0.06%
avq.s	LS	30.33%	0.31%	0.03%
	TS	58.94%	0.65%	0.00%
	SA	62.70%	0.74%	0.00%
avq.l	LS	27.44%	0.29%	0.04%
	TS	58.14%	0.63%	0.00%
	SA	61.89%	0.72%	0.00%

* Time: $\frac{\textit{Time for Module Flipping}}{\textit{Time for Main Iterarive Improvement Loop}} \times 100\%$

Table 5.3: Improvement by Flipping Modules (Postprocessing)

5.4 Summary

In this chapter, preprocessing and postprocessing procedures are introduced. Preprocessing procedures attempt to save computational effort and enhance algorithm efficiency by optimizing row position and orientation before iterative placement process begins. Experimental results indicate that these techniques are effective. On the other hand, postprocessing procedures are employed to further reduce wire length. Optimizing cell orientation after cell locations are determined can slightly improve solution quality.

Circuit		With Postprocessing			With Preprocessing/Postprocessing					
		Main Imp.	Postprocessing	Total	Preprocessing			Main Imp.	Postprocessing	Total
					Flip (x)	Swap (y)	Subtotal			
fract	LS	32.6	2.3	34.9	6.73	7.67	7.2	31.7	2.5	41.4
	TS	56.3	4.3	60.6				51.1	3.4	62.3
	SA	59.0	6.0	65.0				55.1	7.0	69.3
prim1d	LS	23.2	1.2	24.4	4.10	0.00	3.19	19.6	1.2	23.99
	TS	46.2	2.0	48.2				43.2	1.9	48.29
	SA	53.4	2.7	56.1				53.4	2.5	59.09
struct	LS	25.0	0.7	25.7	7.60	11.79	8.48	23.8	0.8	33.08
	TS	55.2	1.6	56.8				55.5	1.5	65.48
	SA	67.2	2.2	69.4				63.3	1.9	73.68
ind1	LS	36.9	0.7	37.6	4.03	0.20	3.53	35.0	0.6	39.13
	TS	50.7	0.9	51.6				52.5	0.9	56.93
	SA	55.7	1.1	56.8				59.7	1.0	64.23
prim2	LS	26.6	0.7	27.3	4.13	0.00	3.38	25.5	1.0	29.88
	TS	52.8	1.4	54.2				52.0	1.5	56.88
	SA	59.6	1.4	61.0				57.5	1.4	62.28
bio	LS	27.7	0.8	28.5	5.75	0.00	5.07	25.5	0.8	31.37
	TS	55.3	1.5	56.8				53.3	1.4	59.77
	SA	61.9	1.7	63.6				61.2	1.8	68.07
ind2	LS	23.6	0.3	23.9	3.96	3.729	3.92	20.0	0.3	24.22
	TS	51.2	0.6	51.8				49.8	0.6	54.32
	SA	57.8	0.7	58.5				55.3	0.7	59.92
ind3	LS	23.7	0.2	23.9	2.72	0.07	2.44	22.4	0.2	25.04
	TS	46.8	0.3	47.1				48.3	0.3	51.04
	SA	53.4	0.4	53.8				57.0	0.4	59.84
avq_small	LS	31.5	0.3	31.8	4.80	0.00	4.46	30.8	0.3	35.56
	TS	60.0	0.7	60.7				58.2	0.7	63.36
	SA	65.7	0.8	66.5				66.4	0.9	71.76
avq_large	LS	27.5	0.3	27.8	2.89	0.00	2.71	27.0	0.3	30.01
	TS	59.4	0.7	60.1				59.0	0.7	62.41
	SA	66.7	0.8	67.5				64.6	0.8	68.11
Average			1.33				4.44		1.31	
* The numbers above are the percentage of improvement obtained in varied procedures										

Table 5.4: Performance of Preprocessing/Postprocessing Procedures

Chapter 6

Conclusions and Future Directions

The exponentially increasing number of transistors on modern VLSI circuits causes the placement problem to be extremely complicated, resulting in a long time to obtain acceptable solutions, especially for large circuits. Moreover, with advanced sub-micron technologies, the effect of interconnect delay dominates the circuit delays and becomes the key factor in determining the speed of a chip. These issues accentuate the importance of the placement problem. Therefore, effective and efficient placement techniques are necessary to deal with these new challenges.

In this thesis, three important topics related to standard cell placement were studied and addressed: heuristic/meta-heuristic optimization techniques, parallel island-based GAs, and some preprocessing/ postprocessing approaches.

Heuristic techniques have been proven to be powerful in solving NP-hard combinatorial optimization problems, including the ones in VLSI CAD design. By comparing two different placement strategies, it can be seen that the strategy of allowing illegal layouts during the search offers greater advantages than the other strategy in terms of computation time and solution quality. Based on this approach, the Local Search (LS) heuristic, Tabu Search (TS) and Simulated Annealing (SA) were implemented for the iterative improvement stage. The Local Search heuristic is a basic hill-climbing technique that iteratively attempt to find the best solution from the neighbourhood of a current solution. Results obtained indicate that wire length can be decreased on average by 27%. One of the problems of the Local Search heuristic is the pitfall of getting trapped in a local optimum, which may be very far away from the global optimal solution. Meta-heuristics (such as TS, SA and GA) are advanced techniques that apply some particular mechanisms to guide the local search in an attempt to explore/exploit solution space more effectively. The Tabu Search heuristic attempts to go beyond local optima and avoid cycling in a certain area in the solution space by making a systematic use of the search information from past history. Experimental results indicate that on average an improvement of 52.4% can be obtained using this approach. Based on stochastic techniques (Markov Chains), Simulated Annealing guides Local Search by accepting deteriorating moves with a certain probability that depends on current temperature and cost change. Experimental results display an average improvement of 61% with this algorithm. The Local Search algorithm usually obtains poor improvements, while the Tabu Search and Simulated Annealing algorithm can yield much better solutions at the expense of a longer time. In addition, it can be concluded from the experimental setup that if more computation time can be used, the TS and SA algorithms can obtain further improvement than the results obtained.

The Genetic Algorithm can produce placement solutions as good as those obtained by SA. However, the computation time for the GA is excessive. Therefore, the implemented island-based parallel GA attempts to divide a population into subpopulations and assign them to multiple processors so that they can be processed in parallel. A synchronous model and an asynchronous model of the GA were implemented on a loosely-coupled parallel system for standard cell placement. A practical migration scheme was proposed and employed in the synchronous model for large circuits. Experimental results show that the synchronous parallel GA implementation can achieve near linear speedup, and the asynchronous model obtained even super-linear speedup. Results obtained also show that by introducing migration, the synchronous PGA may produce better quality solutions than a serial GA.

The preprocessing and postprocessing stages are procedures that are executed prior or after iterative improvement, in an attempt to improve solution quality with a small computational cost. The preprocessing function in this work includes “optimizing row orientation” and “optimizing row position” by a modified Local Search algorithm, while “optimizing module orientation” is part of the postprocessing procedure. Experimental results show that the preprocessing routines can improve solution quality by an average of 4.44% with an increase of 0.47% in computation time, and the postprocessing algorithms can obtain 1.32% improvement with a 0.27% increase in time.

Table 6.1 and Figure 6.1 indicate how the various optimization techniques contribute to the overall solutions quality.

Circuit	Preprocessing	Iterative Improvement		Postprocessing	Overall
fract	7.20	LS:	31.7	2.5	41.40
		TS:	51.1	3.4	62.30
		SA:	55.1	7.5	69.30
prim1d	3.19	LS:	19.6	1.2	23.99
		TS:	43.2	1.9	48.29
		SA:	53.4	2.5	59.09
struct	8.48	LS:	23.8	0.8	33.08
		TS:	55.5	1.5	65.48
		SA:	63.3	1.9	73.68
ind1	3.53	LS:	35.0	0.6	39.13
		TS:	52.5	0.9	56.93
		SA:	59.7	1.0	64.23
prim2	3.38	LS:	25.5	1.0	29.88
		TS:	52.0	1.5	56.88
		SA:	57.5	1.4	62.28
bio	5.07	LS:	25.5	0.8	31.37
		TS:	53.3	1.4	59.77
		SA:	61.2	1.8	68.07
ind2	3.92	LS:	20.0	0.3	24.22
		TS:	48.8	0.6	54.32
		SA:	55.3	0.7	59.92
ind3	2.44	LS:	22.4	0.2	25.04
		TS:	48.3	0.3	51.04
		SA:	57.0	0.4	59.84
avq_small	4.46	LS:	30.8	0.3	35.56
		TS:	58.2	0.7	63.36
		SA:	66.4	0.9	71.76
avq_large	2.71	LS:	27.0	0.3	30.01
		TS:	59.0	0.7	62.41
		SA:	64.6	0.8	68.11
Average	4.44	LS:	26.13	1.32	31.89
		TS:	52.29		58.05
		SA:	59.35		65.11

Table 6.1: Improvement of Solution Quality Contributed by Various Procedures

6.1 Future Work

In Chapter 3, two placement strategies were implemented. The approach that allows infeasibility brings a large saving of computation time to the placer. However, disadvantages that come with this approach are inevitable: (1) introducing penalties makes the objective function noisy; and (2) legalization of modules in the final phase deteriorates solution quality and therefore obtaining near optimal solutions becomes hindered by this strategy, no matter how efficient the optimiza-

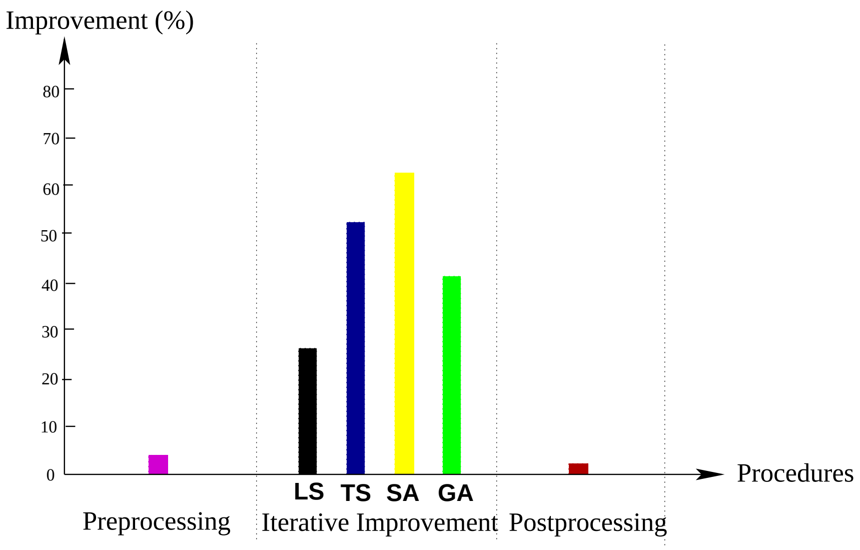


Figure 6.1: Average Improvements Obtained by the Procedures

tion algorithm is. Further investigation of the effect of more efficient techniques for legalization should be studied in more detail.

Besides Genetic Algorithms, there are several meta-heuristics in the VLSI CAD applications, which can produce good quality solutions but demand high computational effort. Parallelizing these algorithms wherever possible may dramatically speed up the VLSI CAD design. In particular within our placement tool, Tabu Search and Simulated Annealing can be good candidates for parallel/distributed processing.

Appendix A

Glossary

CAD	:	Computer Aided Design
CMOS	:	Complementary Metal Oxide Semiconductor
DA	:	Design Automation
FPGA	:	Field Programmable Gate Array
GA	:	Genetic Algorithm
HPWL	:	Half Perimeter Wire Length
LS	:	Local Search
MCNC	:	Microelectronics Center of North Carolina
MIMD	:	Multiple Instruction Multiple Data stream
MISD	:	Multiple Instruction Single Data stream
MPI	:	Message Passing Interface
NP-hard	:	Non Deterministic Polynomial Hard
PVM	:	Private Virtual Machine
RTL	:	Register Transfer Logic

TS	:	Tabu Search
SA	:	Simulated Annealing
SIMD	:	Single Instruction Multiple Data stream
SISD	:	Single Instruction Single Data stream
VLSI	:	Very Large Scale Integration
WL	:	Wire Length

Appendix B

MPI: Message Passing Interface

The Message Passing Interface (MPI) is a specification of a standard library for message passing[1] systems, which was designed by a group of people from academia and industry. Their goal was to develop a standard for writing message-passing programs that would be efficient, flexible and portable on a wide variety of parallel computers. Today, MPI becomes the most widely used parallel paradigm on both network of workstation and massively parallel machines.

The creation of the standard began at a workshop in 1992, and the first outcome was first published in second year. The major versions of MPI highlight the history of this standard:

- Version 1.0 (1994): Fortran77 and C supported;
- Version 1.1 (1995): Minor corrections and clarifications;
- Version 1.2 (1997): Further corrections and clarifications;

[1]Message passing is a programming model that gives the programmer explicit control over interprocess communication.

- Version 2.0 (1997): Major enhancements involving one-sided communications, parallel IO, dynamic process generation and Fortran 90/C++ support.

Although MPI-2 has many powerful new features, it is in many cases not yet fully supported. Consequently the implementation of parallel Island-based GA in this work was developed based on the MPI version 1.2. The basic MPI-1 has 129 subroutines while MPI-2 adds another 193 subroutines. MPI library is large, however with only six different routine calls, serious parallel applications can be programmed. Therefore, programming with a MPI API is easy to start with.

MPI provides support for both the SPMD and MPMD modes of parallel programming, as well as inter-application communications. The communication routines include two classes:

Point-to-point communication routines provide for data exchange between a programmer specified send task and programmer specified receive task. They can be further divided into four groups: blocking, non-blocking, persistent communications, and completion/testing routines.

Collective communication routines provide a convenient means for having all tasks within the communicator participate in a communication operation. Most of them are blocking routines.

In addition, MPI provides abstractions for processes at two levels: (i) Each process has its rank so that processes can identify themselves and perform communications. (ii) Virtual topologies can help to "connect" processes in a convenient and efficient way.

There are several well-tested and efficient implementations of MPI exist. Several of them are free, e.g, MPICH and LAM. Some hardware vendors supply their tuned implementations. With MPI, users can easily develop practical and cost effective parallel applications. Further information about MPI can be found in the following links:

```
The MPI home page at Argonne National Lab:
            ---- http://www-unix.mcs.anl.gov/mpi/
Message Passing Interface (MPI) Forum:
            ---- http://www.mpi-forum.org/
MPI The Complete Reference:
            ---- ftp://ftp.netlib.org/utk/papers/mpi-book/mpi-book.html
MPI - Getting Started with MPI:
            ---- http://www.iu.edu/~rac/hpc/mpi_tutorial/index.html
MPICH - A Portable MPI Implementation:
            ---- http://www-unix.mcs.anl.gov/mpi/mpich/
LAM-MPI Parallel Computing:
            ---- http://www.lam-mpi.org/
```

Bibliography

[Aart90] Emile Aarts and Jan Korst, *Simulated Annealing and Boltzmann Machines*, Wiley, Eindhoven University of Technology, Eindhoven, 1990.

[Acte04] Actel, "Actel Corporation," In *www.actel.coom*, 2004.

[Alba99] Enrique Alba and Jose M. Troya, "An analysis of synchronous and asynchronous parallel distributed genetic algorithms with structured and panmictic islands," In *IPPS/SPDP Workshops*, pp. 248–256, 1999.

[Alte04] Altera, "Altera corporation," In *www.altera.com*, 2004.

[Amda67] G. M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," In *Proceedings of Proc. AFIPS, vol. 30*, pp. 483–485, 1967.

[Andr96] David Andre and John R. Koza, "A parallel implementation of genetic programming that achieves super-linear performance," In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1163–1174, CSREA, Sunnyvale, 9-11 August 1996.

[Arei04] Shawki Areibi and Zheng Yang, "Effective Memetic Algorithms for VLSI Design = Genetic Algorithms + Local Search + MUlti-Level Clustering," 2004.

[Arei93] Shawki Areibi and Anthony Vannelli, "Circuit partitioning using a tabu search approach," In *ISCAS*, pp. 1643–1646, 1993.

[Atal99] Mikhail J. Atallah, *Algorithms and Theory of Computation Handbook*, CRC Press LLC, 1999.

[AY02] Ahmad AL-Yamani, Sadiq M. Sait, and Habib Youssef, "Parallelizing tabu search on a cluster of heterogeneous workstations," *Journal of Heuristics*, vol. 8, pp. 277–304, 2002.

[Bane86] P. Banerjee and M. Joes, "A parallel simulated annealing for standard cell placement on a hypercube computer," In *Proc. Intl. Conf. on Computer-Aided Design*, pp. 34–37, 1986.

[Bane94] Prithviraj Banerjee, *Parallel Algorithms for VLSI Computer-aided Design*, Prentice Hall Publishers, New Jersey, 1994.

[Batt94] Roberto Battiti and Giampietro Tecchiolli, "The Reactive Tabu Search," *ORSA Journal on Computing Vol. 6, N. 2*, pp. 126–140, 1994.

[Blan85] J.P. Blanks, "Near Optimal Quadratic Based Placement for a Class of IC Layout Problems," *IEEE Circuits and Devices*, vol. 1, No. 6, pp. 31–37, September, 1985.

[Brau90] H.C. Braun, "On solving travelling salesman problems by genetic algorithm," In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pp. 129–133, Springer-Verlag, 1990.

[Cai94] Hong Cai, "A gridless routing system for macro-cell design," *Routing, Placement, and Partitioning*, pp. 49–95, 1994.

[Chan96] J. Chandy and P. Banerjee, "Parallel simulated annealing strategies for vlsi cell placement," 1996.

[Chan99] H. Chang, L. Cooks, and M. Hunt, *Surviving the SOC Revolution*, Kluwer Academic Publishers, London, 1999.

[Chen91] C.K. Cheng, S.Z. Yao, and T.C. Hu, "The orientation of modules based on graph decomposition," In *IEEE TRANSACTIONS ON COMPUTERS, VOL. 40, NO. 6*, pp. 774–780, JUNE 1991.

[Chyu83] D.J. Chyuan and M.A. Breuer, "A placement algorithm for array processors," In *Porc. of 20th Design Automation Conf.*, pp. 182–188, 1983.

[Cosn95] Michel Cosnard and Denis Trystram, *Parallel Algorithms and Architectures*, International Thomson Computer Press, Boston, MA, 1995.

[CP00] Erick Cantu-Paz and David E. Goldberg, "Efficient parallel genetic algorithms: Theory and practice," 2000.

[CP98] Erick Cant-Paz, "A survey of parallel genetic algorithms," 1998.

[CP99a] Erick Cantu-Paz, "Migration policies, selection pressure, and parallel evolutionary algorithms," In Scott Brave and Annie S. Wu, editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pp. 65–73, Orlando, Florida, USA, 13 July 1999.

[CP99b] Erick Cantu-Paz, “Topologies, migration rates, and multi-population parallel genetic algorithms,” In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 91–98, Morgan Kaufmann, Orlando, Florida, USA, 13-17 July 1999.

[Damm93] Frank Dammeyer and Stefan Vob, “Dynamic tabu list management using the reverse elimination method,” *Ann. Oper. Res.*, J. C. Baltzer AG, Science Publishers, vol. 41, No. 1-4, pp. 31–46, 1993.

[Dare87] F. Darema, S. Kirkpatrick, and V.A. Norton, “Parallel algorithm for chip placement by simulated annealing,” In *IBM J. Rearch Dev.*, May 1987.

[ea89] S. C. Wong et. al, “A 5000-gate cmos epld with multiple logic and interconnection arrays,” In *Proceedings of 1989 CICC*, pp. 5.8.1–5.8.4, May 1989.

[Emme03] John M. Emmert, Sandeep Lodha, and Dinesh Bhatia, “On using tabu search for design automation of vlsi systems,” In *J. Heuristics 9(1)*, pp. 75–90, 2003.

[Emme99] John M. Emmert and Dinesh Bhatia, “A methodology for fast fpga floorplanning,” In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pp. 47–56, ACM Press, 1999.

[Etaw99] H. Etawil, S. Areibi, and A. Vannelli, “Attractor-Repeller Approach for Global Placement.,” In *Proceedings of IEEE/ACM ICCAD*, pp. 20–24, 1999.

[Fiec94] C.N. Fiechter, “A parallel tabu search algorithm for large travelling salesman problems,” In *Discrete Applied Mathematics*, pp. 243–267, 1994.

[Flyn66] M.J. Flynn, “Very high speed computing systems,” In *IEEE 54, 1901-9*, 1966.

[Fost95] Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[Gare79] M.R. Garey and D.S. Johnson, *Computers and Intractability*, Freeman, San Francisco CA, 1979.

[Glov90] F. Glover, “Tabu search, part i,” *ORSA Journal on Computing*, pp. 190–206, 1990.

[Glov93] Fled Glover, Eric Taillard, and Dominique de Werra, “A user’s guide to tabu search,” In *Annals of Operations Research*, 1993.

[Gold89] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Inc, Reading, Massachusetts, 1989.

[Gord93] V. Gordon and D. Whitley, "Serial and parallel genetic algorithm as function optimizers," In Morgan Kaufmann, editor, *Proceedings of the 5th ICGA*, 1993.

[Grew95] G. Grewal, T. C. Wilson, and D. Stacey, *An Enhanced Genetic Soluiton for Scheduling, Module Allocation, and Binding in VLSI Design*, ANNIE, 1995.

[Grop96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "High-performance, portable implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, No. 6, pp. 789–828, 1996.

[Grov86] L.K. Grover, "A new simulated annealing algorithm for standard cell placement," In *IEEE Intl. Conference on Computer-Aided Design*, pp. 378–380, 1986.

[Gust90] J. Gustafson, "Fixed time, tiered memory, and superlinear speedup," In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, 1990.

[Helm90] D. P. Helmbold and Ch. E. McDowell, "Modeling speedup (n) greater than n," In *IEEE Transactions Parallel and Distributed Systems 1(2)*, pp. 250–256, 1990.

[Holl75] J.H. Holland, *Adaption in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, Michigan, 1975.

[Kiln89] R. Kilng and P. Banerjee, "Esp: Placement by simulated evolution," *IEEE Trans. on Computer Aided Design, 8(3)*, pp. 245–256, 1989.

[Kirk83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[Klin90] Ralph-Michael Kling and Prithviraj Banerjee, "Optimization by simulated evolution with applications to standard cell placement," In *Conference proceedings on 27th ACM/IEEE design automation conference*, pp. 20–25, ACM Press, 1990.

[Koza98] John R. Koza, "Genetic programming," In James G. Williams and Allen Kent, editors, *Encyclopedia of Computer Science and Technology*, pp. 29–43, Marcel-Dekker, 1998.

[Kozm91] K. Kozminski, "Benchmarks for Layout Synthesis - Evolution and Current Status," In *Proceedings of The 28th DAC*, pp. 265–270, IEEE/ACM, Portland, Oregon, 1991.

[Kras00] R. Krashinsky, "Grape: Genetic routing and placement engine," May 2000 Embodied Intelligence term project, Massachusetts Institute of Technology.

[Krav87] S. Kravitz and R. Rutenbar, "Placement by simulated annealing on a multiprocessor," In *IEEE Trans, on CAD, Volume 6*, pp. 534–549, IEEE Computer Society Press, 1987.

[Kuck96] David J. Kuck, *High Performance Computing: Challenges for Future Systems*, Oxford University Press Inc., Kuck and Associates Inc. and University of Illinois, Emeritus, USA, 1996.

[Leon88] D. F. Wong H. W. Leong and C. L. Liu, *Simulated Annealing for VLSI Design*, Kluwer Academic publishers, 1988.

[Lim91] A. Lim, Y.M. Chee, and C.T. Wu, "Performance driven placement with global routing for macro cells," In *Proceedings of Second Great Lakes Symposium on VLSI*, pp. 35–41, 1991.

[Lim96] A. Lim, "Performance driven placement using tabu search," In *Informatica 7(1)*, 1996.

[Lu04] Guangfa Lu and Shawki Areibi, "An Island-Based GA Implementation for VLSI Standard-Cell Placement," In *GECCO 2004*, pp. 1138–1150, 2004.

[Male89] M. Malek, M. Guruswamy, M. Pandya, and H. Owens, "Serial and parallel simulated annealing and tabu search algorithm for travelling salesman problem," In *Annals of Operations Research*, pp. 59–84, 1989.

[Mazu99a] P. Mazumder and E.M. Rudnick, *Genetic Algorithms for VLSI Design, Layout & Test Automation*, Prentice Hall, Toronto, Canada, 1999.

[Mazu99b] Pinaki Mazumder and Elizabeth M. Rudnick, *Genetic Algorithms for VLSI Design, Layout and Test Automation*, Prentice Hall PTR, 1999.

[Mcma98] Matthew T. Mcmahon and Layne T. Watson, "A distributed genetic algorithm with migration for the design of composite laminate structures," 1998.

[Mess04] Message Passing Interface (MPI) Forum, "Mpi: A message-passing interface standard," In *http://www.mpi-forum.org/*, 2004.

[Mitc96] M. Mitchell, *An Introduction to Genetic Algorithms*, The MIT Press, Cambridge, Massachusetts, 1996.

[Mitr86] Mitra and A. Sangiovanni-Vincentelli, "Convergence and finite-time behavior of simulated annealing," In *Advances of Applied Probability, 18*, pp. 747–771, 1986.

[Mune93] Masaharu Munetomo, Yoshiaki Takai, and Yoshiharu Sato, "An efficient migration scheme for subpopulation-based asynchronously parallel genetic algorithms," In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 649, 1993.

[Proj04] The TOP-500 Project, "www.top500.org," 2004.

[PVM 04] PVM Project, "Pvm: Parallel virtual machine," In *http://www.csm.ornl.gov/pvm/*, 2004.

[Raba03] Jan Rabaey, Anantha Chandrakasan, and Borivoje Nikolic, *Digital Integrated Circuits*, Pearson Education Publishing Company, 2003.

[Reev03] Golin R. Reeves and Jonathan E. Rowe, *Genetic algorithms : principles and perspectives : a guide to GA theory*, Kluwer Academic Publishers, School of Mathematical and Information Sciences, Coventry Unversity, USA, 2003.

[Rese02] Intel's Researchers, "Expanding Moore's Law — The Exponential Opportunity," *Intel Internal Journal*, 2002.

[Sait01] Sadiq M. Sait, Habib Youssef, Ahmad Al-Yamani, and Mahmood R. Minhas, "Iterative Heuristics for Multiobjective VLSI Standard Cell Placement," In , pp. , IEEE, Dhahran, Saudi Arabia, 2001.

[Sech85] Carl Sechen and A. Sangiovanni-Vincentelli, "The timberwolf placement and routing package," *IEEE Journal of Solid-State Circuits*, vol. 20, pp. 510–522, April 1985.

[Sech86] C. Sechen and A. Sangiovanni, "The TimberWolf 3.2: A New Standard Cell Placement and Global Routing Package," In *Proceedings of The* 23*rd DAC*, pp. 432–439, IEEE/ACM, Las Vegas, Nevada, June 1986.

[Shah90a] K. Shahookar and P. Mazumder, "Gasp: a genetic algorithm for standard cell placement," In *Proceedings of the conference on European design automation*, pp. 660–664, IEEE Computer Society Press, 1990.

[Shah90b] K. Shahookar and P. Mazumder, "A genetic approach to standard cell placement using metagenetic parameter optimization," *IEEE Trans. on CAD , vol. 9*, pp. 500–511, May 1990.

[Shah91] K. Shahookar and P. Mazumder, "VLSI Cell Placement Techniques," *ACM Computing Surveys*, vol. 23, No. 2, pp. 143–220, 1991.

[SHAR04] SHARCNET, "www.sharcnet.ca," 2004.

[Sher98] Naveed A. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, Intel Corporation, Hillsboro, OR, USA, 1998.

[Song92] L. Song and A. Vannelli, "A vlsi placement method using tabu search technique," In *Micro-electronics Journal*, pp. 167–172, 1992.

[Stev00] Mark A. Stevens, *Merriam Webster's Collegiate Encyclopedia*, Merriam-Webster, 2000.

[Sun94] Wern-Jieh Sun and Carl Sechen, "A loosely coupled parallel algorithm for standard cell placement," In *1994 IEEE/ACM international conference on Computer-aided design*, pp. 137–144, IEEE Computer Society Press, 1994.

[Sun95] Wern-Jieh Sun and Carl Sechen, "Efficient and effective placement for very large circuits," *IEEE Trans. on CAD , vol. 14, No. 3*, pp. 349–359, March 1995.

[Swar95] William Swartz and Carl Sechen, "Timing driven placement for large standard cell circuits," In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pp. 211–215, ACM Press, 1995.

[Tail90] E. Taillard, "Some efficient heuristic methods for the flowshop sequencing problem.," *European Journal of Operational Research*, vol. 417, pp. 65–74, 1990.

[Whit93] D. Whitley, "Cellular genetic algorithms," In Morgan Kaufmann, editor, *Proceedings of the 5th ICGA*, 1993.

[Whit97] Darrell Whitley, Soraya B. Rana, and Robert B. Heckendorn, "Island model genetic algorithms and linearly separable problems," In *Evolutionary Computing, AISB Workshop*, pp. 109–125, 1997.

[Wolf02] Wayne Wolf, *Modern VLSI Design*, Prentice Hall PTR, 2002.

[Xili04] Xilinx, "Xilinx corporation," In *www.xilinx.com*, 2004.

[Yang03] Zheng Yang, "Master's thesis, area/congestion-driven placement for vlsi circuit layout," 2003.