Sadiq M. Sait, Youssef H, Nassar K, et al.Timing driven genetic placement COMPUTER SYSTEMS SCIENCE AND ENGINEERING 14 (1): 3-14 JAN 1999

Timing Driven Genetic Placement

Sadiq M. Sait, H. Youssef, K. W. Nassar, and M. S. T. Benten.
Department of Computer Engineering
King Fahd University of Petroleum and Minerals
Dhahran-31261, Saudi Arabia
Fax: 966(3)-860-2366
Tel: 966(3)-860-2110/2217
e-mail: sadiq@ccse.kfupm.edu.sa

Abstract

In this paper we present a timing-driven placer for standard-cell IC design. The placement algorithm follows the genetic paradigm, with the objective of minimizing both area and path slacks. At early generations, the search is biased toward solutions with superior timing characteristics. As the algorithm starts converging toward generations with acceptable delay properties, the objective is dynamically adjusted toward optimizing area and routability. Experiments with benchmark tests demonstrate delay performance improvement by up to 20%. It is also shown that sizable reduction in runtime is obtained when population size is allowed to decrease in a controlled manner whenever the search hits a plateau. This reduction in runtime is achieved without any noticeable loss in solution quality.

Indexterms:

Combinatorial optimization
Genetic algorithm
Physical design
Standard-cell layout
Timing verification
VLSI
Critical paths
Interconnect delay estimation
Placement
Timing-driven placement
Timing prediction

1 Introduction

In most general terms, placement consists of assigning cells of a given circuit to physical locations on a 2-dimensional layout surface. The placement is performed while optimizing one or several objectives, subject to a given set of constraints. Constraints are imposed by the designer, the implementation process, or layout style. The cells may be standard-cells, macro blocks, etc. In this work we consider standard-cell placement.

Until recently, the total wirelength was a widely used measure of the quality of placement. However, due to advances in VLSI technology, sizes of transistors have been decreasing and their switching speeds increasing. In the last two decades the scaling has been so drastic, that there has been a tremendous increase in the importance of interconnect delays with respect to the overall speed performance of the circuit.

Since the switching time of gates has been lowered (to the order of picoseconds), the clocking speed (timing performance) of VLSI chips has become more and more dependent on signal propagations through the interconnects. As will be illustrated with an example, this phenomenon has resulted in the capacitance of interconnects becoming a key factor in determining the timing performance of the circuits. This reality has been realized by the CAD community. Most modern physical design systems incorporate performance metrics into their objective functions.

1.1 Timing Analysis Concepts

The speed of a circuit is determined by the time it takes for a signal to travel on its *longest* path. A path is a sequence of cells and nets that the signal traverses from a start point (an input pad or the output of a storage element) to an end point (an output pad or the input of a storage element). A signal traveling on any path π is constrained to reach the path end point no later than its latest required arrival time ($LRAT_\pi$). $LRAT_\pi$ is a function of the circuit clock period.

A design is free from long path timing problems if every path is shorter than the latest required arrival time of the signal at the path sink. Referring to Figure 1, there is no long path problem on π iff:

$$T_\pi \leq LRAT_\pi \tag{1}$$

The problem of performance driven placement consists of finding suitable locations of cells so as to minimize the total wirelength while satisfying user specified timing constraints. The problem may also be defined as that of finding a placement that minimizes $T_{\max}$, the delay of the longest path in the circuit (which includes the interconnect delays).

In this work the linear cell delay model is used. The cells for placement are taken from the OASIS standard-cell library [21], where the delay characteristic of a cell is expressed as $BD + LF \times C$; C represents the total loading capacitance in picoFarads, LF the load factor in KiloΩs, and BD the base delay in ns. The total delay seen by the signal entering a gate is defined as follows:

$$TD = BD + LD + ID \tag{2}$$

where,

BD is the base (intrinsic) delay of the cell,
LD is the load delay due to loading pins of the net driven by the cell, and
ID is the interconnect delay on the net.

Using the lumped RC-model, the load delay LD and interconnect delay ID are computed as follows:

$$LD = LF \times C_{in} \tag{3}$$

$$ID = LF \times C_{net} + R_{net} \times (C_{net} + C_{in}) \tag{4}$$

where,

LF is the load factor,
C_{in} is the total input capacitance of the loading cells,
R_{net} is the total interconnect resistance of the net, and
C_{net} is the total interconnect (fringe plus surface) capacitance of the net.

Example

Consider the two point net shown in Figure 2. For a two-input NAND gate from the OASIS standard-cell library, the delay characteristics are $BD = 0.58\ ns$ and $LF = 3.53\ Kilo\Omega s$ (ns per picoFarads). The capacitance at the input pin of a similar gate is $C_{in} = 0.08$ picoFarads. Ignoring the delays due to interconnects, the delay seen by the signal, that is, base delay plus the load delay, is equal to $0.58 + 3.53 \times 0.08 = 0.864\ ns$.

Now let us determine the delay seen by the signal due to the effect of interconnect capacitance alone. Assume that the horizontal and vertical separation of the two points of the net are 1000μ each and the width of the interconnect is 3μ. Also assume that the vertical connection is made with metal-1, and horizontal with metal-2. The interconnect capacitance comprises two components, the plate capacitance and fringe capacitance. The values of sheet resistance and capacitance provided by Orbit Semiconductor (fabrication company for MOSIS Designs) [13] are,

$R_{m1} = 0.060$=Resistance of metal-1 in Ωs per square,
$R_{m2} = 0.033$=Resistance of metal-2 in Ωs per square,
$C_{ma1} = 0.26 \times 10^{-4}$=plate capacitance of metal-1 in picoFarads per μ^2,
$C_{ma2} = 0.15 \times 10^{-4}$=plate capacitance of metal-2 in picoFarads per μ^2,
$C_{mf1} = 0.82 \times 10^{-4}$=fringe capacitance of metal-1 in picoFarads per μ length, and
$C_{mf2} = 0.85 \times 10^{-4}$=fringe capacitance of metal-2 in picoFarads per μ length.

Using these values we obtain the following:

$$R_{net} = R_{m1} + R_{m2} = (\frac{1000}{3})(0.06) + (\frac{1000}{3})(0.033) = 0.033\ Kilo\Omega s \tag{5}$$

$$C_{m1} = C_{area-m1} + C_{fringe-m1} = 0.242492\ pF \quad (6)$$

$$C_{m2} = C_{area-m2} + C_{fringe-m2} = 0.215510\ pF \quad (7)$$

$$C_{net} = C_{m1} + C_{m2} = 0.458\ pF \quad (8)$$

ID, the contribution of delay due to the two point interconnect alone is therefore,

$$ID = LF \times C_{net} + R_{net} \times (C_{net} + C_{in}) = 1.6167 + 0.01775 = 1.64\ ns \quad (9)$$

This value of ID (1.64 ns) is comparable to the delay of the gate, that is, the delay seen by the signal when the interconnect delays are ignored (0.864 ns). These values are for SCMOS 2μ technology. For 0.8μ technology, the ratio of interconnect delay to cell delay is larger, and with continued scaling, this factor is expected to increase in the future. Observe that in Equation 9 the contribution of the delay due to R_{net} is negligible compared to that of delay due to capacitance. Therefore, the delay due to the interconnect is dominated by its capacitance.

1.2 Approaches to Improve Performance

During the design process, the propagation delays on the interconnects are not known prior to layout. Long path timing problems registered after layout are very difficult to correct because they may require, not only new iterations of the physical design step, but possibly, many iterations of the logic design step.

Three general approaches have been suggested to correct long path timing problems. The first approach proceeds by making changes to the logic. For example, the delay of a path can be substantially decreased by reducing the loading on some of its circuits elements. Also collapsing some of the logic on the long paths can reduce some of the paths' delays. Representative implementations of this approach are reported in [4, 6, 16, 22, 28].

The second approach relies on transistor re-sizing to speed up some of the circuit elements on the slow paths. By increasing the sizes of some of the driving transistors, the switching delays of the driving elements as well as the propagation delays along the nets driven by the re-sized transistors can be substantially lowered. Examples of implementations using this approach have been described in [2, 4, 15, 22, 25].

The third approach to make a circuit faster without making any changes in its logic design is to reduce the propagation delay due to interconnects to a minimum or well within tolerable limits. This goal can be achieved by imposing timing constraints on the interconnects and paths of the design. That is, wiring delays must be kept in check so that T_π is kept below a maximum value. Since placement affects the wiring requirements of a layout, the objective of the placement problem is altered to drive T_π below $LRAT_\pi$. This approach has been adopted in our work.

Optimized for timing, a layout system can permit as much as 20% increase in the clock rate without any changes in the logic or cell design. Numerous attempts have been reported which tried to make the physical design sensitive to the timing requirements. Work in this direction was initiated as early as the 1970's [25] where ideas from physics were employed to optimize the power and timing of LSI chips. In [8], a system was

reported in which a circuit which is completely laid out is simulated on the computer to determine long paths that violated constraints. The layout is then adjusted and simulated again. In [3], performance driven circuit partitioning heuristics that resulted in performance improvements with little loss in wirability were reported.

Methods for generating constraints on sizes of nets to guarantee performance are reported in [11, 24, 34]. These methods consist of distributing slacks on the nets. The final layout that satisfied these net bounds was guaranteed to satisfy path timing constraints for desired performance. In [33], factors which are highly correlated with path timing such as number of nets on the path, path slacks, driving strengths of cell output pins, etc., are combined into a score function. Path criticality is decided on the basis of path scores. The predicted critical paths are used by the placement procedure.

In [14], a linear-programming approach has been attempted to reduce the estimate of cycle time. Timing driven placement that uses the idea of rectilinear distance facility location and partitioning to minimize wire delays to satisfy timing constraints were proposed in [27]. A constructive placement method to sequentially place cells using a cost function that captures the timing behavior was presented in [19]. In [29], the problem of performance driven physical design is formulated as a constrained programming problem. Constraints are placed on total path delays including cell and interconnect delays and the behavior of the paths is captured. Mathematical techniques and heuristics based on Lagrangian relaxation are used to find an approximate solution to the constrained problem. In [20], a description of a placement system based on the fuzzy logic approach is presented. A solution that satisfies multiple objectives, which are, area, routability, and timing, is produced using fuzzy logic rules. In [31], the application of constructive successive augmentation methodology to VLSI placement under constraints on routability, area and timing is presented. The placement algorithm uses adaptive look-ahead procedures to improve the effectiveness of constructive decision making. The methodology was successfully implemented for macro-cell-library-based sea-of-gates design style with over the cell routing.

Iterative and non-deterministic techniques have also been employed. In [7], the authors employ simulated annealing [32] to improve both the wirelength and performance.

Other iterative nondeterministic techniques that have been applied to the placement are genetic algorithm (GA) [5, 26] and simulated evolution [18]. But for both, the placement objective was the minimization of wirelength, timing performance was not an issue.

1.3 Justification of Iterative Improvement Approach

With the continuing increase in circuit complexity and performance, physical design is becoming increasingly harder. Constructive performance driven procedures have the advantage of being faster than iterative procedures such as simulated annealing or genetic algorithm. However, at each decision step, due to its greedy nature, a constructive procedure has only a local view. Therefore, the procedure might reach the point where design constraints are not met. This will require several iterations to attempt various modifications to the solution to bring it to a feasible state. For VLSI designs with tens

of thousands of gates, it is unthinkable to perform this modification manually.

We believe that automatic iterative improvement procedures which combine quality of constructive algorithms and iterative improvement procedures constitute the most reasonable approach to produce feasible solutions with the desired performance. However, in order to speed up the search, care must be taken so that the iterative procedure be tuned to quickly converge to a solution satisfying all design constraints.

In this work such a strategy is adopted. Initially, a number of placement configurations are constructively produced. Then, the genetic algorithm is used to iteratively search for a new solution that combines the good characteristics of the initial configurations. The overall objective is two-fold; (1) satisfy path timing constraints and (2) minimize overall wiring length (area).

The principle reason for selecting the genetic algorithm is that it allows several placement configurations to be maintained. This makes the technique easy to adapt to the multi-objective nature of the placement problem in general and timing-driven placement in particular. Further, the chances of getting trapped in a local minima are reduced since several alternatives exist.

Central to the success of performance driven placement program is the accurate prediction of path timing constraints.

1.4 Timing Prediction for Placement

Long path timing problems are caused by large interconnect delays. Obviously, the critical paths are those that are most inclined to exhibit a long path problem. Large interconnect delays are mostly due to placement, and to a lesser extent, to routing.

Two general approaches have been suggested to make the placement sensitive to the design timing requirements: (1) net oriented approach, and (2) path oriented approach. For the net oriented approach, a timing analyzer is used to assign weights to every net based on the paths slacks. The weights could be length bounds [11, 34, 35], or a level of criticality [9]. These weights are used to guide the placer to produce a timing correct placement. For the path oriented approach the timing analyzer is used to predict the most critical paths. These paths are supplied to the following placement step. *Because timing constraints are path oriented in nature, this second approach is superior.* However, the number of paths in a given design is exponential, but, fortunately only a small subset of these paths are timing critical. Hence a pre-requisite for the success of this approach is the accurate prediction of the critical paths prior to placement. A number of strategies used to predict the critical paths are described in [35].

In this work, we propose a new approach to predict the critical paths, which is a variation of one of the approaches reported in [35].

Prediction of the critical paths

From past layouts of circuits with similar complexity, the average and standard deviation of net length are estimated for each type of net (2 pin-, 3 pin-,..., k pin-nets). These are converted to capacitances for the particular technology of the design at hand.

To simplify the explanation, we shall assume that all cells are single output cells. Let $\pi = [c_1, c_2, \cdots, c_i, \cdots, c_{k+1}]$ be a particular path from cell c_1 to cell c_{k+1}. Under the assumption that the nets are statistically independent, the expected delay on path π can be expressed as follows,

$$T_\pi = \sum_{i=1}^{k} (BD_i + LD_i + LF_i \cdot \overline{x_i}) \tag{10}$$

where,

BD_i is the base (intrinsic) delay of cell c_i,
LD_i is the load delay due to the loading pins of the net driven by cell c_i,
LF_i is the load factor of the output pin of cell c_i, and
$\overline{x_i}$ is the expected interconnect capacitance of the net driven by cell c_i, estimated from past designs.

Similarly, the delay variance on path π can also be computed as follows,

$$S_\pi^2 = \sum_{i=1}^{k} (LF_i^2 \cdot S_i^2) \tag{11}$$

where, S_i^2 is the variance of the interconnect capacitance of net i as estimated from past designs.

Let $T_{\max}$ be the estimated delay of the longest path in the design, that is,

$$T_{\max} = \max_{\pi} \{T_\pi\} \tag{12}$$

A path π is called *α-critical* if and only if,

$$T_\pi + \alpha \times S_\pi \geq T_{\max} \tag{13}$$

The parameter α acts as a confidence level. The larger α is, the larger is the number of predicted critical paths, and the higher is the probability of including all potentially critical paths. Reasonable values of $\alpha \times S_\pi$ are 2 to 3 ns.

The algorithm used to enumerate all paths which satisfy Equation 13 is a variation of the algorithm reported in [1]. The prediction strategy described was very effective in reporting all the critical paths prior to placement.[1] Nevertheless, one has to be careful as to when such a strategy is possible. In our case, the variances on net lengths were relatively much smaller than the overall path interconnect length. However, it may happen that for design styles other than the standard-cell design style (used in this work) and very dense and large designs, the net length variances would be unacceptably large. For example, such a phenomenon has been observed in [35] for very dense CMOS sea-of-gates chips.

[1]This approach is being validated on several test cases to determine its success ratio. It is important to have a success ratio very near 100%.

2 TDGAP: A Timing Driven Genetic Algorithm for Placement

Genetic algorithms (GA) is a search technique which emulates the natural process of evolution as a means of progressing toward the optimum [12, 10]. GA has been applied in solving various optimization problems including those in VLSI physical design [5, 26].

In GA approach, at any given instance, a number of possible solutions called *population*, exist. Their number, denoted by N_p, depends on the problem instance, size of the problem, and the available memory. Each individual in the population is a string of symbols. These symbols are known as *genes* and the string made up of genes is termed *chromosome*. The chromosome represents a possible solution to the optimization problem. During each iteration (*generation*), the individuals in the current population are evaluated using some measure of fitness. Based on the *fitness* value, two individuals at a time (called *parents*) are *selected* from the population. Individuals with high fitness are more likely to be selected. Then genetic operators are applied on the selected parents to generate new individuals (solutions) called *offsprings*. These genetic operators combine features from both parents. The two principal genetic operators are *crossover* and *mutation*. They are derived by analogy from the biological process of evolution. The structure of the genetic algorithm for timing-driven placement referred to as TDGAP is given in Figure 3.

It is the first time that GA is used for timing (performance) driven placement. All previous reported applications of GA for placement have used wirelength as an objective function. In this work the cost function used includes both the timing performance of the circuit and the total wirelength. Standard-cell layout style is assumed, however, the same work can be applied to gate-array design style with minor modifications.

TDGAP strives to satisfy the following constraints and objectives:

1. *Timing constraints*, in which the delays of the critical paths should be smaller than a certain predefined value as reported by the timing analysis program.
2. *Geometric fit*, in which cells are placed at legitimate locations within the circuit boundary without overlapping. The circuit boundary or the shape of the layout can be controlled via an aspect ratio variable, which is user specified.
3. *Routability*, where sufficient routing space is left to properly interconnect the cells without affecting the performance of the circuit. This is done by using a procedure to estimate the density of the routing channels (using the vertical constraint graph).
4. *Total wirelength*, where a relative weight has been given for the total wirelength of the layout. This is to give a chance to solutions that are good in terms of total wirelength but within a small margin from good solutions with respect to timing.

2.1 Approach Overview

Given an initial solution, TDGAP conducts a search in the solution space using its operators and functions to repetitively modify the population of placement configurations. The process of modification and search is repeated until all the timing constraints or

some terminating criteria are met. Timing constraints consist of delay bounds on the critical paths of the circuit.

The implementation of TDGAP follows the general structure of the genetic algorithm outlined in Figure 3. In TDGAP, the selection of the next generation is done on the set of the offsprings (without mutation) and the parents. Then, the mutation operation is applied on the new constructed generation. Our experiments strongly indicate that performing mutation after selection is superior to the case when mutation is performed before selection.

First we shall see how a solution can be mapped to a chromosome. Then we will discuss how to tune the genetic operators to search for an optimum placement that satisfies path timing constraints with reduced wirelength (area).

2.2 Solution Representation

Each individual (solution) in the population is encoded (represented) as a set of rows. Each row contains modules (genes) that are represented as a set of three integers indicating the cell serial number, the row number, and the displacement from the left edge of the layout. The encoding scheme for n rows solution is shown in Figure 4.

2.3 Initial Population Constructors

Initial solution construction is very critical to GA. Five initial population constructors IPC_1, IPC_2, IPC_3, IPC_4 and IPC_5 were investigated. All of these constructors allow the generation of an initial population of any desired size. The five constructors are itemized below.

- Constructor IPC_1 selects modules at random and places them in rows. The quality of the generated solutions using this constructor are unpredictable.
- Constructor IPC_2 attempts to cluster cells (modules) affecting the same path to improve the score of the initial population. Based on the linear delay model used in this work, the cells affecting the same path are defined to be all cells on the nets that are on that path (see Figure 5). It is known that when the solutions in the initial population are all of good quality, GA tends to get quickly trapped into a local minima. Our experimental results confirm this observation.
- Constructor IPC_3 is similar to constructor IPC_2. The difference between IPC_2 and IPC_3 is in the way they start placing the cells on the layout surface. For a layout of n rows, IPC_2 places cells, left to right, starting from row 0, while for IPC_3 cells are placed starting from the middle row and proceeding outward.
- Constructor IPC_4 combines IPC_1 and IPC_3. This constructor exhibits superior average fitness value to that of IPC_1 but lower than that of IPC_3. This constructor generates initial solutions with a wide fitness spectrum. This helps avoid getting trapped in local minima.
- Constructor IPC_5 is similar to constructor IPC_4 with one difference. IPC_5 includes in its initial population a placement configuration obtained using the mincut

i	$cost(\mathcal{P}(i))$	$exp\_count(\mathcal{P}(i))$	sure copies in L	probabilistic copies in L
1	37	0.78	0	0.78
2	52	1.10	1	0.10
3	60	1.27	1	0.27
4	21	0.45	0	0.45
5	98	2.08	2	0.08
6	15	0.32	0	0.32

Table 1: An example to show how the expected count of individuals in a certain generation is computed: $\overline{cost}$=47.17, $N_p = 6$.

partitioning algorithm. Algorithms based on mincut partitioning are well known for generating layouts with minimum chip area. Therefore, a solution based on this algorithm is included in the initial population. TDGAP uses the OASIS mincut placement program to obtain this solution. For the OASIS placer, the circuit netlist is recursively partitioned into quadrants until the partition contains one cell per quadrant [21, 30].

Therefore, initial populations constructed with IPC_5 consist of three classes of individuals: (1) placements obtained randomly, (2) placements constructively built to exhibit good timing characteristics, and (3) placements constructively obtained with mincut partitioning.

2.4 Choice Function

The choice function adopted is based on the *stochastic remainder without replacement* scheme. This selection scheme has been proven to be superior over the expected value scheme [10].

The implementation of the choice function involves two procedures. First, a preselect procedure prepares a list L of candidate parents. Then a second procedure, called parent selection, selects parents from the list L for crossover (mating).

Let $exp\_count(\mathcal{P}(i))$ be the value of the expected count of an individual $\mathcal{P}(i)$. This value is computed as follows:

$$exp\_count(\mathcal{P}(i)) = \frac{cost(\mathcal{P}(i))}{\overline{cost}} \tag{14}$$

where,

$$cost\mathcal{P}(i) = \text{ cost value of individual } \mathcal{P}(i) \tag{15}$$

and,

$$\overline{cost} = \frac{1}{N_p} \times \sum_{i=1}^{N_p} cost(\mathcal{P}(i)) \tag{16}$$

Stochastic remainder without replacement works as follows. For each individual $\mathcal{P}(i)$, $\lfloor exp\_count(\mathcal{P}(i)) \rfloor$ instances of $\mathcal{P}(i)$ are included in the list L. The fractional part $f_i = exp\_count(\mathcal{P}(i) - \lfloor exp\_count(\mathcal{P}(i)) \rfloor$ is interpreted as a probability, that is, with probability f_i one more instance of $\mathcal{P}(i)$ is included in the list L. This operation is repeated until all individuals are processed.

Table 1 illustrates this scheme for a population with six individuals. From this table, $exp\_count(\mathcal{P}(5)) = 2.08$ which means that we have to assign two copies of individual $\mathcal{P}(5)$ in the list L. Since $f_5 = 0.08$, then one additional copy of $\mathcal{P}(5)$ is included in L with probability 0.08.

After the pre-select procedure, the parent selection procedure operates on the generated list L. Each time a parent is needed for a crossover operation, the parent selection procedure randomly selects an individual from the list L. In case we have two identical parents, we run the selection again so as to get two different parents for the crossover operation. The selected parents are removed from the list.

2.5 Crossover $\mathcal{X}$

Crossover is the most important genetic operator and has the most effect on the convergence rate and the quality of solution. For the two parents selected by the choice function, crossover is applied to generate an offspring. Two types of crossover operators $\mathcal{X}_1$ and $\mathcal{X}_2$ are considered in TDGAP. Both use information passing from one parent to the other, but they differ in the way they pass information. Both operators are aimed at improving the timing aspects of the reported *α-critical* paths. They try to pass the information about some of satisfied paths (paths with no timing problems) from one generation to another. Operator $\mathcal{X}_1$ does this by maintaining the same locations of the cells affecting these satisfied paths. Operator $\mathcal{X}_2$, however, does it by keeping the cells affecting these satisfied paths within a certain window.

Let $\mathcal{P}(s)$ and $\mathcal{P}(t)$ be the passing and target parents respectively, and CP be the set of the *α-critical* paths of the circuit. Operator $\mathcal{X}_1$ operates in the following way. An identical copy ($\mathcal{P}(o)$) of $\mathcal{P}(t)$ is made. A critical path *cp* is selected from CP according to a criterion that will be explained later. Then, the set β of cells affecting *cp* are identified. The goal of $\mathcal{X}_1$ is to reconfigure offspring $\mathcal{P}(o)$ such that the cells in β occupy the same locations as in the passing parent. This operation may overwrite some of the modules in $\mathcal{P}(o)$. Collisions created by this operation are resolved as follows. Suppose a module $e_i \in \beta$ is to be assigned to location loc_j which is occupied by module e_j. This conflict is resolved by interchanging the locations of the two modules. For example, e_i is assigned to loc_j and e_j to loc_i where loc_i is the location of e_i in $\mathcal{P}(s)$ and loc_j is location of e_j in $\mathcal{P}(t)$. The steps of $\mathcal{X}_1$ operation are illustrated in Figure 6. In this figure, the set $\beta = \{c_1, c_2, c_3, c_4\}$. Then, the cells of β in $\mathcal{P}(o)$ are moved to the same location as they were in $\mathcal{P}(s)$. As shown in Figure 6(c), cell c_1 is moved to the location of cell d_1. To avoid overwriting cell d_1, cells c_1 and d_1 interchange locations.

Crossover operator $\mathcal{X}_2$ operates in the following manner. It starts like $\mathcal{X}_1$ by making a copy $\mathcal{P}(o)$ from $\mathcal{P}(t)$, selecting a critical path *cp* from CP, and identifying the set β of cells affecting *cp*. The size and location of the smallest bounding window ω_x that

encloses the cells of β in $\mathcal{P}(s)$ is determined. A window ω_y is also determined in parent $\mathcal{P}(t)$ with the same dimensions and location as ω_x in $\mathcal{P}(s)$. Comparing the contents of these two windows, three sets are defined:

σ = cells $\in \beta - \omega_y$, (cells in β but not in ω_y);
ρ = cells $\in \omega_x - (\omega_y + \beta)$, (cells $\in \omega_x$ but are neither in ω_y nor in β); and
η = cells $\in (\omega_x \cap \omega_y)$.

Then the operator $\mathcal{X}_2$ tries to reconfigure $\mathcal{P}(o)$ as follows. It first defines a window ω_o in $\mathcal{P}(o)$ of the same size and location of ω_x. After that, it scans the contents of ω_o cell by cell and one row at a time. Then, for each scanned cell e_i, operator $\mathcal{X}_2$ works according to the algorithm shown in Figure 7. A graphical representation of the operation of $\mathcal{X}_2$ is depicted in Figure 8.

Selection of critical path *cp*

For each placement configuration $\mathcal{P}(i)$ in current generation, the critical paths are maintained in two lists: (1) CP_i^+ is the list of the critical paths with positive slack, and (2) CP_i^- is the list of the critical paths with negative slack. Let $\mathcal{P}(s)$ and $\mathcal{P}(t)$ be the source (passing) and target parents respectively. The process of selecting a *cp* is aimed at generating an offspring with better timing characteristics. The critical path selection algorithm proceeds as follows:

1. If target parent has no timing problems, that is $CP_t^- = \emptyset$, then select at random a *cp* from CP_s^+;
2. else, if there is a critical path *cp* with long path timing problem in target parent, but is problem free in passing parent, that is $cp \in CP_s^+ \cap CP_t^-$, then select this path;
3. else, if there is a *cp* with a long path timing problem in both the target and passing parents ($cp \in CP_s^- \cap CP_t^-$), but with better timing (larger slack) in the passing parent, then select this path;
4. else, select a *cp* at random.

The path selection algorithm is outlined in Figure 9.

2.6 Selection (§) of the Next Generation

The crossover applications have the effect of augmenting the current generation with their offsprings. In order to maintain the population size fixed, we need a systematic rule to determine which individuals to use as part of the next generation. We experimented with four selector functions $\S_1, \S_2, \S_3$, and $\S_4$. Let $\mathcal{P}$ be the current population and $\mathcal{J} = \mathcal{P} \cup Offspring$; then the selector functions proceed as follows:

- Selector $\S_1$ selects from $\mathcal{J}$ the best scoring individual and $N_p - 1$ other individuals at random, where $N_p = \mid \mathcal{P} \mid$. Hence, the best individual always survives to the next generation.

- Selector $\S_2$ selects the best 10% of N_p and the rest are selected at random. With this selector, a reasonable number of good placement configurations survive for next generation. This may lead to the undesirable situation where most of the individuals in some later generation are all alike, which may cause the search to be trapped into local minima.
- Selector $\S_3$ selects all N_p individuals from $\mathcal{J}$ at random. With this selector, the genetic algorithm will most likely require a large amount of time before converging to an acceptable solution.
- Selector $\S_4$ selects individuals on a competitive basis with each individual $\mathcal{P}(j)$ having a probability $Prob(j)$ to be selected, where

$$Prob(j) = \frac{score(\mathcal{P}(j))}{\sum_{i=1}^{q} score(\mathcal{P}(i))} \tag{17}$$

where, $q = \mid \mathcal{J} \mid$, and $score(\mathcal{P}(i))$ is the fitness of individual $\mathcal{P}(i)$. With this selector, the algorithm has a higher probability than with other selectors to be trapped into local minima. This might happen because individuals with low fitness values are quickly discarded at the early generations.

2.7 Mutation μ

Two mutation operators μ_1 and μ_2 are investigated. Operator μ_1 is targeted toward improving the timing of the placement, while operator μ_2 is targeted at improving the wirelength of the placement. Both operators use the notion of *center of net* in order to improve the path timing and length of selected nets. For a given net i with m modules, the center of net i is defined as a pair (x_i^*, y_i^*), where,

$$x_i^* = \frac{1}{m} \cdot \sum_{k=1}^{m} x_k \tag{18}$$

$$y_i^* = \frac{1}{m} \cdot \sum_{k=1}^{m} y_k \tag{19}$$

In TDGAP, except for the best individual, any individual in the newly selected generation may be a candidate for mutation. This is with the intention of not loosing the best individual. Loosing the best individual can weaken the population and cause a slower convergence. Experimental results in support of this statement are provided in Section 3.

Mutation operator μ_1 works as follows. For each individual $\mathcal{P}(i)$ that is selected for mutation, first a critical path cp is randomly selected from the set CP_i^- (those paths with long path problems in $\mathcal{P}(i)$). Next, a module e_s that is affecting the performance of the selected critical path is selected randomly. Let Net_s be the net driven by module e_s, and e_j be the module located at the center of net Net_s. Module e_j may or may not belong to net Net_s. The two modules e_j and e_s are pairwise-interchanged, thus improving the *slack* value of cp. This mutation operation is illustrated in Figure 10.

Mutation operator μ_2 operates in a manner similar to that of operator μ_1. It starts by selecting at random a two-pin net, where neither of the two pins is an I/O pad. Then, one of the two modules is chosen to be swapped with a module at the location of the center of the selected net. The operation of μ_2 is illustrated in Figure 11.

The requirement that the selected net be a two-pin net is motivated by experimental observations. Analysis of several layouts revealed that most of the two-pin nets that are on critical paths have their modules separated by large distances. This wide separation has two undesirable effects: (1) it increases the number of feedthroughs and (2) it increases the total wirelength. These effects are illustrated in Figure 11. Furthermore, the reason for requiring that the selected net should have neither of its pins as an I/O pad is that, if a two-pin net has one of its pins as an I/O pad, then the other pin is connected to a cell, say e_i, which is connected to other nets. Moving cell e_i tends to disturb the solution much more than moving the I/O pin between the top and bottom sides of the layout. Due to this reason, we have included a separate procedure that takes care of assigning the I/O pads to either the top or bottom sides of the layout, so as to improve the placement solution. In TDGAP, operator μ_2 is applied on 10% of the total number of nets.

2.8 Score Function

The score function (objective function) is a combination of three terms. The three terms are directed toward the improvement of the circuit performance and total wirelength. The score function is a way to determine which individual is fitter than others. Let $\mathcal{P}(i)$ and $\mathcal{P}(j)$ be two individuals in current population $\mathcal{P}$; if $Score(\mathcal{P}(i)) > Score(\mathcal{P}(j))$ then individual $\mathcal{P}(i)$ is fitter than individual $\mathcal{P}(j)$. Let N_p be the population size and CP be the set of critical paths. The score of a given individual $\mathcal{P}(i)$ is computed as follows:

$$Score(\mathcal{P}(i)) = w_1 \times \frac{worst\_slack(\mathcal{P}(i))}{S} + w_2 \times \frac{relax(\mathcal{P}(i))}{R} + w_3 \times \frac{wirelength(\mathcal{P}(i))}{W} \tag{20}$$

where,

$$worst\_slack(\mathcal{P}(i)) = \text{ } -\text{Slack value of the worst critical path} \tag{21}$$

$$S = \max_{j=1\cdots N_p} worst\_slack(\mathcal{P}(j)) \tag{22}$$

$$relax(\mathcal{P}(i)) = \sum_{\pi \in CP_i^\dagger} slack_\pi \tag{23}$$

$$R = \max_{j=1\cdots N_p} relax(\mathcal{P}(j)) \tag{24}$$

$$wirelength(\mathcal{P}(i)) = \text{the total wirelength of individual } \mathcal{P}(i) \tag{25}$$

$$W = \max_{j=1\cdots N_p} wirelength(\mathcal{P}(j)) \tag{26}$$

w_1, w_2, and w_3 are different weights assigned for each term.

The values of $worst\_slack(\mathcal{P}(i))$, $relax(\mathcal{P}(i))$, and $wirelength(\mathcal{P}(i))$ are used after they are linearly scaled. Originally, these three functions have different domains of definition, and thus, they are not comparable. In order to make them comparable and combine them into one equation, we have to make their values fall within the same range. This is achieved through linear scaling. Another reason necessitating linear scaling is that some of the values taken by these functions are either very near each other or very far apart. By scaling these values we normalize their variances. For example, $wirelength(\mathcal{P}(i))$ is linearly scaled as follows:

$$wirelength(\mathcal{P}(i)) = New\_AVG - (\frac{wirelength(\mathcal{P}(i)) - Old\_AVG}{Old\_STD} \times New\_STD)$$

where,

$Old\_AVG = \frac{1}{N_p} \times \sum_{j=1}^{N_p} wirelength(\mathcal{P}(j))$,

Old_STD = sample standard deviation = $\sqrt{\frac{1}{N_p - 1} \sum_{j=1}^{N_p} (wirelength(\mathcal{P}(j)) - Old\_AVG)^2}$,

New_AVG= the new average after linear scaling, and
New_STD= the new standard deviation after linear scaling.

As an example, Table 2 shows the steps of scaling the values of $worst\_slack(\mathcal{P}(i))$, $relax(\mathcal{P}(i))$, and $wirelength(\mathcal{P}(i))$ for each of the six individuals in a given population. From this table, the values of S, R, and W are:

S=61.85
R=59.85
W=64.70

then, the score value of individual 5, $Score(\mathcal{P}(5))$, is computed as follows: $Score(\mathcal{P}(5)) = w_1 \times \frac{51.17}{61.85} + w_2 \times \frac{59.85}{59.85} + w_3 \times \frac{47.90}{64.70}$. Let $w_1 = 0.70$, $w_2 = 0.05$, and $w_3 = 0.25$; then, $Score(\mathcal{P}(5)) = 0.83 \times 0.70 + 1.0 \times 0.05 + 0.74 \times 0.25 = 0.816$

For a given individual $\mathcal{P}(i)$, $relax(\mathcal{P}(i))$ is the summation of the $SLACK$ values of all satisfied critical paths (paths with no long path problems). This value is a measure of the amount by which these paths can be made longer and still be satisfied. Making some of these paths longer may give other unsatisfied critical paths (paths with long path problems) the chance to become shorter.

The values of w_1, w_2, and w_3 affect TDGAP solution quality. For example, if $w_1 > w_3$ then TDGAP favors solutions that have superior timing characteristics over those which have smaller overall wirelength. The relative weights used in TDGAP are, $w_1 = 0.70$, $w_2 = 0.05$, and $w_3 = 0.25$.

	Before Scaling			*After Scaling*		
$\mathcal{P}(i)$	$worst\_slack(\mathcal{P}(i))$	$relax(\mathcal{P}(i))$	$wirelength(\mathcal{P}(i))$	$worst\_slack(\mathcal{P}(i))$	$relax(\mathcal{P}(i))$	$wirelength(\mathcal{P}(i))$
1	10	3	100	33.38	57.66	54.20
2	3	10	200	58.29	42.34	33.20
3	2	15	150	61.85	31.40	43.70
4	4	4	90	54.73	55.47	56.30
5	5	2	130	51.17	59.85	47.90
6	8	5	50	40.50	53.28	64.70
AVG	5.33	6.5	120	50	50	50
STD	2.81	4.57	47.61	10	10	10
MAX	10	15	200	61.85	59.85	64.70

Table 2: Example illustrating linear scaling the values of $worst\_slack(\mathcal{P}(i))$, $relax(\mathcal{P}(i))$, and $wirelength(\mathcal{P}(i))$ in certain population.

3 Choice of Genetic Control Parameters

The performance of a genetic algorithm depends on several essential parameters. These consist of the following:

(i) Initial population – How should the individuals of the initial population be selected?

(ii) Size of the population – How many individuals should be maintained by the algorithm and whether the population size is kept constant or gradually decreasing.

(iii) Scoring function – How should the individuals be rated? This is dictated by the problem instance and objectives.

(iv) Formation of new generation – This requires answers to the following questions: (a) How are parents selected for mating (crossover)? (b) What crossover operation to use? (c) How should the individuals of the next generation be selected from the offsprings and the individuals of current generation? (d) And when should individuals be allowed to mutate and at what rate?

(v) Stopping criterion – Is the stopping criterion function of solution quality, the run time, the number of generations, or a combination of these?

What items (i)-to-(v) indicate is that the genetic algorithm is a very elaborate and relatively hard to tune algorithm. It is harder than other iterative heuristics such as simulated annealing [17] and simulated evolution [18]. The tuning of the genetic algorithm parameters to a particular problem (such as placement) requires extensive experimentation. Such an extensive experimentation has been conducted on TDGAP. The objectives of this experimentation are threefold:

(1) To tune TDGAP parameters as indicated above;

(2) to measure how much timing improvement would be possible without noticeable loss in area and routability; and

(3) to measure the scale of improvement in overall solution quality (timing, area, and routability) with that obtained by the popular mincut constructive heuristic.

We measured the effects of the various genetic control parameters on the area and timing of the circuit. In the remaining text we summarize the outcome of these experiments.

3.1 Mutation Prior To Or After Selection?

Since the mutation probability is low and uniformly distributed, it is intuitive to think that the application of mutation before or after selection will not lead to significantly different outcomes. However, experiments indicate that performing mutation after selection leads to sizable improvement with respect to both timing and wirelength. Figure 12 compares the performance of these two strategies with respect to timing and wirelength metrics.

The superior performance of applying mutation after selection can be explained as follows. Although the mutation operation is applied with a small probability, it can not be ignored. It is the process that helps the genetic algorithm escape from local minima. Performing mutation before selection reduces the effect of the mutation operation. The reason is that mutated solutions with inferior scores are less likely to be selected and be part of next generation. This has the effect of weakening the role of the mutation operation as a way to escape from local minima. Hence, performing mutation prior to selection will increase the likelihood that the search procedure gets trapped into local minima. In contrast, the application of mutation after selection allows the effects of the mutation operation to be inherited by next generation. This way, the algorithm is less likely to be trapped into local minima.

3.2 Tuning of TDGAP Parameters

Extensive experimentation has been conducted to identify which combination of parameters will allow TDGAP to perform best. Figures 13 and 14 summarize the relative performance of the various genetic parameters with respect to timing and wirelength respectively. There is one axis per control parameter. For each axis, the origin is at 0 and the parameter that achieved worst performance is farthest from the origin. All points in the K-charts correspond to solutions obtained from the same number of generations. Only one parameter is varied at a time. Further, for each axis, the various points are positioned according to their performances relative to each other. For example, for the crossover axis in Figure 13 $\mathcal{X}_1$ is closer to the origin than the point corresponding to $\mathcal{X}_2$. This should be interpreted as $\mathcal{X}_1$ is superior to $\mathcal{X}_2$.

Next, we shall briefly discuss the results obtained with the various genetic control parameters.

Initial population constructors

The choice of constructor function affects both the quality of the solution and the number of generations needed to reach a good solution. IPC_1 builds the initial population randomly. Therefore, the number of generations required to obtain an acceptable solution will be much larger than that required for other constructors to reach a solution of similar quality. In contrast, initial solutions constructed with IPC_2 are all of similar quality causing the search procedure to very quickly get trapped into a local minima of poor quality. Constructor IPC_3 exhibits a slow convergence toward good solutions. Constructor IPC_4 is the best with respect to timing characteristics while IPC_5 is best with respect to wirelength. However, in some cases IPC_5 exhibited superior behavior to that of IPC_4 with respect to both timing and wirelength. Figure 15 compares the performance of the five constructors during the first 2000 generations.

Population size

The value of the population size affects the run time, the convergence rate of the algorithm, as well as the solution quality. Based on experimentation, it has been found that a population size of about 24 is best for all circuits we placed (see Figures 13 and 14.) This confirms results reported in [5].

Crossover operators $\mathcal{X}_1$ and $\mathcal{X}_2$

Both crossover operators performed well. However, $\mathcal{X}_1$ performed noticeably better than $\mathcal{X}_2$. The performance of both operators are compared in Figure 16. Crossover operator $\mathcal{X}_1$ exhibited faster convergence rate and better solutions than $\mathcal{X}_2$ with respect to both improvement in timing characteristics and reduction in wirelength. The sizable difference between the performance of crossover operators $\mathcal{X}_1$ and $\mathcal{X}_2$ can be explained as follows. Operator $\mathcal{X}_2$ tends to disturb the solutions much more than operator $\mathcal{X}_1$. The reason is that the window size which controls the operation of $\mathcal{X}_2$ is dependent on the positions of cells affecting the selected critical path. Therefore, if two cells affecting the selected critical path are far apart, then the window size will be proportionally larger. Having a large window size means that there is a higher chance of disturbing more cells contained in the window but which are not related to the selected critical path. In contrast, for $\mathcal{X}_1$ only those cells that are affecting the selected path are disturbed. As a result, $\mathcal{X}_1$ has been chosen as the crossover operator in TDGAP.

The value of the crossover probability can also affect the solution quality. The effect of different crossover probabilities on timing and wirelength performances for the first 2000 generations are shown in Figure 17. From this figure we note that, as we increase the probability of crossover, the search tends to converge toward a better solution. However, when the crossover probability is too high, the search gets trapped at a local minima. Experimental results indicate that a crossover probability between 0.5 and 0.7 leads to solutions with superior timing and wirelength characteristics.

Selector functions $\S_1, \S_2, \S_3$, and $\S_4$

The selection operation is the gateway to next generation. At this stage decisions are taken as to who survives and who dies among the parents and offsprings. It is therefore very important to have a good selection function to speed up the convergence toward the desired solutions.

Figure 18 shows the effect of different selector functions. The behavior of selector functions $\S_2$ and $\S_4$ is similar. But selector $\S_2$ performed slightly better than $\S_4$ because of the limited freedom in randomly selecting some of the individuals. Both selectors $\S_2$ and $\S_4$ favor individuals with high fitness values, which made these selectors more likely to be trapped into a local minima. For example, as shown in Figure 18 selector $\S_4$ got trapped into a local minima from which it could not escape.

Selector $\S_3$ exhibited the worst behavior. This is because selector $\S_3$ has almost no control over the selection of individuals. This has caused the fitness of next generation to fluctuate in an unpredictable manner. In contrast, selector $\S_1$ performed the best among other selectors. Selector $\S_1$ gives a chance for all individuals to be selected at random, while allowing the best individual to always survive to the next generation. This saves the search from getting trapped into a local minima as with selector $\S_3$.

Mutation operators μ_1 and μ_2

The effect of the different mutation operations is shown in Figure 19. From this figure we note the preference of each mutation operator. For example, the improvement in the timing aspects with μ_1 was more than that with μ_2. The combination of these two operators resulted in a good timing improvements that is near the one obtained with μ_1 alone. Also, the combination resulted in a smaller total wirelength than the one obtained with either of the two operators when individually applied.

The effect of allowing the mutation operation to be applied on the best individual or not is shown in Figure 20. From this figures it is clear that disallowing the best individual from being mutated is better then allowing it. By mutating the best individual, a great deal of the past history of the search that produced the good genes of the best individual may get lost.

The algorithm is sensitive to the mutation probability. Experiments indicate that a mutation probability around 0.1 is a good choice. The effect of different mutation probabilities is shown in Figure 21. From this figure we note that having high probability such as 15% or 20% disturbed the solutions a lot causing them to loose their homogeneity and decreasing their rate of convergence. On the other hand, a low probability such as 5% did not allow the mutation operation to have any real effect, causing the search to get trapped into a local minima.

A summary of the final recommended parameters used in TDGAP is given in Table 3.

Constructor function	IPC_5
Crossover operator	$\mathcal{X}_1$
Mutation operator	$\mu_1 \cup \mu_2$
Selector function	$\S_1$
Population size	24
Crossover probability	0.5 - 0.7
Mutation probability	0.1

Table 3: A summary of the genetic parameters that were found to perform better than others with TDGAP.

Constant vs. dynamically decreasing population size

Run time requirements of genetic algorithms are larger than those of other iterative techniques such as simulated annealing and simulated evolution. Whereas simulated annealing and simulated evolution maintain a single solution, a genetic algorithm works with a population of individuals. The number of such individuals has a great effect on the total run time of the algorithm. The smaller the population size is, the smaller the run time will be. Our experiments with TDGAP indicated that the best population size would be 24 to 30 individuals, regardless of the problem size. We also noticed that the population fitness improves very rapidly during the early generations. The change in the population fitness is less rapid as the number of generations increases, until it becomes insignificant. The reason is that, the improvements in the population are caused by crossovers and mutations involving high scoring individuals. Therefore, toward the middle and later generations, the role of low scoring individuals becomes insignificant as a source of new fit individuals for the next generations. Hence, it seems reasonable to allow the population size to progressively decrease (in a controlled manner) with the number of generations. Such a decrease will cause a sizable reduction in run time without any noticeable change in solution quality.

To view the effect of this idea, we run TDGAP on CRC16 test case using two strategies. In the first strategy the population size was fixed to 30 individuals. For the second strategy, the population was allowed to progressively decrease using the following reduction procedure. The reduction procedure determines when to reduce the population size and by how much. The performance of the best solution is checked periodically every *Reduction_Period*. If no timing improvement by at least 3% is achieved during the last *Reduction_Period*, the population size is reduced by 20%. This reduction is allowed as long as the population size does not become less than half of the initial population size. The *Reduction_Period* parameter is also dynamically decreasing. Initially it is assigned a large value. Then each time the population size is checked, the *Reduction_Period* is reduced by a *Period_Factor*. This is performed because the convergence rate of TDGAP in the early generations is higher than in the later ones. Thus, the reduction procedure monitors the performance after shorter periods in those generations where the improvement is too slow. For the CRC16 test case, we set the *Reduction_Period* equal to 5000

Best Solution	Dynamic Population		Fixed Population	
	Slack	Area	Slack	Area
Initial	-2.47	87786.3	-2.47	87786.3
Final	+0.61	69959.1	+0.78	70580.8
Run Time %	54%		100 %	

Table 4: A summary of the initial and final values of the best solution with dynamic and fixed population size for wirelength and timing performance. Slack values are given in ns.

generations and the $Period\_Factor$ equal to 5%.

This idea bears some similarity with the cooling schedule of simulated annealing. Whereas in simulated annealing the cooling schedule does not have much influence on the runtime of the procedure, for the genetic algorithm a smaller $Reduction\_Period$ increases the rate at which the population size is reduced, thus leading to a much smaller runtime.

Figure 22 shows the timing and area performance of the best solution with dynamic and fixed population size. Both cases were run for 300,000 generations. The total run time for the case of fixed population size was almost double that of the case when the population size was dynamically reduced. A summary of the initial and final values of the best solution with dynamic and fixed population sizes is given in Table 4. From these tables we note that the quality of the results in both cases were of comparable quality, but the total run time for the case of dynamic population size has decreased by 46%. However more experimentation is required to tune the reduction schedule (that is, $Reduction\_Period$ and $Period\_Factor$).

4 Experimental Results

We experimented with several circuits, some of which are benchmark circuits. The characteristics of the circuits used are given in Table 5. The function of each circuit given in this table is as follows:

1. **Ck1**: A sample AHPL model that performs part of the stop and wait protocol.
2. **CRC16**: A 16-bit Cyclic Redundancy Checker.
3. **Highway**: A traffic light controller (test case from the OASIS system).
4. **Fract**: A fractional multiplier. The description of this circuit is given in [23].
5. **Struct**: A 16-bit multiplier (pure combinational circuit).

A summary of the initial and final values of the best solution for all test cases with respect to timing and area metrics is given in Table 6. The slack values given are obtained after the placement phase, but before routing. To obtain these values we have developed

Circuit Name	Bench-mark	# of cells	# of critical paths	# of rows in final layout
Ckl	No	209	200	8
CRC16	No	209	330	9
Highway	No	56	14	4
Fract	Yes	149	368	6
Struct	Yes	1952	1500	22

Table 5: Characteristics of the circuits used.

Circuit Name	Clock in ns	Run time in Hrs	Area			Slack		
			TDGAP	% increase	OASIS	TDGAP	% Impr	OASIS
Ckl	21	10.1	928 ×1016	9.1	923 × 936	+0.52	8.1	-1.14
CRC16	18	14.4	1131 × 968	5.5	1072 × 968	+0.61	17.7	-2.47
Highway	20	5.4	478 × 496	6.2	465 × 480	+0.51	11.4	-1.72
Fract	38	9.0	798 × 824	5.9	768 × 808	+0.31	6.5	-2.14
Struct	140	19.8	3046 × 2880	1.4	3019 × 2864	−1.20	0.5	-1.89

Table 6: Area and Slack performance comparison between TDGAP and OASIS.

and used a timing evaluator that checks for timing violations based on the reported α-critical paths. The placements obtained by OASIS and TDGAP were evaluated with respect to timing as well as overall wirelength. Timing performance improvements of up to 20% were obtained. The area values given are obtained after completing the routing phase and generating the layout. The *Magic* layout system has been used to view the layouts of the circuits and get their actual height and width. The improvement achieved by TDGAP with respect to timing aspects has resulted in a slight increase in the overall area (between 1% and 9%).

Many parameters affect the speed of TDGAP. Among these are the design size and the number of reported critical paths. The larger the number of cells and/or the number of critical paths, the larger will be the runtime.

The timing performance of the circuit Struct was not improved much after almost 20 hours of run time. This was due to two principal reasons: (1) the design is a very large purely combinational circuit; (2) the circuit has a very large number of critical paths (over 10,000) paths all of which were within $1.5ns$. These two characteristics combined to cause very large run-time with small improvements in performance.

This strongly suggests that path oriented approaches of the type described in this paper are not suitable for circuits similar to 'Struct'. When the number of *α-critical* paths is too large (near or larger than the number of cells), it would be more efficient to back annotate the nets of the circuit with their estimated interconnect delays (after placement) and re-verify the placement instance for timing problems using a timing-

analysis algorithm such as the zero-slack algorithm (ZSA) [24]. The ZSA algorithm computes for each individual cell the slack remaining on the longest path traversing that cell. If all cell slacks are positive this indicates that the placement instance is timing-problem free: otherwise it has long path problems. ZSA has the property that all cells on the slowest paths have the same smallest slack values. These cells are the critical cells and the nets driven by them are the critical nets. Therefore, to improve the timing of the placement instance, one would identify a number of these critical nets and attempt to improve their timing by assigning their cells to more suitable locations which will increase their respective slacks. This obviously requires that the crossover and mutation operators be designed to suit this strategy. Such an approach would be practical since ZSA exhibits a near linear time complexity, and would be faster than working with a very large set of paths. Furthermore, a local perturbation of a placement instance would not require the re-evaluation of all cell slacks. Only a limited number of these cells would have to be evaluated. We are currently investigating this approach.

Another observation that is in order concerns the genetic control parameters. A close examination of the plots reveals that if some of the genetic parameters are adapted during the run one would improve the run time requirement and the quality of the final solution. From experiments we identified two parameters that, if allowed to adapt, would lead to superior results: (1) the crossover probability; and (2) the mutation probability. Referring to Figure 17, starting with a high crossover probability of 90% then gradually reducing it to 70% seems to be a better choice then fixing it. Similarly, Figure 21 suggests a similar strategy for the mutation probability that is, starting with relatively large mutation probability of 20% and then gradually reducing it until it reaches 10%. These observations require further investigations.

TDGAP is implemented in the C language. Experiments were performed on a 64-bit DEC Alpha workstation that is running OSF/1 operating system at the speed of 100 MIPS.

All test cases require runtime in the order of hours. This is partly due to the nature of the genetic algorithm. However, in our case it is also due to a not so efficient implementation of the algorithm. Our initial concern was to correctly implement the idea. We were not initially too concerned about efficiency. For example, in the current implementation, for each generation and each individual placement all the paths are re-checked. This is not necessary since only a fraction of the paths would be affected by crossover or mutation. The data structure used by the algorithm is being changed to allow incremental verification of only the affected paths.

5 Conclusions

In this paper we presented a timing-driven placement program. The placement procedure follows the genetic algorithm. The program uses path timing data from a timing analyzer. The timing analyzer uses a new technique and a new criterion (*α-criticality*) to predict the critical paths prior to placement.

Extensive experiments were conducted to tune the parameters of the program and evaluate the extent of timing improvement. Results from benchmark circuits show that

timing improvement of upto 20% can be achieved by our placement system without any change to the logic of the circuit. The sizable timing improvement is accompanied with a very slight increase in the area of the circuit.

References

[1] T. Asano and S. Sato. Long path enumeration algorithms for timing verification on large digital systems. *Graph theory with applications to algorithms and computer sciences, John Wiley*, pages 25–35, 1985.

[2] R. Brayton, G. D. Hachtell, and A. L. Sangiovanni-Vincentelli. A survey of optimization techniques of integrated circuit design. *Proceedings of IEEE*, 69(10):1334–1362, October 1981.

[3] M. Burstein and M. N. Youssef. Timing influenced layout design. *Proceedings of 22nd Design Automation Conference*, pages 124–130, 1985.

[4] H. C. Chen, D. H. C. DU, and L.-R. Liu. Critical path selection for performance optimization. *IEEE Transactions on Computer Aided Design*, CAD-12:185–195, February 1993.

[5] J. P. Cohoon and W. D. Paris. Genetic placement. *IEEE Transactions on Computer Aided Design*, CAD-6:956–964, November 1987.

[6] J. Darringer et al. LSS: A system for production logic synthesis. *IBM J. Res Develop.*,, 28(5):259–274, 1984.

[7] W. Donath et al. Timing-driven placement using complete path delays. *Proceedings of 27th Design Automation Conference*, pages 84–89, June 1990.

[8] A. E. Dunlop et al. Chip layout optimization using critical path weighting. *Proceedings of 21st Design Automation Conference*, pages 133–136, 1984.

[9] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer Aided Design*, CAD-4(1):92–98, January 1985.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, INC., 1989.

[11] P. S. Hauge, R. Nair, and E. J. Yoffa. Circuit placement for predictable performance. *Proceedings of International Conference on Computer-Aided Design*, pages 88–91, 1987.

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

[13] Orbit Semiconductors Inc. Foresight Manual. *Sunnyvale, California.*, July 1992.

[14] M. A. B. Jackson and E. S. Kuh. Performance-driven placement of cell-based IC's. *Proceedings of 26th Design Automation Conference*, pages 370–375, June 1989.

[15] N. P. Jouppi. Timing analysis and performance improvement of MOS VLSI design. *IEEE Transactions on Computer Aided Design*, CAD-6(4):650–665, July 1987.

[16] B. Kick. Timing correction in logic synthesis. *Proceedings of ICCAD'87*, pages 299–302, 1987.

[17] S. Kirkpatrick, Jr. C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):498–516, May 1983.

[18] R. M. Kling and P. Banerjee. Empirical and theoretical studies of the simulated evolution method applied to standard cell placement. *IEEE Transactions on Computer Aided Design*, CAD-10:1303 – 1315, October 1991.

[19] I. Lin and D. Du. Performance driven constructive placement. *Proceedings of 27th Design Automation Conference*, pages 103–105, 1990.

[20] Rung-Bin Lin and E. Shragowitz. Fuzzy logic approach to placement problem. *Proceedings of 29th Design Automation Conference*, pages 153–158, 1992.

[21] MCNC Group. *OASIS 2.0 Reference Manual*, 1990.

[22] G. D. Micheli. Performance-oriented synthesis in the yorktown silicon compiler. *Proc. of ICCAD'86*, pages 138–141, 1986.

[23] H. Troy Nagle et al. *Intro to Computer Logic.* Prentice Hall, 1975. page 461.

[24] R. Nair et al. Generation of performance constraints for layout. *IEEE Transactions on Computer Aided Design*, CAD-8(8):860–874, August 1989.

[25] A. E. Ruelhi, P. K. Wolff, and G. Goertzel. Analytical power/timing optimization technique for digital system. *Proceedings of 14th Design Automation Conference*, pages 142–146, 1977.

[26] K. Shahookar and P. Mazumder. A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer Aided Design*, 9(5):500–511, May 1990.

[27] J. F. Shapiro. *Mathematical Programming: Structures and Algorithms.* New York, Wiley,, 1979.

[28] K. J. Singh et al. Timing optimization of combinational logic. *Proc. of ICCAD'88*, pages 282–285, 1988.

[29] Arvind Srinivasan, Kamal Chaudhary, and Ernest S. Kuh. Ritual: a performance-driven placement algorithm. *IEEE Transactions on Circuits and Systems*, pages 825–840, November 1992.

[30] P. R. Suaris and G. Kedem. A new approach to standard cell layout. *Proc. of ICCAD'87*, pages 474–477, November 1987.

[31] S. Suthanthavibul, E. Shragowitz, and Rung-Bin Lin. An adaptive timing-driven placement for high performance VLSI's. *IEEE Transactions on Computer Aided Design*, 12(10):1488–1498, October 1993.

[32] M. Vecchi and S. Kirkpatrick. Global wiring by simulated annealing. *IEEE Transactions on Computer Aided Design*, CAD-2:215–222, 1983.

[33] H. Youssef et al. Critical path issue in VLSI design. *Proceedings of International Conference on Computer-Aided Design*, pages 520–523, 1989.

[34] H. Youssef and E. Shragowitz. Timing constraints on signal propagation in VLSI. *Proceedings of International Conference on Computer-Aided Design*, pages 24–27, 1990.

[35] H. Youssef, E. Shragowitz, and S. Suthanthavibul. Prelayout timing analysis of cell-based VLSI designs. *Computer Aided Design*, 24(7):367–379, July 1992.

List of Figures

List of Tables

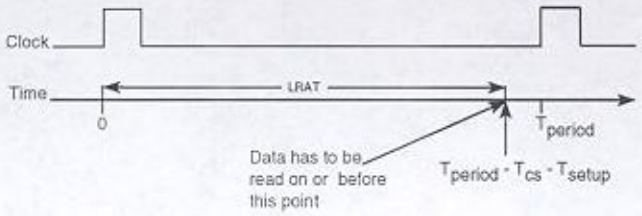

Figure 1: Long path problems. T_{cs} is the maximum clock skew and T_{setup} is the setup time.

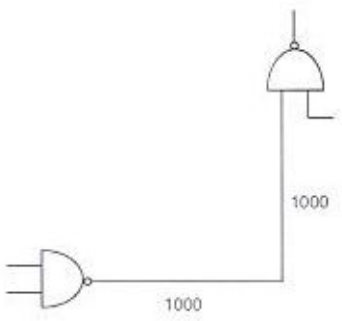

Figure 2: Illustration of a two point net.

ALGORITHM (*TDGAP*)
N_p= Population Size.
N_g= Number of Generations.
N_o= Number of Offsprings.
P_c= Crossover Probability.
P_μ= Mutation Probability.
Begin
Initial_Population(N_p)
For $j = 1$ to N_p
Evaluate *Fitness*(Population[j])
EndFor
For $i = 1$ to N_g
For $j = 1$ to N_o
$(s, t) \leftarrow$ *Choose_parents* (*CHOICE*)
With probability P_c Perform Crossovers
Offspring[j] $\leftarrow$ *Crossover*(x, y)
EndFor
(*SELECTION of next generation*)
Population $\leftarrow$ *Select*(Population, offspring, N_p)
For $k = 1$ to N_p
With probability P_μ Perform Mutation
Population[k] $\leftarrow$ *Mutation*(Population[k])
EndFor
For $m = 1$ to N_p
Evaluate *Fitness*(Population[m])
EndFor
EndFor
Return highest scoring individual in Population.
End

Figure 3: General algorithm of TDGAP.

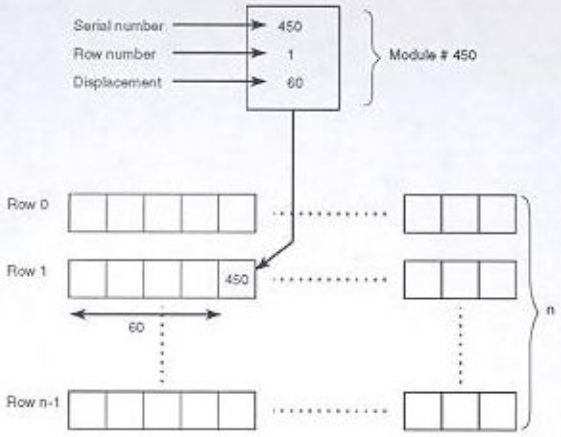

Figure 4: Encoding scheme of an individual.

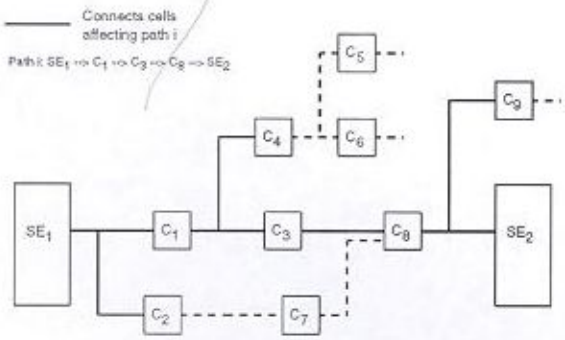

Figure 5: Cells affecting path i are: C_1, C_2, C_3, C_4, C_8, C_9, and SE_2.

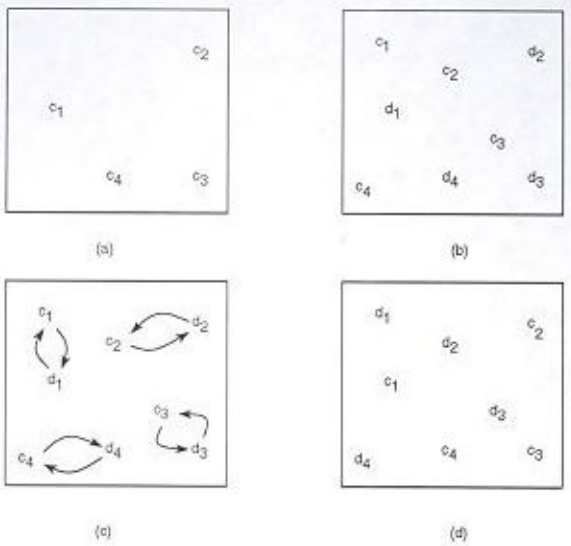

Figure 6: Crossover operator $\mathcal{X}_1$. $\beta = \{c_1, c_2, c_3, c_4\}$. (a) Parent 1 (passing parent), (b) Parent 2 (target parent), (c) Information passing, (d) Offspring.

ALGORITHM(Reconfigure)
Stop=0
Repeat
If $e_i \in \eta$ **Then**
skip this cell and go to the next one,
ElseIf $\sigma \notin \emptyset$ **Then**
Begin
pick a module e_j from σ and swap the modules e_i and e_j,
remove module e_j from σ
End
ElseIf $\rho \notin \emptyset$ **Then**
Begin
pick a module e_j from ρ and swap the modules of e_i and e_j,
remove module e_j from ρ
End
Else
Begin
skip this cell (all other cells will stay in their locations)
Stop=1
End
Until(all cells $\in \omega_o$ are scanned or Stop=1)

Figure 7: An algorithm used by $\mathcal{X}_2$ to reconfigure an offspring ξ_o.

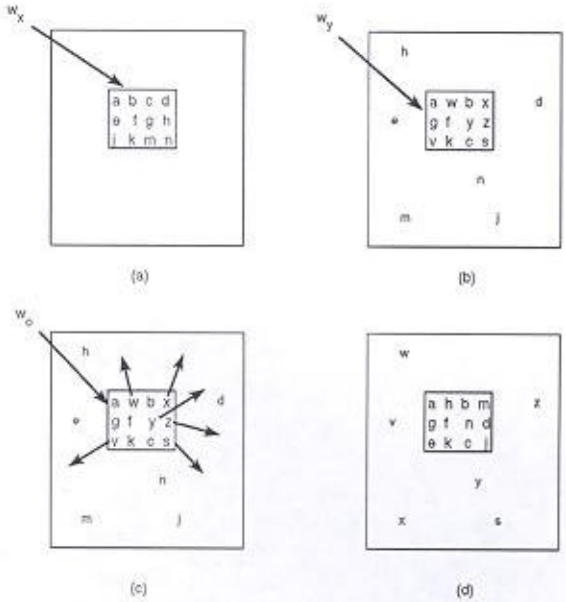

Figure 8: Crossover operator $\mathcal{X}_2$. β={a,h,m,n}, σ= {h,m,n}, ρ={d,e,j}, η={a,b,c,f,k,g}. (a) Parent 1 (passing parent), (b) Parent 2 (target parent), (c) Information passing, (d) Offspring.

```
ALGORITHM(Select Critical Path)
Total_CP=| CP |
CP_s^+ : critical paths in the passing parent with slack ≥ 0;
CP_s^- : critical paths in the passing parent with slack < 0;
CP_t^- : critical paths in the target parent with slack < 0;
loop_1=1, loop_2=1
If (CP_t^- ≠ ∅)
 For all cp ∈ CP_t^- Do Unmark(cp)
 While (loop_1=1)
  If (∃ cp ∈ CP_t^- | cp is not marked) Then
   Mark(cp)
    If (cp ∈ CP_s^+) Then loop_1 = 0
    EndIf
  Else
   loop_1=0
    For all cp ∈ CP_t^- Do Unmark(cp)
    While (loop_2=1)
     If (∃ cp ∈ CP_t^- | cp is not marked Then
      Mark(cp)
       (* let slack_s(cp) = slack of cp in source parent *)
       (* and slack_t(cp) = slack of cp in target parent *)
       If (cp ∈ CP_s^- and
        (SLACK_s(cp) > SLACK_t(cp))) Then
        loop_2 = 0
       EndIf
     Else
      loop_2=0
      cp=random(1,Total_CP)
     EndIf
   EndWhile(loop_2)
  EndIf
 EndWhile(loop_1)
Else
 cp=random(1,N_CP)
EndIf
End.
```

Figure 9: An algorithm used by $\mathcal{X}_1$ and $\mathcal{X}_2$ to select cp from CP.

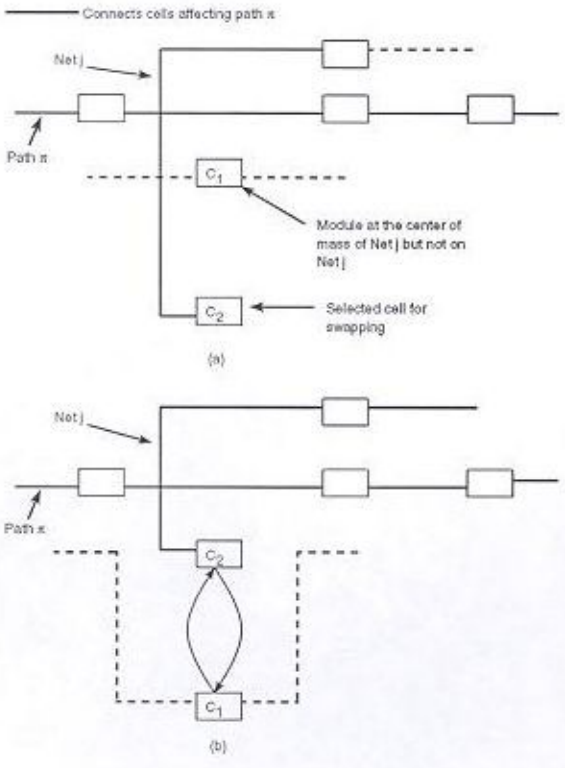

Figure 10: The operation of mutation operator μ_1. (a) Before mutation, (b) After mutation.

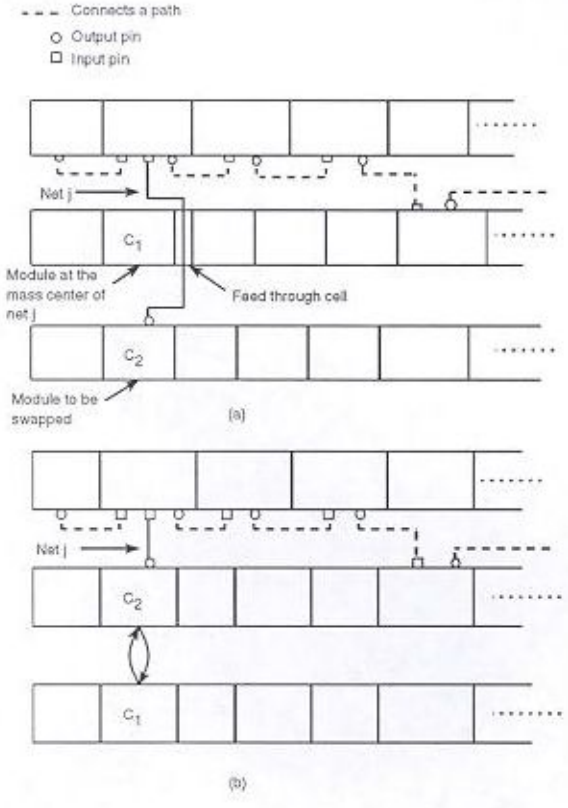

Figure 11: The operation of mutation operator μ_2. (a) Before mutation, (b) After mutation.

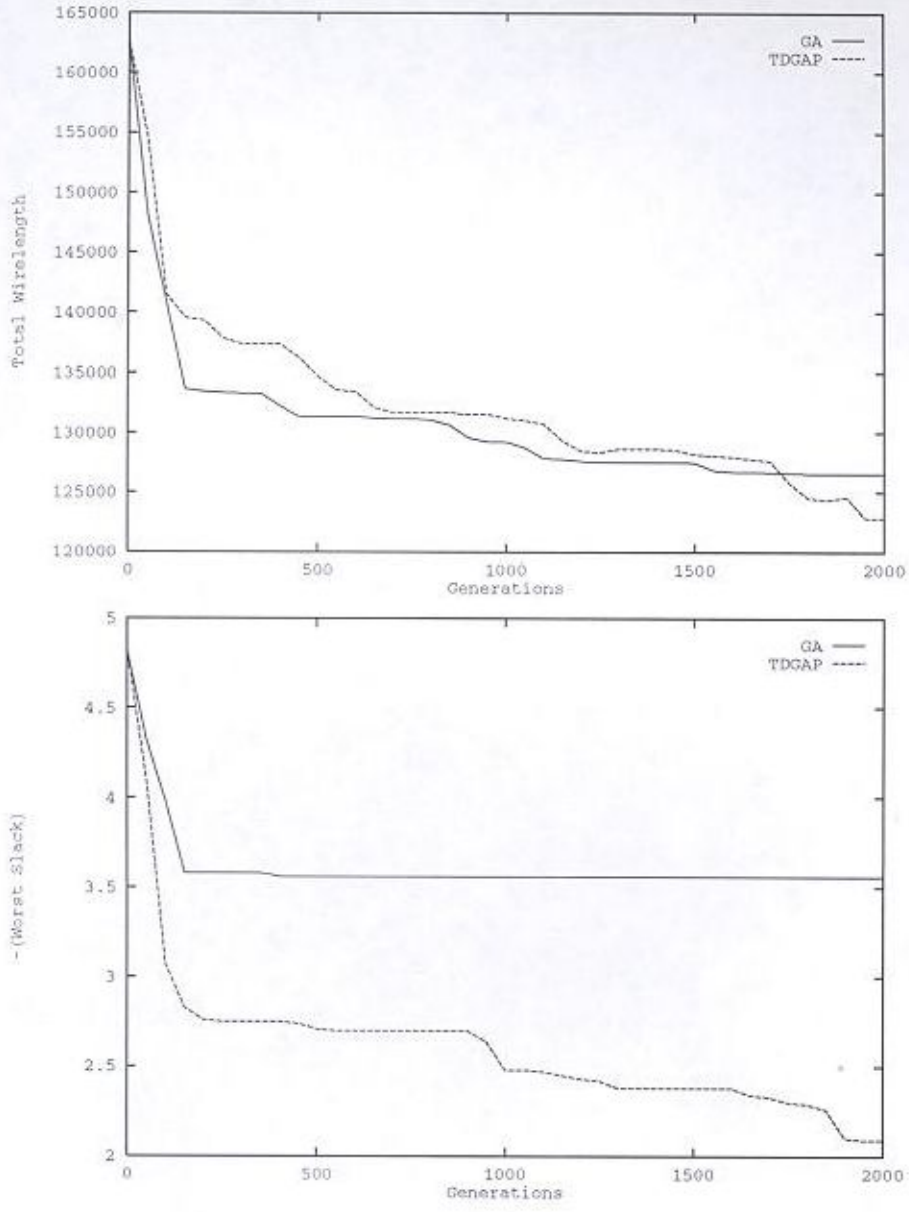

Figure 12: Comparison of best solution with respect to timing and wirelength performance obtained by the basic GA (mutation before selection) with that obtained by TDGAP (mutation after selection).

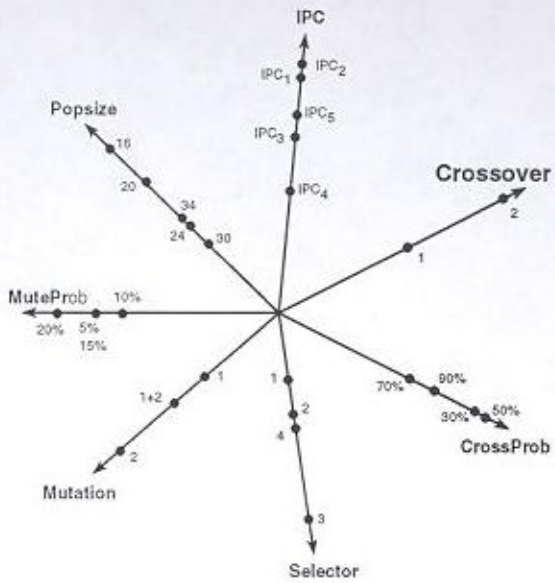

Figure 13: Relative timing performance of TDGAP with various combinations of genetic parameters. S=§, M=μ, X=$\mathcal{X}$.

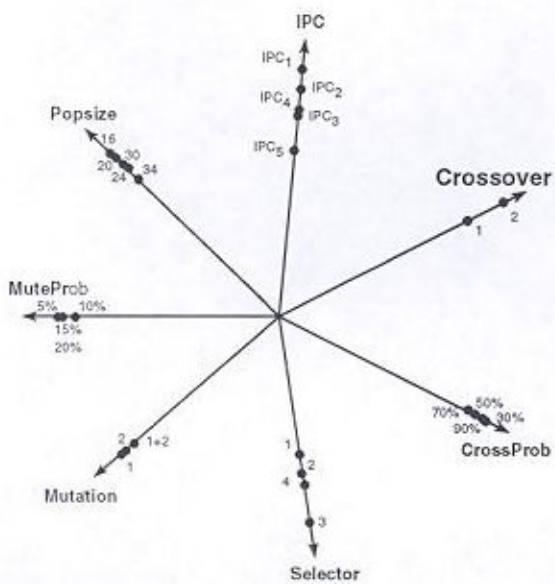

Figure 14: Relative wirelength performance of TDGAP with various combinations of genetic parameters. S=§, M=μ, X=$\mathcal{X}$.

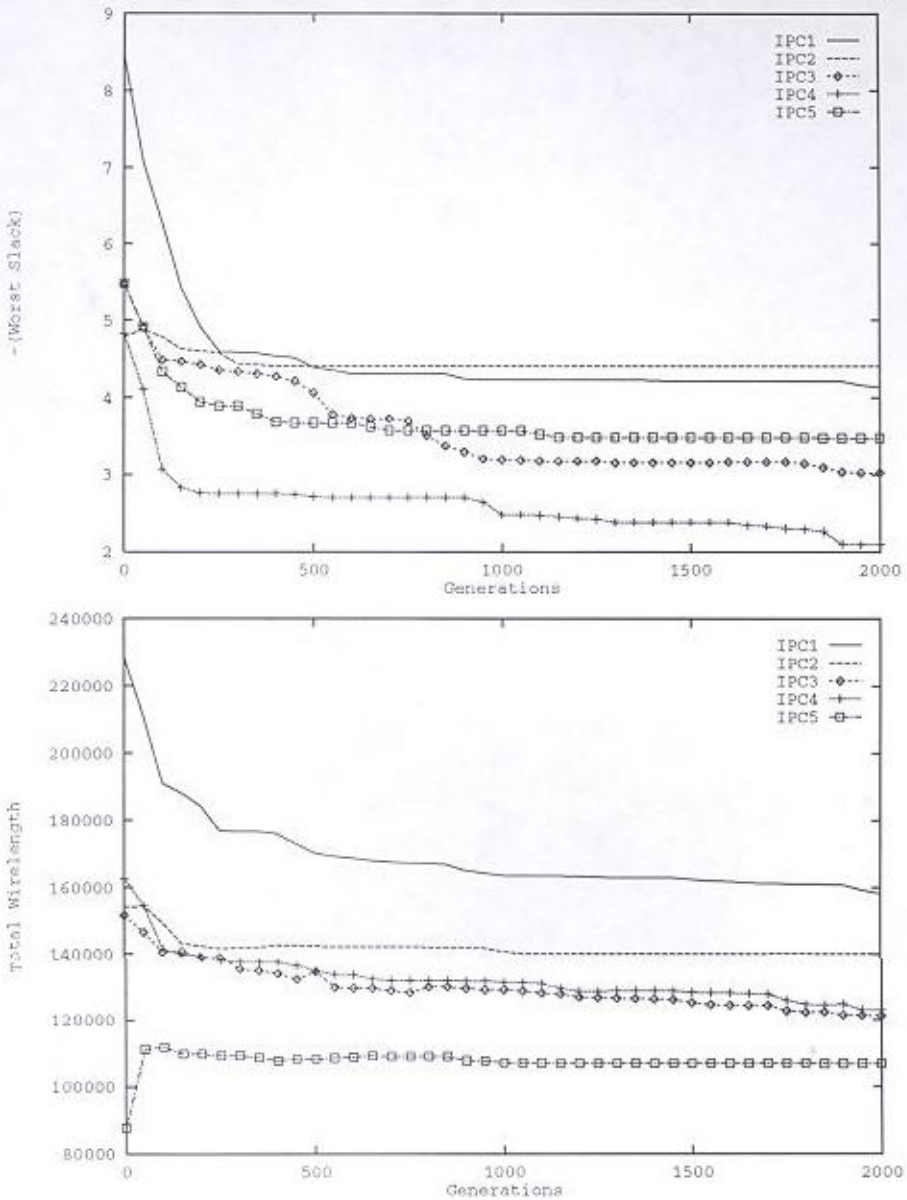

Figure 15: Timing and wirelength performance of the best solution with constructor functions IPC_1, IPC_2, IPC_3, IPC_4, and IPC_5.

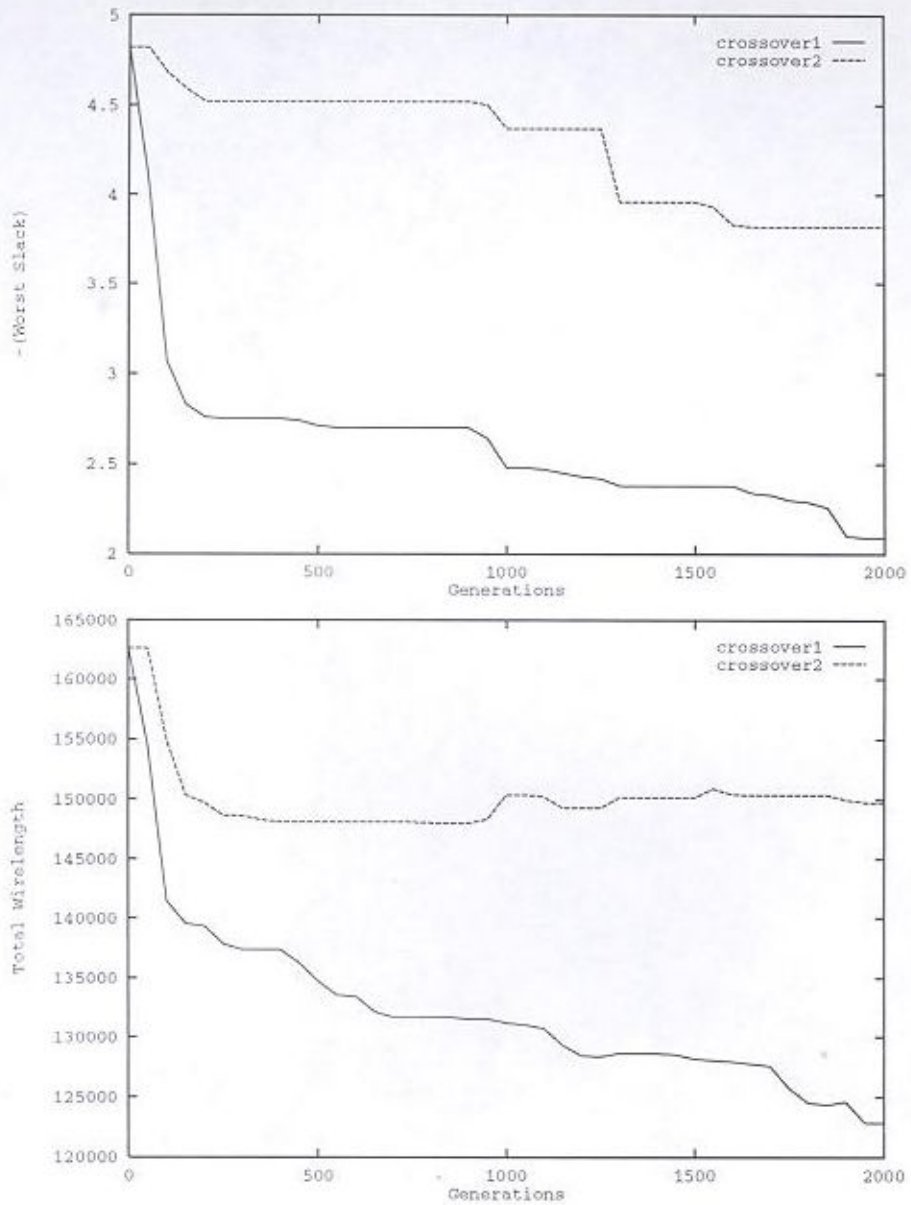

Figure 16: Timing and wirelength performance of the best solution with crossover1 ($\mathcal{X}_1$) and crossover2 ($\mathcal{X}_2$).

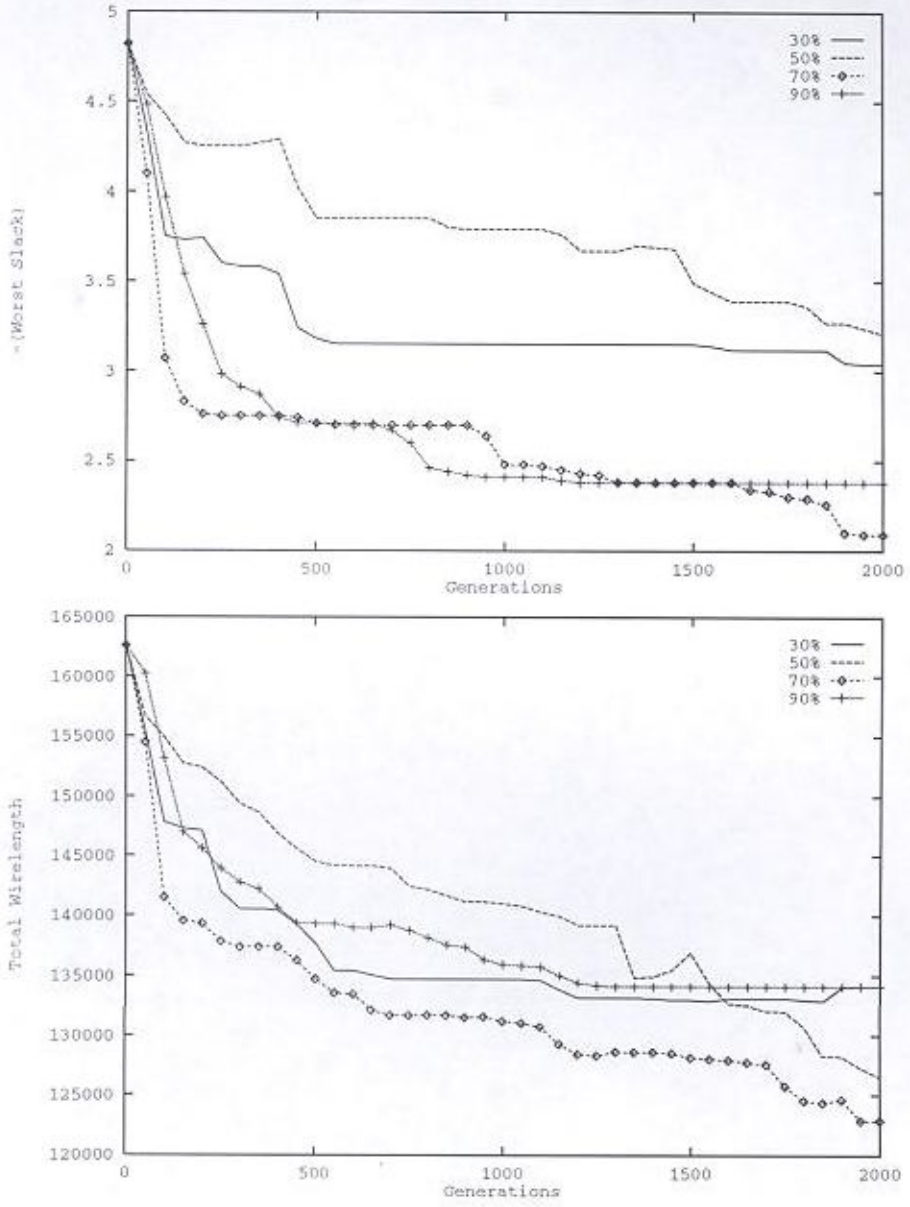

Figure 17: Timing and wirelength performance of the best solution with different crossover probabilities.

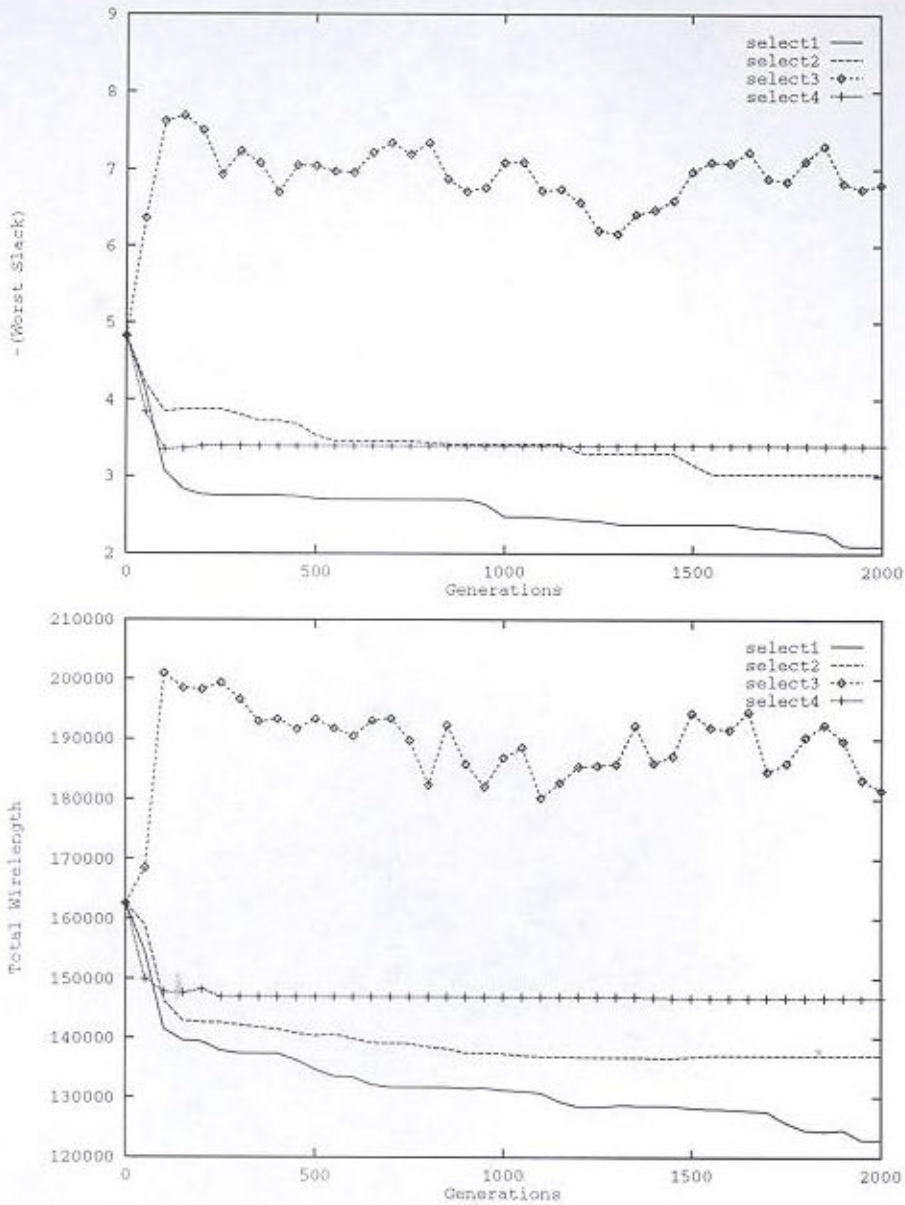

Figure 18: Timing and wirelength performance of the best solution with selector functions select1 ($\S_1$), select2 ($\S_2$), select3 ($\S_3$), and select4 ($\S_4$).

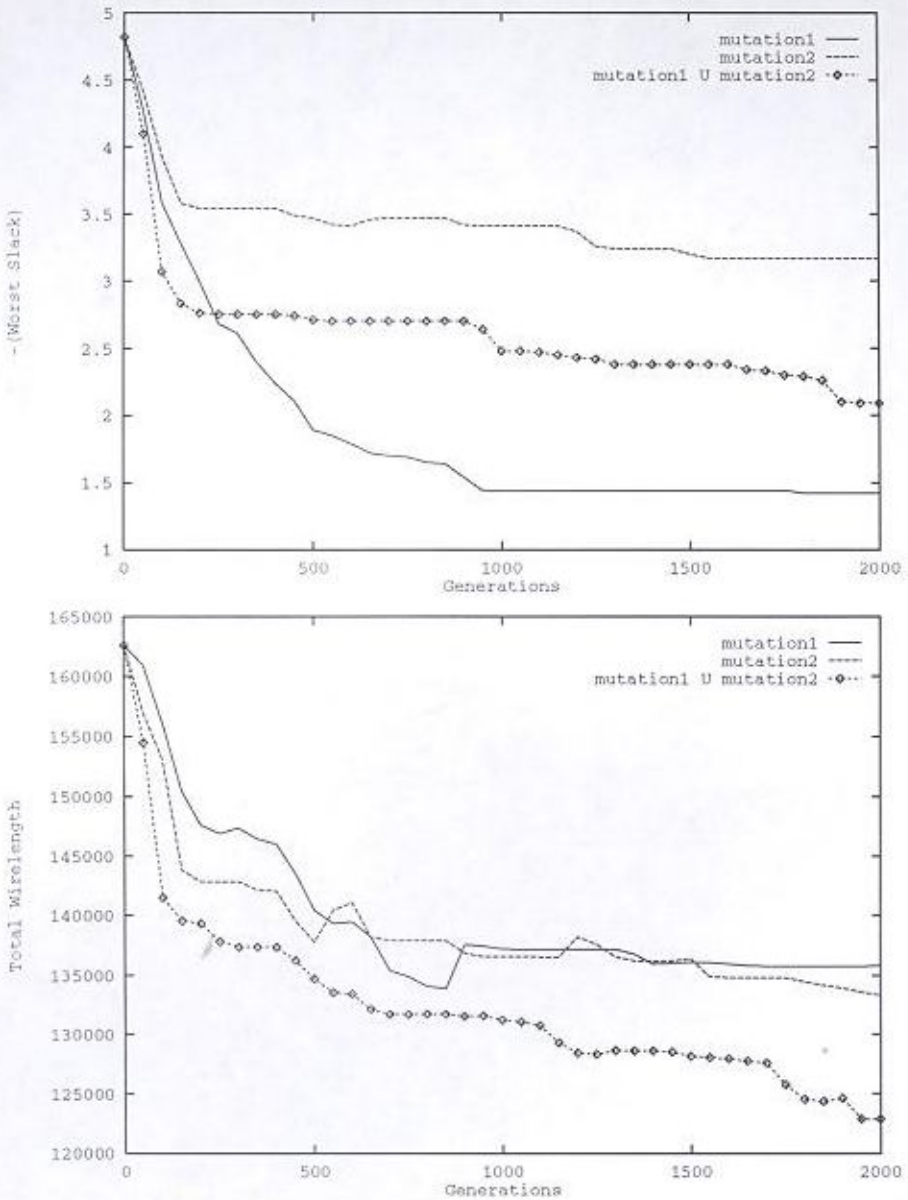

Figure 19: Timing and wirelength performance of the best solution with mutation1 (μ_1), mutation2 (μ_2), and mutation1 $\cup$ mutation2 ($\mu_1 \cup \mu_2$).

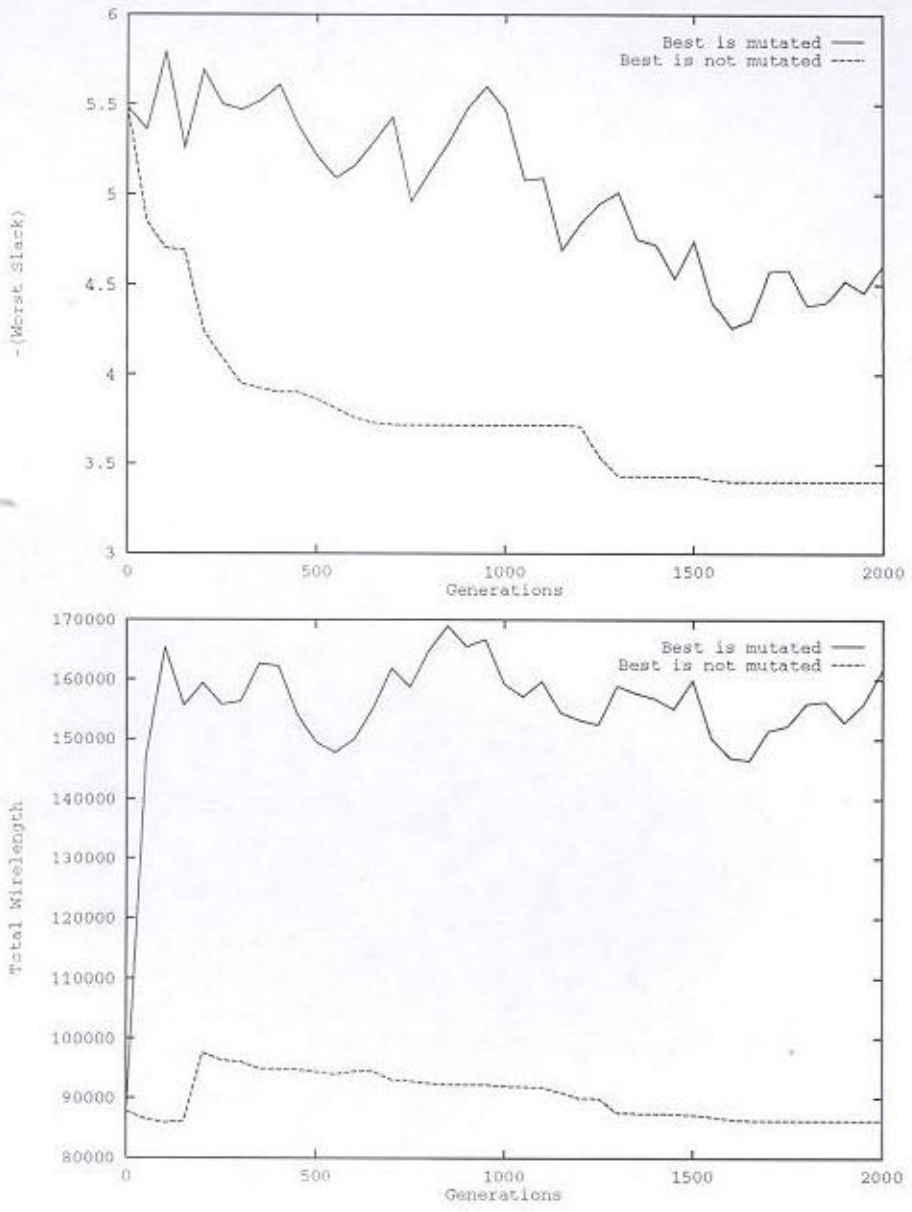

Figure 20: Timing and wirelength performance of the best solution in case of allowing or disallowing the best individual from being mutated.

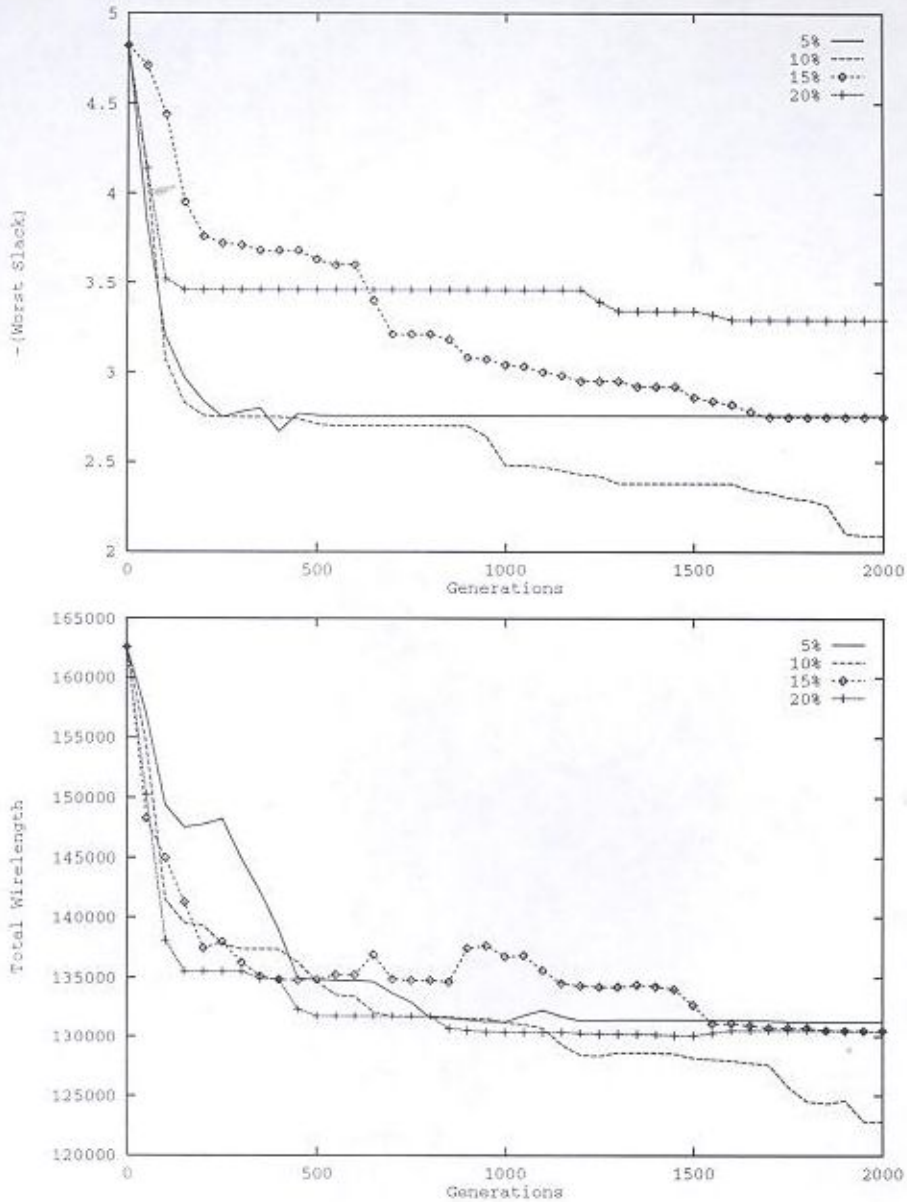

Figure 21: Timing and wirelength performance of the best solution with different mutation probabilities.

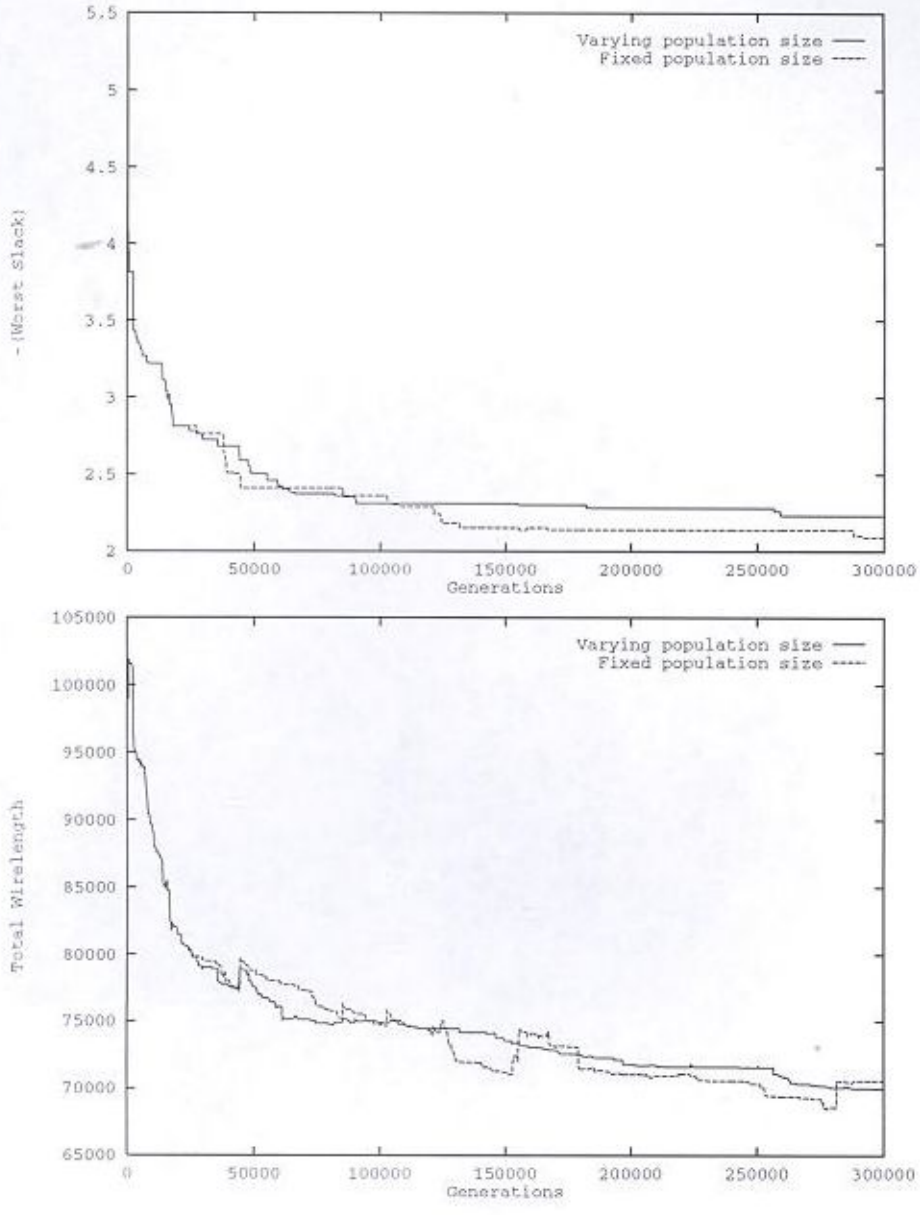

Figure 22: Timing and wirelength performance of the best solution with dynamic and fixed population size.