Universität Paderborn — Fakultät EIM-I
Fachgebiet Technische Informatik

A Comparison of Algorithms for the Generation of Layouts based on Reconfigurable Slots on FPGAs

Master's Thesis
am Fachgebiet Technische Informatik
Prof. Dr. M. Platzner

Institut für Informatik
Fakultät für Elektrotechnik,
Informatik und Mathematik
Universität Paderborn

vorgelegt von
Yashwanth Tadakamalla
am
21.06.2023

Prüfer: Prof. Dr. M. Platzner
Prof. Dr. Sybille Hellebrand

Yashwanth Tadakamalla
Matrikelnummer: 6896890
Paderborn
33102

Acknowledgement

I would like to express my heartfelt gratitude to my parents for their unwavering support throughout my academic journey. Their constant encouragement, love, and belief in me have been the driving force behind my accomplishments.

I am also deeply indebted to my supervisor, Prof. Dr. M. Platzner, for their invaluable guidance, mentorship, and unwavering belief in my abilities. I am grateful for the countless hours they have dedicated to reviewing my work, providing constructive feedback, and inspiring me to reach new heights. Their support and encouragement have been truly transformative, and I am fortunate to have had the opportunity to learn from them.

Lastly, I would like to acknowledge and thank myself for the determination, perseverance, and hard work I have put into completing this master's thesis. This journey has challenged me in countless ways, but through it all, I have remained committed to my goals. I am proud of my accomplishments and the personal growth I have experienced along the way.

Abstract

The execution of real-time tasks on dynamically reconfigurable FPGAs poses significant challenges, particularly in the areas of scheduling and placement. To overcome these challenges, in [13] a new approach for mapping real-time periodic tasks to the FPGA based on micro-slots has been introduced. On the one hand, this model supports the application of prior real-time scheduling methods and lends itself to a practical realization. However, the model also introduces new problems such as reconfigurable slot generation and layout generation. For the layout generation problem, Heuristic [13] and Optimal [12] solution has been presented. In this work, we solve the layout generation problem using Simulated Annealing and Evolutionary Strategy algorithms. The performance of these algorithms is evaluated against the existing Heuristic and Optimal solution based on FPGA utilization and runtime. The comparison's outcomes have offered information on the efficiency and effectiveness of Simulated Annealing and Evolutionary Strategy algorithms for layout generation based on reconfigurable slots. This study intends to contribute to the field of computational design by exploring different methods for layout generations based on reconfigurable slots on FPGA.

Contents

List of Figures

List of Tables

1 Introduction

Field-programmable gate arrays (FPGAs) have emerged as versatile integrated circuits that can be reprogrammed after manufacturing to perform any desired logic function. Unlike application-specific integrated circuits, FPGAs offer flexibility and customization to meet specific requirements, leading to sustained growth in their development [3]. FPGA technology has its roots in the late 1970s, when the concept of programmable logic devices (PLDs) was introduced. PLDs enabled the customization of digital logic circuits, but their capacity and flexibility were limited. The breakthrough came in the mid-1980s, when Xilinx introduced the first commercial FPGA, which provided a new level of programmability and reconfigurability. This marked the beginning of a new era in digital design, where hardware functionality could be modified and optimized post-manufacturing.

Today, FPGAs find extensive applications in various domains, particularly in embedded systems. They are widely utilized in telecommunications, automotive electronics, aerospace systems, medical devices, and many other industries. Their adaptability and high-performance capabilities make them indispensable in scenarios where specialized hardware is required to handle complex computations or data processing in real-time applications [13]. FPGAs provide several advantages over traditional CPUs and GPUs, such as higher processing speeds and lower latency, attributed to their reconfigurable hardware nature. However, FPGA design poses significant challenges, demanding expertise in digital design, computer architecture, and electrical engineering.

Within the field of FPGA design, the placement of logic components is a critical stage that significantly influences system performance. Placement is one of the most critical stages in the design of FPGAs [19]. It involves establishing precise locations for the various logic components, including flip-flops, multiplexers, look-up tables, and memories. Placement aims to minimize power consumption, avoid congestion in the routing resources, and minimize signal propagation delay. Achieving optimal placement is particularly challenging due to limited routing resources, making it difficult to connect all logic components efficiently. Poor placement can result in longer signal propagation times, increased power consumption, and diminished system performance. However, there are scenarios where modules exist independently and do not require any connections between them. By removing the constraint of interconnections, we can focus on optimizing other aspects of placement, such as area utilization, resource allocation, and overall performance.

When working with FPGAs in real-time settings, placement becomes even more critical as it directly affects the performance of the system. In real-time settings, applications require high speed and low latency processing, which means that the placement of logic elements must be carefully optimized to minimize the delay and maximize the performance. In the past, many approaches have been presented to address these challenges, but most of them rely on idealized assumptions about the reconfigurability of FPGAs

and the capabilities of commercial tool flows. As a result, they may not be effective in addressing the real-world challenges associated with scheduling and placement on FPGAs. In a paper [13] a new approach for mapping real-time periodic tasks to the FPGA based on micro-slots has been introduced. In this work, the authors have revisited the problem of scheduling and placement of real-time tasks on FPGAs. The authors have proposed a 2D slot-based reconfiguration area model, where reconfigurable slots comprise a number of so-called micro slots [13]. Here a micro slot is defined as a certain rectangular area on the FPGA with a given set of logic resources. The creation and partition of these micro slots are supported by the commercial tool flows[13]. On the other hand, a reconfigurable slot is a rectangular area of logic resources comprising a number of such micro slots. Depending on the task set, a number of micro slots are aggregated into a reconfigurable slot that can accommodate a single task at a time. This area model will enable us to take advantage of research in the field of reconfigurable hardware operating systems[8]. Under this special 2D area model, the problems of scheduling and placement problem are converted into the problem of creating a reconfigurable slot with a specific size and placing them on the FPGA. The authors have introduced Heuristic [13] and Optimal [12] solutions for both creating a reconfigurable slot and placing them on the FPGA.

This paper will mainly concentrate on the placement of the reconfigurable slots on the FPGA. In this paper, the layout generation problem of this special 2D area model is solved using Simulated Annealing (SA) and Evolutionary Strategy (ES) algorithms. These two algorithms are popular optimization algorithms that are often used in the field of FPGA layout generation. The SA algorithm is well suited for solving problems where the solution space is large and complex, and the objective function is noisy or has many local optima. SA is a stochastic optimization algorithm that is inspired by the annealing process in metallurgy. On the other hand, ES is also a stochastic optimization algorithm that is inspired by biological evolution. The FPGA layout generation problem is a good candidate for ES as the optimization objective is highly non-linear, and there are most likely to be multiple local optima.

This layout generation problem is divided into two separate research questions that require solving:

1. **Optimization problem**: The optimization problem involves determining the minimum bounding rectangle area that can enclose a given set of reconfigurable slots, each characterized by its associated area, and the width (W) and height (H) of an FPGA. The objective is to minimize the bounding rectangle while accommodating all the reconfigurable slots.

2. **Decision problem**: The decision problem involves selecting a specific FPGA from the Zynq family and determining whether it is possible to place all the given reconfigurable slots, each associated with an area, on the selected FPGA without any overlap.

The results of these algorithms are then compared with the previously presented Heuristic [13] and Optimal [12] solutions using the runtime and the quality of the result, i.e., FPGA utilization as the main metric.

The remainder of this paper is structured as follows: In Section 2 states related work and will briefly go through the background topics where the special 2D slot-based recon-

figuration area model and the solutions of Heuristic and Optimal layout generation are explained and knowledge about SA and ES algorithms. In Section 3 the approach of SA and ES for solving the Optimization problem is discussed. In Section 4 the approach of SA and ES for solving the Decision problem is explained. In Section 5 the experimentation and comparison results are shown. Finally, Section 6 concludes the paper.

2 Related Work and Background

This section provides an overview of the existing literature on floorplanning, a crucial aspect of electronic design automation. Subsequently, we will delve into the necessary background topics that are required for a more comprehensive understanding of subsequent sections.

2.1 Related Work

Previous work on hardware tasks has predominantly utilized rectangular-shaped modules. In solving the rectangle packing problem, the P-admissible solution space has been used in [19], where the authors utilized a sequence pair to represent each packing and employed a simulated annealing algorithm to search the solution space. Another approach, proposed in [2], utilized a fast online placement algorithm based on handling empty spaces on the FPGA device.

In [28], the authors utilized a placement method that depends on a partitioning of reconfigurable resources and used a hash matrix data structure to maintain the free space. The authors in [29] employed both simulated annealing and genetic algorithm for the placement of symmetrical FPGA. This paper included two stages; the first stage optimized the placement solution globally using GA, and the second stage locally improved the solution to overcome the slow convergence of the genetic algorithm.

The Vertex List Set technique was used to maintain contour information for each fragment of an unoccupied area in the reconfigurable device using the 2D Adjacency heuristic for placement proposed in [25]. The same authors then proposed the 3D Adjacency heuristic for placement, adding time as a fourth dimension. In [18], the authors introduced an efficient Simulated Annealing-Based VLSI Floorplanning Algorithm for Slicing Structure. In this work, the authors first analyzed the reason for dead space and proposed an intuitive fast approach to determine module orientation. They used normalized Polish expression to represent the floorplan and employed the Simulated Annealing algorithm to search the solution space for the optimal solution.

Chazelle et al. in [6] introduced a fast successful algorithm for strip packing known as the bottom-left heuristic, implemented in total quadratic time. However, this algorithm cannot be applied to reconfigurable computing, as the modules might leave the system. The BLD strategy, an improved version of Bottom-left, was introduced by Hopper and Turton in [14], in which the best result is chosen after rectangles are sorted according to various criteria, including height, perimeter, width, and area. In [30], Zhang proposed a recursive heuristic HR, where the algorithm places the first rectangle in the bottom-left corner of the sheet and repeats the process until all areas have been filled.

Metaheuristics were also utilized, with authors introducing a new and improved level heuristic to address the rectangular strip packing problem and variable-sized bin backing problem in [21]. In [16], the author designed an order-based GA combined with a bottom-left algorithm. In [4] authors have hybridized the best-fit heuristic with metaheuristic

approaches, such as simulated annealing (BF + SA), Tabu search (BF + TS), and genetic algorithms (BF + GA).

2.2 Background

FPGAs are becoming increasingly popular in the field of embedded systems due to their high processing speed and ability to perform parallel processing. With the advances in FPGA technology and the availability of real-time operating systems, there is now a growing trend towards using FPGAs in real-time contexts as well [13]. In such contexts, FPGAs can be used to process data in real-time and respond to events or signals with very low latency, which is essential for applications such as digital signal processing, control systems, and real-time data acquisition. While executing real-time tasks on FPGA there is always a tight interdependency between scheduling and placement. While scheduling determines the order in which tasks are completed, placement determines where on the reconfigurable fabric a given task is mapped. Next, we will delve into the utilization of a specialized 2D area model designed for slot creation, task assignment, and layout generation. Further, we will concentrate on Heuristic and Optimal approaches for layout generation.

2.2.1 Area Model

This section describes the special 2D slot-based reconfiguration model used for the slot creation and task assignment as well as for the layout generation algorithms. The summary of the area model serves as a foundation for the layout generation algorithms.

The special 2D slot-based reconfiguration model is based on the definition of a so-called micro slot, which is a rectangular area of hardware resources. The whole FPGA is divided into rectangular reconfigurable regions to create micro slots. This process is done in such a way that each micro slot comprises the same type and an equal number of resources which makes it easy to map the hardware tasks on an FPGA. It should also be considered that each micro slot can be supported by commercial tool flows for partial reconfiguration and for bitstream generation [13]. This makes the area model practically useful. When a task needs to be executed on an FPGA the task needs to be mapped to one or more micro slots on an FPGA. A "slot" is the term for the rectangular area formed when micro-slots are allotted and combined to effectively serve a task on an FPGA. This slot serves as a specific location on the FPGA where the task can be successfully carried out [12]. Here the size of the micro slot needs to be chosen big enough to accommodate small tasks. Nonetheless, the micro-slot size is kept as tiny as feasible to reduce fragmentation and optimum resource use. Here all area characterizations for tasks and devices are shown in the number of micro slots. In Figure 2.1, you can see the partitioning of the Zynq devices into above mentioned micro slots. Here each micro slot comprises the resources shown in Table 2.1.

Now assuming a list of periodic independent hardware tasks are given as $\Gamma = \{\tau 1, \tau 2, \ldots, \tau n\}$, where for each task τ_i comprises 3 variables that are k_i which represents the amount of required micro slots, c_i represents the upper bound for the task execution time and p_i represents the period which equals to the relative deadline of the task. By this, each task can be represented as $\tau_i = (k_i, c_i, p_i)$. Hardware task reconfiguration can only be justified if the total amount of required resources Σk_i exceeds the number of micro slots available

resource type	amount
slices	600
LUT	2400
registers	4800
DSP	20

Table 2.1: Minimum resources of a single micro slot

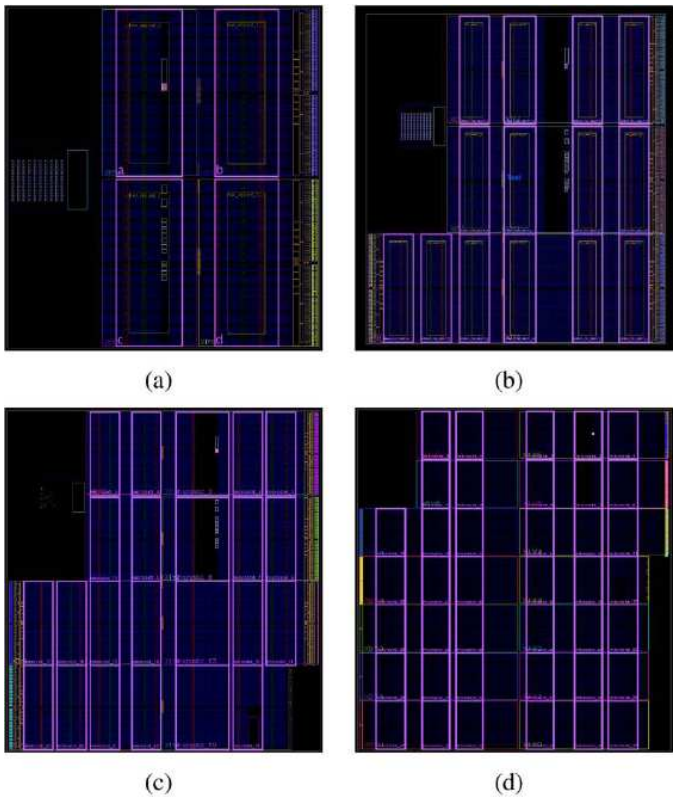


Figure 2.1: Partitioning into micro slots for the Xilinx Zynq 7010 (a), Xilinx Zynq 7020 (b), Xilinx Zynq 7030 (c), Xilinx Zynq 7020 (d) [13].

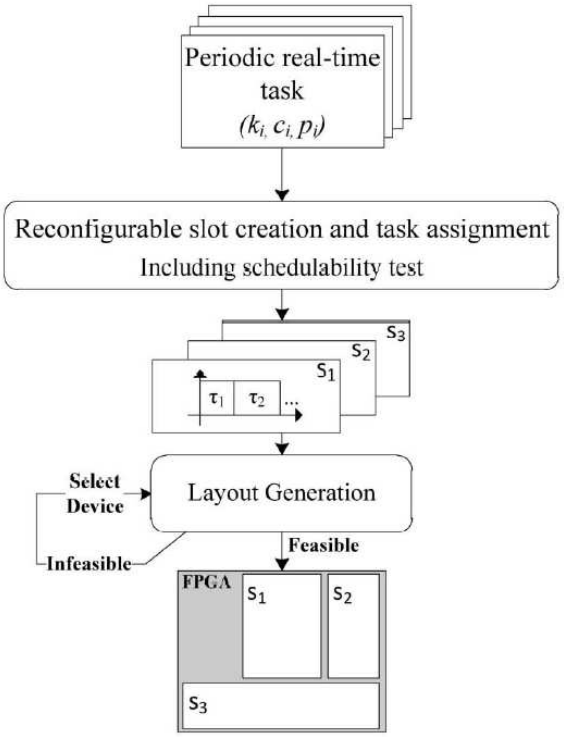


Figure 2.2: An approach for mapping periodic real-time tasks to an FPGA [13].

on the FPGA. In other words, if there are not enough micro slots available on an FPGA then reconfiguration is required. Once task sets are generated the next steps are divided into two parts which are hardware task assignment or reconfigurable slot creation and layout generation. Figure 2.2 shows the approach that the authors have used for mapping periodic real-time tasks to an FPFA. For the reconfigurable slot creation and layout generation steps, authors have proposed heuristic [13] and optimal [12] techniques. The reconfigurable slot creation step usually involves creating reconfigurable slots so that each task, along with all of its instances, can fit into exactly one slot. Furthermore, all tasks assigned to a single slot must be schedulable to ensure optimal performance. At the end of this phase, a list of reconfigurable slots with the required area is given in the number of micro slots. In the layout generation task list of reconfigurable slots with the required area is provided to the algorithm which in turn provides a layout that provides slots with widths and heights.

2.2.2 Heuristic Approach for Layout Generation

The list of reconfigurable slots with their corresponding areas that was generated from the creation of reconfigurable slots serves as the starting point for this step. The heuristic layout generation method does not take any particular FPGA from the Zynq family into consideration. The main objective here is to find how many micro slots are required to fit the given reconfigurable slots in the specified FPGA of given width W and height H.

This approach can be easily explained using an example using Figure 2.3. Let us

consider there are 7 reconfigurable slots $S_1 ... S_7$ that need to be placed on an FPGA of width $W = 5$ and height $H = 9$. W x H gives the number of available micro slots on a given FPGA here for example it is 45. The heuristic approach to the layout generation problem is divided into 3 parts.

In the first part, each reconfigurable slot S_j from the list needs to get a rectangular shape w x h is determined using the following rules. If the required area k_{Sj} corresponding to the reconfigurable slot S_j is less than or equal to W which is the width of the given FPGA, then the resulting shape of that reconfigurable slot S_j will be $W * 1$ and on the other hand, if k_{Sj} is more than W, then a shape needs to be created using multiple rows with the constraint that all rows of the shape are fully utilized. The shape of the reconfigurable slot is chosen in such a way that $cols$ x $rows = k_{Sj}$ with the largest possible $cols \in [1, 2, \ldots., W]$. Following this rule ensures that there will not be any wastage of space of micro slots. By using this now the below given 7 reconfigurable slots can be converted into rectangular shapes. S_1 can be converted to 3×2, $S - 2$ will be 2×2, S_3 will be 4×2, 4×1 for S_4, 3×2 for S_5, 1×1 for S_6, and 3×1 for S_7. Once every reconfigurable slot gets the rectangular shape using the above-mentioned rule. Now it is time for grouping these slots into distinct groups, denoted as G_i. Here i denotes the height of the rectangle-shaped slots. The set of groups G is subsequently sorted in a non-increasing order based on the value of i. In the example G_3 contains S_5, G_2 contains S_3 and S_1, G_1 contains S_4, S_7, S_2 and S_6 [13]. Once the first part is done there will be groups with sorted and non - increasing i.

In the second step, the authors introduced so-called frames which are formed from the rectangle-shaped slots. A frame can be referred to as a rectangular strip that covers a portion of the FPGA surface. By using the sorted groups G from the first part, The first shaped slot of the group with the highest height is inserted into the frame to construct the initial frame. This method can be applied to the example. First, place S_5 into the first frame, F_1, which sets the size of the frame to be $W \times 3$. If there is still room in F_1, try to horizontally stack more slots into the frame by scanning the remaining shaped slots in G in sorted order. Then choose S_2 and insert it into F_1, aligning it with the bottom row, since F_1 can only hold shaped slots with a maximum width of two. A new frame is started if no more shaped slots can be added to the existing one without going against its size restrictions. In the given example F_1 has the shape 5×3 containing S_5 and S_2, F_2 took the shape 5×2 containing S_3 and S_6, F_3 with shape 5×2 containing S_1, F_4 with shape 5×1 containing S_4, and finally F_5 with shape 5×1 containing S_7.

Once the frames are ready with all the reconfigurable slots in them. The third part begins by placing the frames onto the FPGA fabric. This is accomplished by taking into account the actual frame widths in F, which refers to the number of columns that are (partially) really used. F_1, the frame with the largest real width, is put in the bottom-right corner of the FPGA surface first. The remaining frames are then placed in order of non-increasing actual widths. This placing technique frequently results in a larger unused area at the fabric's top-left corner. The authors have mentioned that currently there is no use for this unused area thus they could have placed the frames randomly onto the FPGA fabric. And the authors also mentioned that there is no guarantee that the resulting layout fits within an FPGA of dimensions $W \times H$. If this is the case, a larger FPGA will need to be selected to accommodate the layout. The heuristic is a simple rule-based construction technique for the layout. The heuristic layout generation approach is very efficient in terms of runtime [13].

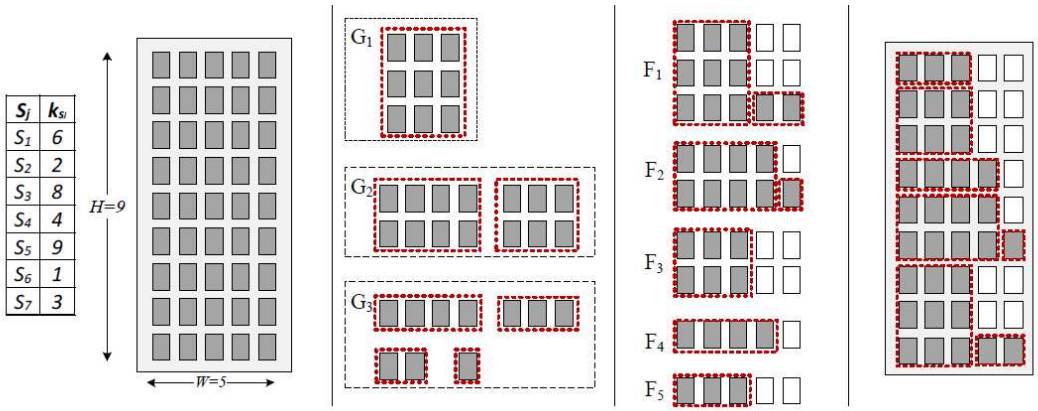


Figure 2.3: An example layout generation using the heuristic approach. Dotted lines indicate the shape of the reconfigurable slot created in step one [13].

2.2.3 Optimal Approach for Layout Generation

As same as heuristic approach the optimal layout generation step starts with the list of reconfigurable slots, defined only by their required area or sizes. Here in [12] the layout generation is solved in the form of a Quadratic Constraint Program (QCP). QCP is a mathematical optimization technique used to solve problems that involve quadratic functions subject to constraints. QCP can be used in the context of layout generation to optimize the arrangement of reconfigurable slots inside a specific region while adhering to particular constraints. QCPs can then be solved to optimality [12].

The task of layout generation is generally computationally complex and the search space gets bigger if the reconfigurable slots are soft modules which means these modules are configured only by the area and an interval for the aspect ratio [12]. The optimal layout generation approach draws inspiration from a strategy created for organizing modules on a chip die [24]. The authors work [24] uses a hybrid Integer Linear Programming paradigm to define the problem of putting blocks in a rectangular chip area. This model served as the starting point, but the authors have made some adjustments to meet specific requirements [12].

Here in the optimal layout generation approach authors have used preplaced blocks and all the reconfigurable slots as soft modules. As previously stated, soft modules are characterized solely by their required area, which is precisely the information available for positioning the reconfigurable slots on the FPGA fabric. On the other hand, preplaced blocks are important since these blocks are used to mark the FPGA regions that do not contain any micro slots, e.g., areas where an FPGA's processing system is located. By using this technique now the authors can define or select a specific FPGA architecture. For placing the reconfigurable slots optimally on the FPGA fabric the authors have defined the geometrical position of each slot S_j in terms of its lower-left corner (x_i, y_i) width w_i and height h_i. Here the main constraints are slots should not overlap, and they should not be placed outside the chip area with width W and height H.

Binary variables p_{ij} and q_{ij} has been added by the authors for each slot to impose these limitations. These variables make sure that only one of the given inequalities holds, preventing the slots from overlapping and placing them inside the chip area [12].

Below are the following inequalities:

$$\begin{array}{ll} x_i + w_i \leq x_j + W\left(p_{ij} + q_{ij}\right), & \text{slot i to the left of slot j} \\ x_i - w_j \geq x_j - W\left(1 - p_{ij} + q_{ij}\right), & \text{slot i to the right of slot j} \\ y_i + h_i \leq y_j + H\left(1 + p_{ij} - q_{ij}\right), & \text{slot i below slot j} \\ y_i - h_j \geq x_j - H\left(2 - p_{ij} - q_{ij}\right), & \text{slot i above slot j} \end{array} \tag{1}$$

The authors have also introduced constraints to avoid placing boxes outside the FPGA's area:

$$\begin{array}{l} x_i + w_i \leq W, \\ y_i + h_i \leq H. \end{array} \tag{2}$$

Here the objective of the QCP is to reduce the size of the rectangle which is area xy that encompasses all reconfigurable slots. Here instead of using a quadratic objective function, the authors have minimized the rectangle's $2x + 2y$ perimeter, which also implicitly minimized the area of the rectangle. In order to guarantee that each slot is located within the chip area, the previously stated constraints are refined as:

$$\begin{array}{l} x_i + w_i \leq x, \\ y_i + h_i \leq z. \end{array} \tag{3}$$

For soft modules which are only defined by the required area A_i, the authors have included a quadratic constraint that ensures the width and height of each slot are large enough to accommodate the given area:

$$wi \cdot hi \geq Ai \quad \forall i \in \{1, \ldots, n\} \tag{4}$$

This model provides a flexible and efficient way to optimize the placement of reconfigurable slots and ensures that they are placed inside the chip area while minimizing the rectangle's perimeter. This technique is feasible only for small sets of reconfigurable slots[12].

2.2.4 Simulated Annealing

Simulated Annealing is a popular metaheuristic optimization algorithm that is inspired by the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy. It is a good approach for solving the problem of layout generation of FPGA because it can produce high-quality placements while accommodating various constraints [27] [5]. It has been widely used in solving combinatorial optimization problems [27], including the problem of layout generation for FPGAs. Its key benefit is that it makes it simple to incorporate an optimization goal into the objective function. In the case of Layout generation, the algorithm starts with an initial solution which is also known as an initial layout. The initial layout can be layout with random placements of the reconfigurable slots on the FPGA layout ensuring some constraints and then the algorithm iteratively improves the initial solution through random changes and accepting changes that lead to better solutions[27]. Given an initial solution S, the SA algorithm tries to find the globally optimal floorplan solution with the lowest cost. Many greedy approaches might get stuck in the locally optimal solutions. Since it iteratively searches for a neighboring solution with a lower cost than the current cost. Once the greedy approach gets to a locally optimal solution it is very hard to

escape from that point, because all the neighboring solutions have a higher cost than the optimal solution [7]. SA approach does not work in that way. It adopts a hill-climbing technique to escape from the local optimum solution. SA provides a non-zero probability to move from the current solution to the neighboring solution with a higher cost. With this uphill capability, it is possible that the SA can reach the global optimum solution G no matter what the initial solution or initial layout looks like. The algorithm accepts the worse solutions (layout with a higher cost than the current layout cost) during the optimization process with a decreasing probability over time. As mentioned above SA is the process of annealing in metallurgy, the algorithm starts with a high temperature and allows the search to explore the solution space. As the temperature decreases, the algorithm becomes less likely to accept worse solutions, and the search tries to focus on the best solution found so far. When the algorithm encounters a worse solution the probability of accepting that new solution is dependent on two factors [7]

1. The magnitude of the uphill move

2. The annealing temperature

This can be defined as

$$\text{Prob}\left(S \rightarrow S'\right) = \begin{cases} 1 \text{ if } \Delta C \leq 0 (\text{ down-hill move }) \\ e^{\Delta C/T} \text{ if } \Delta C > 0 (\text{ up-hill move }) \end{cases} \tag{5}$$

Here $\Delta C = cost(S') - cost(S)$, and T is current temperature. Every downhill move is accepted but the probability of accepting an uphill move depends on the magnitude (cost difference between the new layout and current layout) and the annealing temperature. As the optimization process goes on the temperature decreases by a certain constant or it can be dynamic or adaptive. Generally the simple annealing schedules looks like $T = T_0, T_1, T_2, ...$ and $T_i = r_i T_{i-1}, r < 1$. Here r is the rate at which temperature decreases in every loop. At each temperature, the algorithm perturbs the current solution to search for a number of neighboring solutions for k times, where k is a user-defined value and keeps the best solution found so far. This process continues until the temperature reaches the frozen state or a predefined termination condition is reached. Once it reaches this point. The algorithm returns the best-founded solution so far. The initial temperature, the cooling schedule, and the acceptance probability functions are the three main variables in Simulated Annealing. The trade-off between exploration and exploitation during the search can be managed by adjusting these parameters. Overall, Simulated Annealing is a flexible optimization method that has been used to solve a variety of issues, including layout generation [27] [20].

Below you can find the general pseudo-code for the Simulated Annealing algorithm

Algorithm 1 Simulated Annealing Algorithm

Require: Set T_{min} , Set max iterations
Ensure: Best layout found S_{Best}
Get initial floorplan S; $S_{Best} = S$;
Set initial temperature $T_0 > 0$;
while $T > T_{min}$ **do**
    for $ite = 1$ to max iterations **do**
        Perturb the initial floorplan S to generate neighboring floorplan S';
        $\Delta C = cost(S') - cost(S)$
        if $\Delta C \leq 0$ or $Random < e^{\Delta C/T}$ **then**
            Set $S = S'$
        end if
        if $cost(S_{Best}) > cost(S)$ **then**
            Set $S_{Best} = S$
        end if
    end for
    T = rT; //reduce temperature
end while
return S_{Best}

2.2.5 Evolutionary Strategy

ES is an optimization algorithm that is based on the ideas of natural evolution, where individuals with favorable traits are more likely to survive and reproduce. It is an iterative process that uses the concepts of population, fitness, selection, and mutation to generate a set of candidate solutions that evolve towards an optimal solution.

It is a population-based optimization technique that belongs to the border class of evolutionary algorithms. ES is used in various fields such as engineering, economics, medicine, and computer science to solve problems that are too complex for traditional optimization methods. It is particularly useful when dealing with non-linear, non-differentiable, and multi-objective problems where the search space is vast and the optimal solution is not well-defined.

Using ES in layout generation or FPGA placement has several benefits. The ability of ES to produce high-quality solutions by examining a wide search space of potential placements is one of its main advantages [26]. ES also has the ability to handle high-dimensional optimization problems, such as those encountered in FPGA layout generation. Evolutionary algorithms including evolutionary strategy, are heuristic-based approaches to solve the problems that cannot be solved in polynomial time, such as classically NP-Hard problems.

The EA process starts with a population of potential solutions that are generated at random and are each represented by a set of parameters. The fitness function evaluates each potential solution's effectiveness in solving the optimization problem. The fitness function determines how well a solution solves the problem, and the goal is to maximize or minimize it depending on the problem. The selection operator is used to choose the

best solutions from the population based on their fitness values. The individuals with higher fitness are then selected to create a new set of candidate solutions or offspring through mutation. The algorithm goes through this process several times until it finds a satisfactory solution or when the stopping criterion is met [15].

EA has a number of benefits over conventional optimization techniques. Being a population-based approach that can search the entire solution space rather than being constrained to a single point is one of its main advantages. This enables it to find better solutions in less time compared to other methods. EA can also handle multiple objectives by using a weighted sum of objectives or Pareto optimization to find a set of optimal solutions that trade-off between objectives. another advantage of EA is its ability to handle noisy and incomplete data by using robust fitness functions and adaptive mutation operators. This makes it suitable for problems in which the objective function is noisy, the data is incomplete, or the problem constraints are uncertain.

The population size, mutation strength, and termination criteria are the main variables when working with ES. ES requires careful tuning of its parameters, such as the population size and mutation strength, to obtain the best results. Additionally, the termination criteria must be chosen appropriately to avoid premature convergence or excessively long computation times. It should be noted that ES, like other optimization algorithms, has its limitations. It is not guaranteed to find the global optimum for all optimization problems, and it may get stuck in local optima. However, ES has been shown to be effective in a wide range of optimization problems, including those encountered in layout generation and FPGA placement.

Below is the general pseudo-code for the Evolutionary Strategy algorithm

Algorithm 2 Evolutionary Strategy Algorithm

1: Initialize the population of $population\_size$ chromosomes with random values.
2: Evaluate the fitness of each chromosome in the population.
3: **for** $ite = 1$ to $num\_generations$ **do**
4: Sample a parent using tournament selection with $tournament\_size$
5: Mutate the selected parent with probability $mutation\_rate$ to create offsprings
6: Replace current population with new offsprings.
7: Evaluate the fitness of the new population
8: **end for**
9: Return the chromosome in the population with the best fitness value

3 Optimization Problem

In this chapter, our primary focus revolves around addressing the optimization problem. We will begin by providing an overview of the optimization problem, followed by an exploration of floorplan representation and the factors that contribute to the quality of placement. Our aim is to solve the optimization problem using SA and ES algorithms. Throughout the chapter, we will delve into the details of SA and ES approaches employed in this work, shedding light on the key functions and operations involved in these algorithms.

3.1 Overview

The floorplan optimization problem can be also stated as a rectangular packing problem. The most visible characteristic of the rectangular packing problem is that the shape of the block and board is rectangular [23]. Here, the floorplan optimization problem F can be stated as follows: Let $S = S_1, S_2, ..., S_n$ be a set of reconfigurable slots that we get from slot creation and task assignment step from the 2D slot-based reconfiguration model. These reconfigurable slots are associated with respective areas a_i, $1 \leq i \leq n$ and have no connection between them. These reconfigurable slots are converted into rectangular modules whose width w_i and height h_i are not fixed. These reconfigurable slots can also be called soft modules.

Let (x_i, y_i) be the coordinates of the bottom left corner of the module S_i, $1 \leq i \leq n$, on the chip. The optimization problem is a placement of (x_i, y_i) of each module S_i, $1 \leq i \leq n$, such that no two modules overlap with each other and also cannot be placed outside the FPGA boundary. The goal of the optimization problem is to minimize the total area of the Minimum Bounding Rectangle (MBR) that encloses all the modules while satisfying placement constraints.

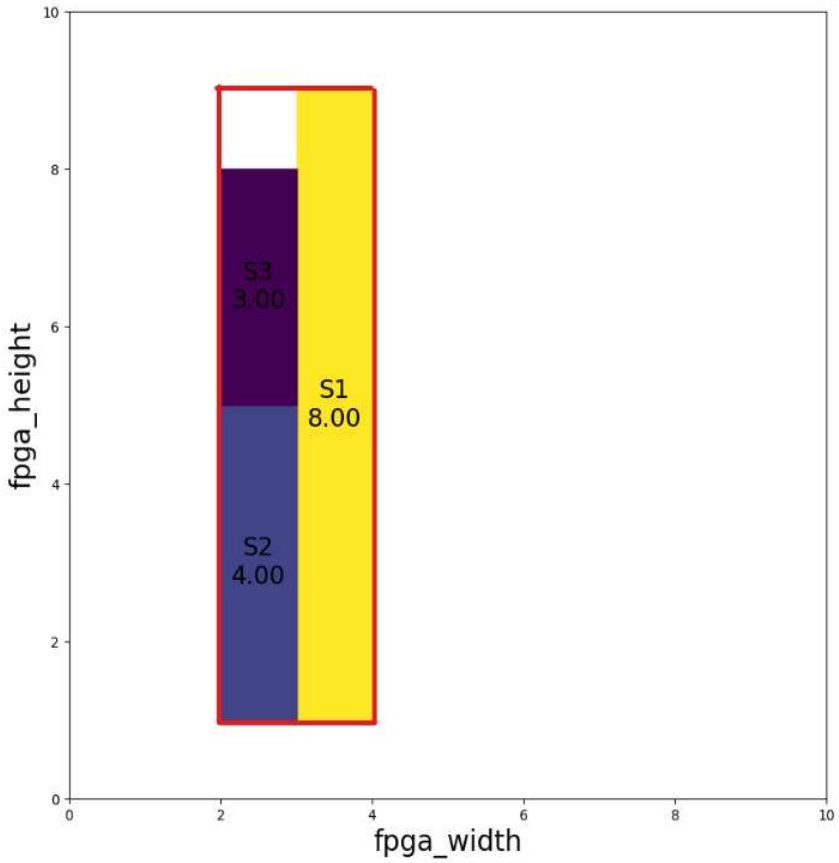
$$F(x) = minimize(area\_MBR) \tag{1}$$

Here, x is the layout and $area\_MBR$ is the total area of the MBR.

Figure 3.1 shows the goal of the optimization problem. Figure 3.1a shows the input that 3 reconfigurable slots with block name and required area and Figure 3.1b shows the following shaped slots are created: 1×8 for S_1, 1×4 for S_2, and 1×3 for S_3 and then placed these slots on FPGA without overlapping with each other and also not paced out of the boundaries of FPGA. We need to find the MBR area here it is indicated with a red line. The MBR area for Figure 3.1 is 16 micro slots

Block Name	Area
Block1	8
Block2	4
Block3	3

(a)



(b)

Figure 3.1: 3 reconfigurable slots as input for optimization problem (a) and after placing given reconfigurable slots on FPGA. Here red line denotes the Minimum Bounding Rectangle area (b)

.

We will consider a sample test data for solving the optimization problem. By using this data we will explore different functionalities of the algorithms.

Block Name	Area
Block1	12
Block2	10
Block3	6

Table 3.1: Sample test data for Optimization problem

In Table 3.1, there are three slots that need to be placed on an FPGA. To facilitate placement, these slots will be represented as rectangular modules, which will then be positioned on the surface of the FPGA.

Before discussing the SA and ES approach for the optimization problem first we will go through the floorplan representation and quality of the placement function we used in SA and ES algorithms.

3.2 Floorplan Representation

Floorplan representation is very important while working with the SA and ES algorithm. Floorplan representation provides the geometrical information regarding the blocks that are on the FPGA layout. It not only represents the solution but also enables for optimization of the floorplan which in turn helps in finding the floorplan with the lowest cost in the solution space.

There are two kinds of floorplans. They are slicing floorplans and non-slicing floorplans. A slicing floorplan is obtained by repetitively cutting the floorplan horizontally or vertically, whereas this can't be possible in a non-slicing floorplan[7]. The technique used in this paper is a non-slicing floorplan.

There are many types of floorplan representation such as B* trees, sequence pairs, polish expressions and etc. These representations are very useful when we consider the connection between the modules.

In this paper, the reconfigurable slots have no connection to the other modules. So, a coordinate-based floorplan representation is used where each block is shown as a rectangle, with its dimensions(w, h) and bottom-left coordinate (x, y). In situations where there are no connections between modules, this representation is very helpful.

The coordinate-based system, which makes use of the universally known Cartesian coordinate system, offers a very simple and intuitive way to represent a floorplan. Each block in the floorplan is represented using its dimensions, width w and height h, and its bottom left coordinate (x, y). The placement of each block is determined by its bottom left coordinate. The bottom-left corner's coordinates can be used for determining the location of the entire block. The coordinate-based system provides a quick visualization of the layout of the chip or electronic device. Additionally, it makes it simple to specify and check constraints.

The floorplan representation for the previously mentioned example 3.1, in SA algorithm appears as follows:
Rectangle(xy=(3, 2), width=3, height=4)
Rectangle(xy=(1, 1), width=2, height=5)
Rectangle(xy=(7, 1), width=3, height=2)

Figure 3.2 shows the initial layout of the above-considered example. The 3 blocks are placed randomly placed on the FPGA layout. Block1 is at (3,2), Block2 is at (1,1) and Block3 is at (7,1). The layout follows the constraints that blocks should not be overlapping with other blocks and also should not be placed outside the FPGA.

The ES algorithm starts the process by creating a population. Each member of the

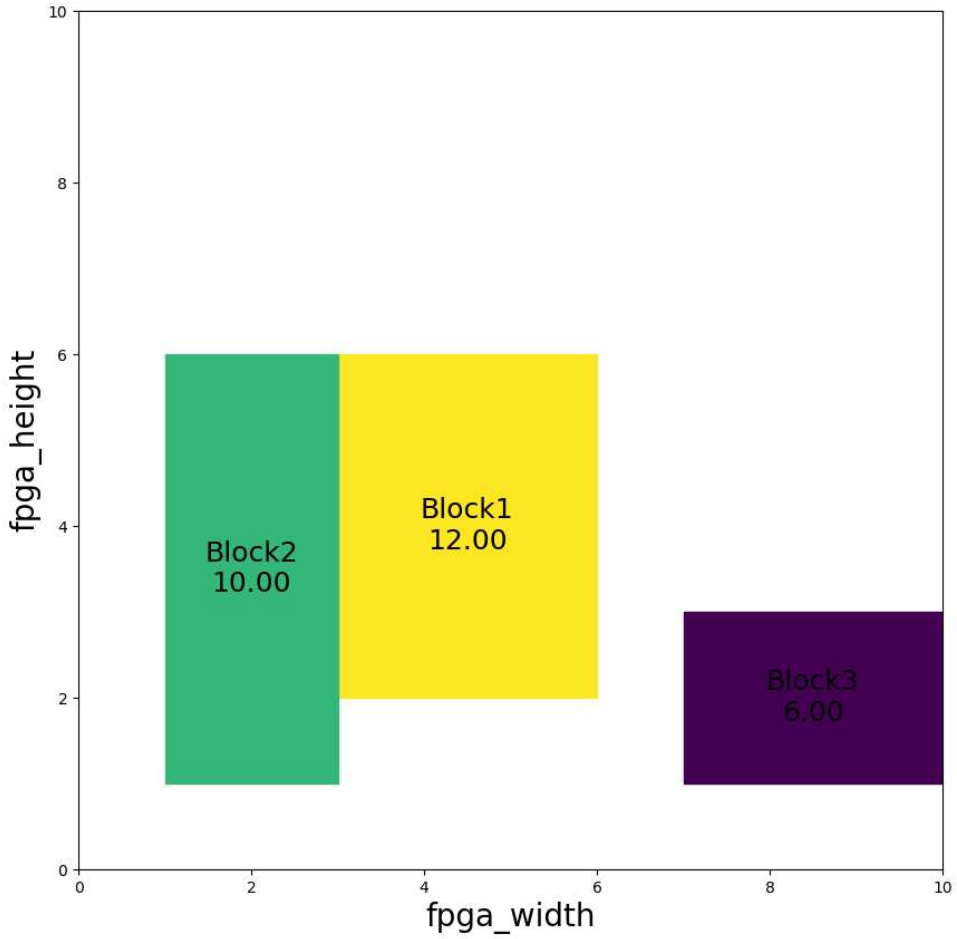


Figure 3.2: Initial layout for the sample data from SA.

population is a candidate solution to the given problem. Each candidate solution is called a chromosome. Each chromosome is encoded into a fixed-length string or sequence of genes, where each gene represents a specific attribute of the solution. Here, a chromosome representation typically encodes the position and size of the rectangular modules that need to be placed on an FPGA surface. A chromosome represents the layout of the FPGA and genes in the chromosome represent the rectangular modules. In a chromosome, each gene is represented using a tuple that includes five elements.

1. The name of the reconfigurable slot a_i
2. The x coordinate of the slot x_i
3. The y coordinate of the slot y_i
4. The width of the slot w_i

5. The height of the slot h_i

Let $S = S1, S2, ..., Sn$ be the set of reconfigurable slots. Let a_i be the name of the reconfigurable slot S_i, $1 \leq i \leq n$. Let x_i and y_i be the coordinates of the bottom-left corner of slot S_i, $1 \leq i \leq n$ on the FPGA fabric and w_i and h_i, $1 \leq i \leq n$ be the width and height of the slot. The chromosome can be represented in the following way:

$[(a_i, x_i, y_i, w_i, h_i), (a_{i+1}, x_{i+1}, y_{i+1}, w_{i+1}, h_{i+1}).....(a_{i+n}, x_{i+n}, y_{i+n}, w_{i+n}, h_{i+n})]$,$1 \leq i \leq n$

The floorplan representation for the previously mentioned example 3.1, in ES algorithm appears as follows:

[('Block1', 5, 1, 2, 6), ('Block2', 3, 8, 5, 2), ('Block3', 7, 4, 2, 3)]

Figure 3.3 shows the initial chromosome for the selected example. Here 3 genes are randomly placed on the FPGA surface. gene1 which is Block1 is located at (5,1) taking 2×6 as the shape, gene2 that is Block2 is positioned at (3,8) takes 5×2, gene3 that is Block3 is located at (7,4) and took 2×3as the shape.

The above-mentioned SA and ES floorplan representation techniques are used in solving both the optimization and the decision problem.

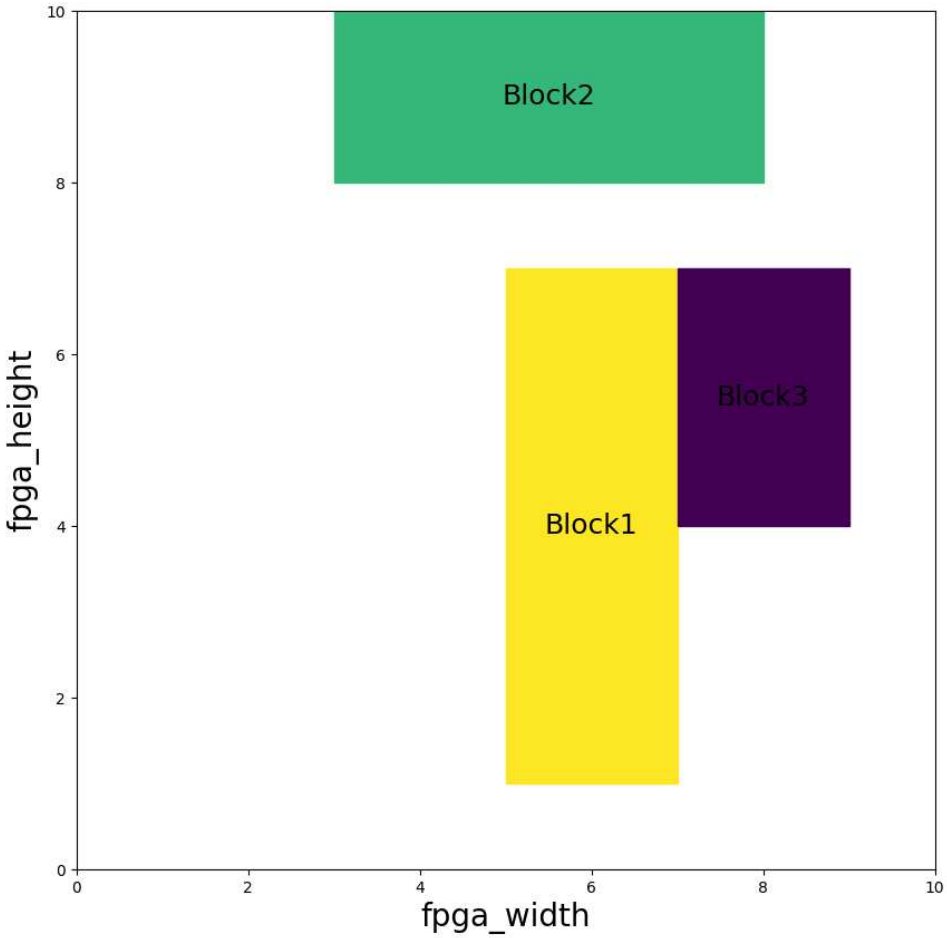


Figure 3.3: Initial chromosome visualization for the selected example from ES.

3.3 Quality of Placement

The quality of the output states the cost and fitness function of SA and ES algorithms. The cost and fitness function plays an important role in the performance of the SA and ES algorithms. In SA and ES algorithms, the cost or the fitness function is a measure of how well a particular layout of blocks satisfies the constraints of the problem. Both functions are used to assign a numerical value to the given layout, which indicates how well the solution satisfies the given criteria. In this work, the cost and fitness function is designed to minimize both the overall area occupied by the reconfigurable slots on the layout and the amount of dead space within the MBR which eventually reduces the total area used by the reconfigurable slots on the FPGA.

The floorplan area is directly dependent on the chip silicon cost which means the higher the floorplan area, the higher the silicon cost. Dead space, or white space, refers to the unoccupied area within the bounding rectangle that is not covered by any module [7].

For instance, In the given layout the MBR represents the smallest rectangle that encloses all the modules in the layout. To find the MBR for the given layout. We need to find the minimum and maximum of x_i and y_i coordinates among all the reconfigurable slots.

1. The bottom-left corner of the MBR is $(minimum(x_i), minimum(y_i)$.
2. The top-right corner of the MBR is $(maximum(x_i), maximum(y_i)$.

Then the area can be found using

$$AreaofMBR = (maximum(x_i) - minimum(x_i)) * (maximum(y_i) - minimum(y_i)) \quad (2)$$

In this work, our primary objective is to minimize the MBR area of the floorplan. To achieve this, we have devised a cost function that incorporates both the MBR area and the deadspace within the MBR. By including deadspace in the cost function, we actively encourage the optimization algorithm to reduce unused or wasted space within the MBR. This approach promotes efficient resource utilization by effectively utilizing the available area and has the potential to yield more optimized solutions.

The final floorplan cost is considered as the area of the MBR plus a penalty for the total dead space in the MBR. This technique resulted in a more efficient and compact design. The dead space is calculated by finding the MBR area of the layout minus the total required area of the modules that need to be placed. It can be computed using the following formula:

$$Deadspace = Area(MBR) - Area(totalmodules) \quad (3)$$

The penalty factor used here is flexible. By using this technique we can optimize the FPGA layout not only to minimize the MBR area but also to minimize the dead space in the FPGA layout. By minimizing both factors, the optimized layout can improve the overall performance of the FPGA while reducing the costs associated with the silicon area. The final fitness with MBR and dead space is as follows:

$$Cost(layout) = AreaofMBR + \alpha * Deadspace \quad (4)$$

The cost in SA and the fitness in ES use the penalty factor α as a weighting factor to balance the significance of reducing the MBR area and decreasing the MBR's dead space. The penalty factor is used to regulate the algorithm's level of preference between minimizing dead space and minimizing MBR area. A lower value of α emphasizes minimizing the MBR area while a higher value emphasizes minimizing the dead space. we have used 0.1 as the value for α. This approach results in an optimized layout that balances the trade-offs between layout compactness and efficiency, leading to improved performance and reduced costs.

3.4 Simulated Annealing Approach

SA is used as an independent algorithm or as an enhancement step of analytical placement algorithms. It is very useful because of its significant advantage of easily incorporating the optimization goal into an objective function[7]. To apply the SA for floorplan, it needs to encode a floorplan as a solution called floorplan representation which provides information regarding the geometric positions of the blocks on the floorplan. The algorithm starts

with an initial solution which is also known as an initial layout. The initial layout can be layout with random placements of the reconfigurable slots on the FPGA layout ensuring some constraints and then the algorithm iteratively improves the initial solution through random changes and accepting changes that lead to better solutions[27].

These are the main functions that one needs to concentrate on when working with SA: (1) Floorplan Representation, (2) Cost Function, (3) Solution Space, (4) Neighborhood Structure, and (5) Annealing Schedule. We have already discussed the floorplan representation and cost function in the above sections. Further, we will go through the remaining functions.

3.4.1 Solution Space

The solution space contains all possible solutions that can be formed using given rectangular soft modules in the given width W and height H of the FPGA. It contains all possible rectangular configurations of soft modules within a given layout area. Each configuration can be represented using a set of coordinates that specify the position and orientation of the block. Since the modules are soft, which means that it can change their height and width by keeping the same module area [7]. The solution space also includes the variation in the dimensions and aspect ratio of the modules. The size and aspect ratio of each module is very effective for solution space. For instance, if all modules have the same size and aspect ratio then there can be a chance that the solution space is limited to the variations in the placement and orientation. On the other hand, if the modules can vary in their aspect ratio the solution space can become much bigger, allowing a greater range of possible layouts.

In this work, the reconfigurable slots S_i change their height h_i and width w_i in between the given aspect ratio ranges. Every slot should have dimensions that fall in the given aspect ratio bounds and also maintain the given module area.

Here for example let us consider there is a reconfigurable slot with a required width given as 12 micro slots. As mentioned above every reconfigurable slot is converted into a rectangular shape before placing on an FPGA layout or fabric. Figure 3.4 shows all possible shapes with area = 12. They are $(1, 12), (12, 1), (2, 6), (6, 2), (3, 4), and (4, 3)$. Since the slots can vary their aspect ratio the solution space has so many different possible shapes for each slot. It is also made sure that each slot has the same module area as given.

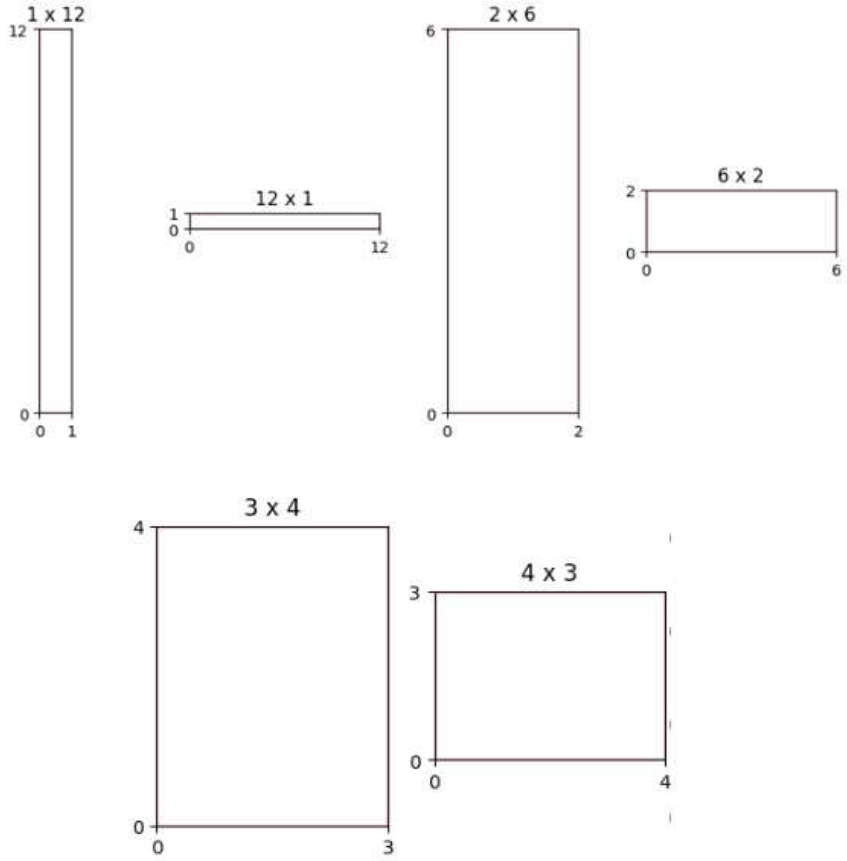


Figure 3.4: Different possible shape with area 12.

We have also implemented a trick aimed at improving the placement of blocks on the FPGA. This technique involves a clever strategy that increases the total number of shapes for certain blocks. Specifically, we apply this strategy to blocks whose area is a prime number or can only be formed by two rectangular shapes.

For instance, let us consider Block3, which requires an area of 3 units. Since 3 is a prime number, it can be transformed into either a shape of 1 unit width and 3 units height, or a shape of 3 units width and 1 unit height. To enhance the placement quality, we introduce a modification by incrementing the area of Block3 by adding +1. As a result, Block3 now has additional rectangular shapes available.

By employing this trick, we effectively increase the number of shape options for blocks that have restricted area configurations. This expanded flexibility in shape selection contributes to generating improved block placements on the FPGA.

3.4.2 Neighborhood Structure

Given an initial solution, the algorithm can perturb it to obtain a new "neighboring" solution. The perturbation plays a vital role in finding the desired solution or layout[7]. In this work, there are two main types of moves have been considered to get a neighboring solution.

1. Changing the shape of the module

2. Changing the location of the module

We will compare the initial layout for the example problem we have considered. Then, we will explore how the layout changes when we apply various moves in the neighborhood structure. Below we will compare the initial layout provided in Figure3.2 with the resulting updated layouts after applying neighborhood functions. By doing so, we can gain a better understanding of the optimization process and how the different moves affect the solution.

An approach to improve the layout generated by the SA algorithm is to change the shape of the module. Changing the shape of the module involves randomly selecting a module from the layout and changing its shape such that the shape has the aspect ratio in the given aspect ratio bounds and also maintaining the same module area. Here the aspect ratio is defined as the height divided by the width of the selected module. Changing the shape of the module also includes the rotation of the module since the algorithm considers all possible shapes for the given module area. The new shape of the module is accepted such that it helps in reducing the bounding rectangle area. The bounding rectangle is the smallest rectangular area that can enclose all the modules in the layout. By decreasing the bounding rectangle area, the algorithm can improve the overall efficiency of the layout. This technique of changing the shape of the modules helps the algorithm to explore a larger search space and potentially find better solutions. This technique also introduces more diversity in the layouts and can help avoid getting stuck in local optima.

The initial layout for the considered example problem after applying the "change the shape of the module" move is depicted in Figure 3.5. Specifically, Block3 has changed its shape from 3×2 to 1×6, resulting in an updated layout reducing the MBR.

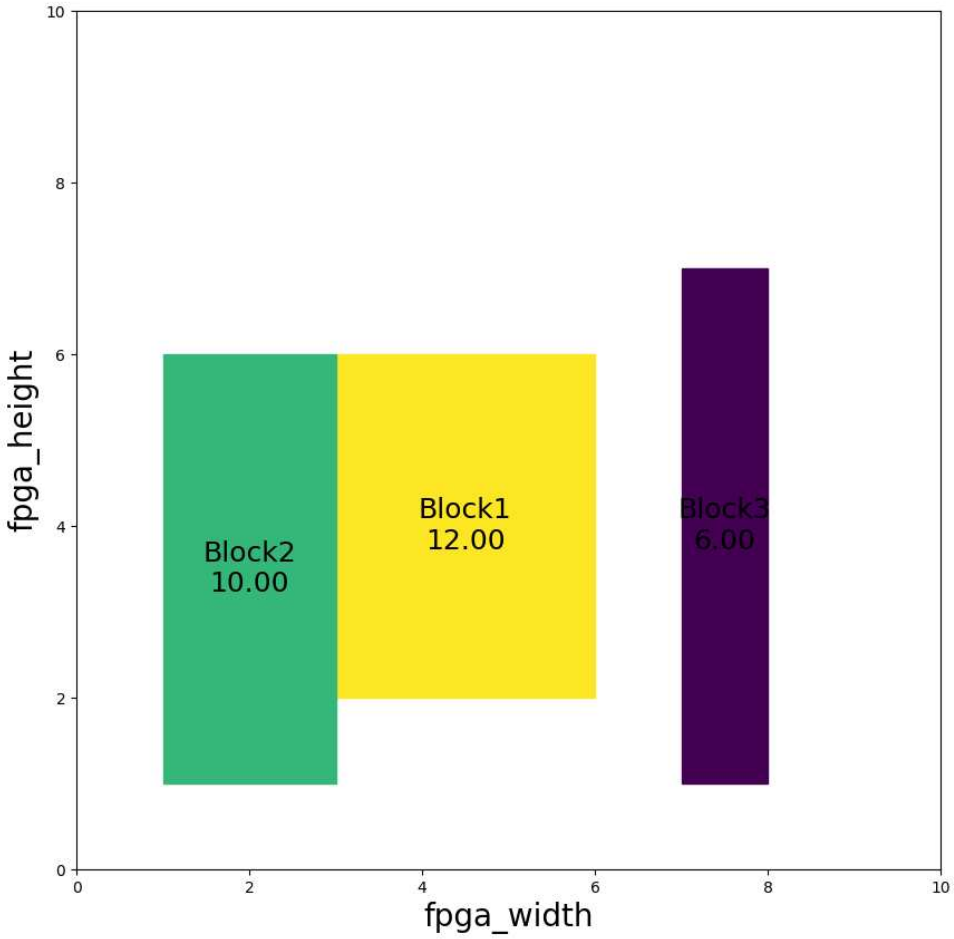


Figure 3.5: Initial layout after applying the "change the shape of the module" move.

The process of changing the location of the module involves randomly selecting a module from the layout and changing its position inside the current bounding rectangle area. The algorithm ensures that the new location does not violate the constraints of the problem, i.e., it should not go outside the bounds of the FPGA, and also ensures that the module does not overlap with the other modules. This technique of selecting a new location inside the current bounding rectangle of the layout helps minimize the cost function, leading to a better layout. This technique is useful in exploring various possible layouts and improving the overall quality of the solution.

Figure 3.6 depicts the initial layout after applying the "change the location of the module" move from the neighborhood structure. Specifically, here Block3 has changed its location from (7,1) to (6,1) on the FPGA surface.

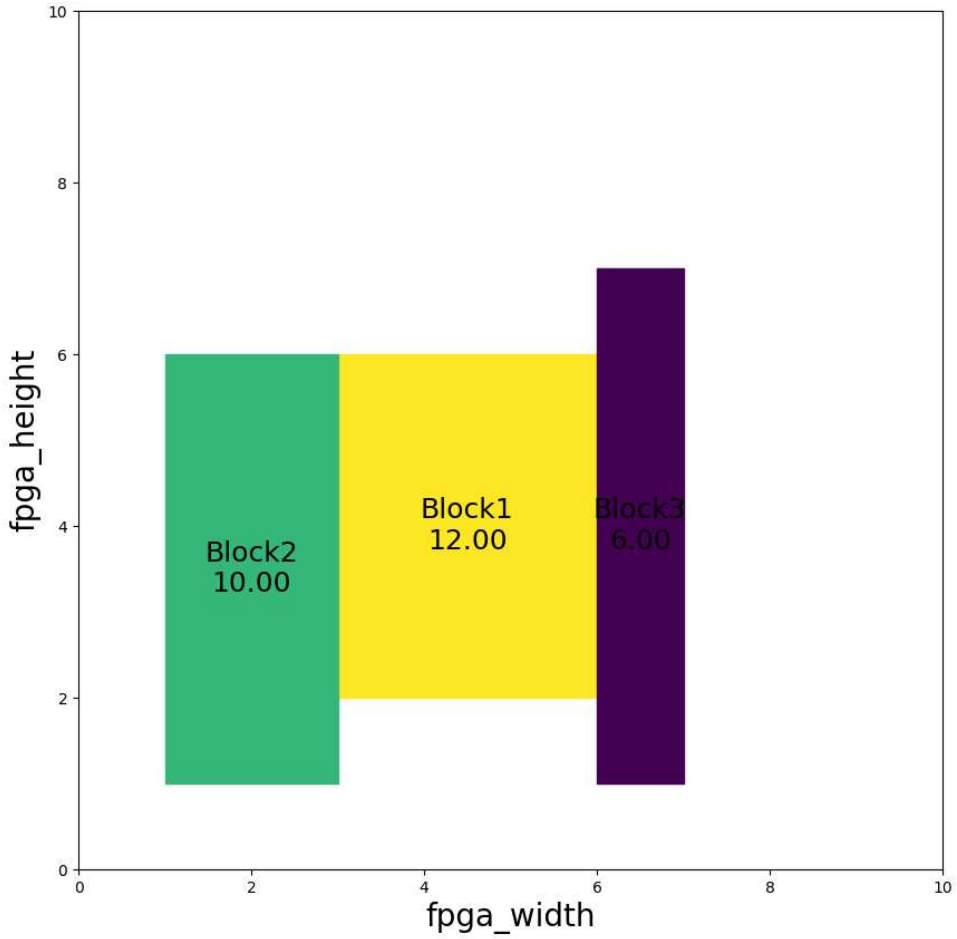


Figure 3.6: Initial layout after applying the "change the location of the module" move.

3.4.3 Annealing Schedule

The annealing schedule is one of the most important steps in SA. The annealing schedule determines how the temperature is decreased over time and is critical to the performance of the SA algorithm. In the classical simulated annealing algorithms the annealing schedule $T = T_0, T_1, T_2, ...$ and temperature declines according to the following function:

$$T_i = r_i T_{i-1}, r < 1 \tag{5}$$

Here T_i is the temperature in iteration i, and r is a coefficient that controls the rate of temperature declines over time. It is adjusted to balance the rate of exploration and exploitation [10]. The search begins with the initial temperature being set to a very high value so that the probability of accepting all perturbations is close to 1[7], allowing a higher chance of transition to a worse solution. By following this, the search will be able to get out of the local minima. However, the temperature continues to drop as the

search goes on, decreasing the likelihood of an uphill transition. Such a strategy might be advantageous if the local minima are located close to the search's starting point, but it might not result in a near-optimal solution if some local minima are found near the search's end at a temperature that is too low. So, to overcome this here in this work we have kept the r as adaptive.

There are many previous works [10] [9] that have shown that changing r into adaptive can give better solutions while working with SA. In this work, The cooling parameter r is initially set to a high value, enabling the algorithm to explore the solution space more thoroughly. As the algorithms progress the cooling rate r is decreased to reduce the rate of exploration and increase the rate of exploitation. The cooling parameter r is adaptive which means the r will differ in some iterations. For deciding the r value after every iteration we calculate the acceptance rate such that the total number of accepted moves is divided by the total iterations in the inner loop of SA. Here the acceptance rate r is calculated as follows:

$$Acceptance\,rate = No.\,of\,acceptedmoves/Total\,number\,of attempted\,moves \tag{6}$$

The value of r is raised to enable more exploration if the acceptance rate falls below a predetermined cutoff. The value of r is decreased to allow for greater exploitation if the acceptance rate rises above a particular threshold. This helps to fine-tune the solution and improve its quality. The larger the r, the shorter the annealing time [7].

The initial temperature parameter is very important since it controls the acceptance rate of uphill. If the initial temperature is too high then the algorithm might stuck in the local minima and if the initial temperature is too low then the algorithm does not have much time to explore the solution space. For these reasons setting the initial parameter is very important. For setting the initial temperature in this work, before starting the simulated annealing loop, we perturb the initial layout for a constant time to compute the average of all positive cost change Δ_{avg} [7]. Then the T_0 is initialized as follows:

$$T_0 = -\Delta_{avg}/\ln P \tag{7}$$

where P is the initial probability of accepting the higher-cost solution. In this attempt if every perturb gives a downhill then we set the T_0 value to high manually.

After considering all the above functions. The resulting algorithm will give the minimum bounding area required to place the reconfigurable slots. For the above-considered problem 3.1. The final bounding area is 30 micro slots. Figure 3.7 depicts the final layout by the SA algorithm for the above-considered problem 3.1.

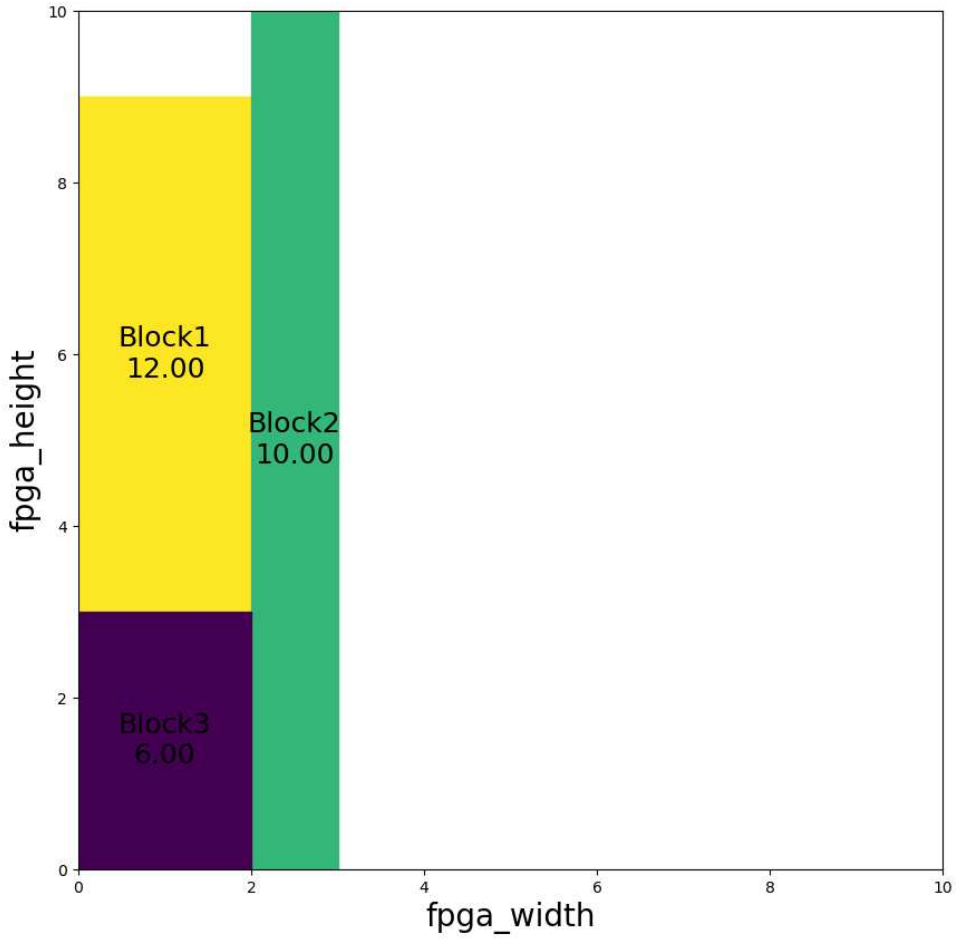


Figure 3.7: Final layout from SA algorithm for the above-considered problem3.1

3.4.4 Final Simulated Annealing Algorithm for Optimization Problem

Below is the final algorithm used in this work, for solving an optimization problem using SA.

Algorithm 3 Final Simulated Annealing algorithm for Optimization Problem

Require: Set T_{min} to lowest temperature, Set k to 100, Set Predefined value = 0.4, Set Random range to [0,1)

Ensure: Return the best layout found S_{Best}

Get initial floorplan S; $S_{Best} = S$
Set initial temperature $T_0 = -\Delta_{avg} / \ln P$
while $T > T_{min}$ **do**
    Set accepted attempts = 0
    Set total attempts = 0
    for $ite = 1$ to k **do**
        Perturb the initial floorplan S to generate neighboring floorplan S' by choosing between changing the shape of some blocks in the layout and changing the location of some blocks in the layout.
        $\Delta C = cost(S') - cost(S)$
        if $\Delta C \leq 0$ or $Random < e^{\Delta C/T}$ **then**
            Set $S = S'$
            accepted attempts += 1
        end if
        total attempts += 1
    end for
    if $cost(S_{Best}) > cost(S)$ **then**
        Set $S_{Best} = S$
    end if
    Compute Acceptance rate = accepted attempts / Total attempts
    if Acceptance rate < Predefined value **then**
        Increase r (Cooling rate)
    else
        Decrease r
    end if
    T = rT; //reduce temperature
    return S_{Best}
end while

3.5 Evolutionary Strategy Approach

Evolutionary Strategy (ES) is a class of Evolutionary Algorithms (EA) that uses a stochastic search strategy to find the optimal solution to a given problem. EAs, such as Genetic Algorithms (GAs) and Evolutionary Programming are popular methods used for solving optimization problems [11]. The main difference between the GA and ES is that ES does not depend on the crossover operator for evolving solutions [11]. As mentioned earlier ES works on the idea of natural evolution. It uses the mutation operator to generate new candidate solutions and improve upon existing ones.

In this work, a simple $(1 + \lambda)$ ES has been used. Here the "1" represents the single parent and the "λ" represents the number of offsprings that will be produced in every iteration using the mutation operator. The evolutionary operators, such as mutation is used to modify the chromosome and produce new candidate solutions. In every generation the best individuals from the current generation are directly carried over to the next generation without any modifications this technique is known as elitism. The main idea behind elitism is that the best individuals are likely to be near-optimal solutions and can be used to guide the search towards better solutions. However, there are also some drawbacks while using elitism since it can reduce the diversity in the population. To overcome this problem only a few percent of the individuals are selected from each generation. Next, we will examine the key functions that govern the parent selection and offspring generation processes.

3.5.1 Selection Process

The selection process is a crucial step in EA since it determines which candidate solution is selected further to generate the next generation off-springs. In this work tournament selection is selected as the selection function. Researchers have mentioned tournament selection is known as an established selection operator that has been employed in various problems [1]. Tournament selection involves selecting a random subset of individuals from the population and the individual with the highest fitness score from the subset is selected as the parent for the following generation [22]. The parameter named tournament size decides the size of the subset we select from the population. This parameter is used to adjust the selection pressure of the algorithm. Stronger selection pressure is created when the tournament size is larger because the fittest people are more likely to be chosen as parents.

3.5.2 Evolutionary Operators

Evolutionary operators play a very important role in Evolutionary algorithms. Usually, there are mutation and crossover operators which are used to generate the next generation from the selected parent. In this work, only the mutation operator is used to generate the offsprings. The mutation operator helps to explore the search space and generate diverse candidate solutions. It also helps the algorithm not to stuck at the local minima [26]. In this work, the mutation operator involves two ways of modifying the selected gene. The mutation operation starts by selecting a random gene in the chromosome and then applying one of the two modifications to the selected gene. Here are the two different ways.

1. Change the position of the gene

2. Change the shape of the gene

We will try to compare the initial layout shown in Figure 3.3 with the later visualizations to see how the layout changes when we apply various modifications using evolutionary operators.

Changing the position of the gene means randomly perturbing the x_i and y_i coordinates of the gene inside the current bounding rectangle area. This technique helps to reduce the minimum bounding rectangle area. This technique is useful in exploring various possible layouts and improving the overall quality of the solution. Here for perturbing the new x_i and y_i coordinates, there are two rules that need to be satisfied to accept the new x_i and y_i. Those rules are:

1. The new x_i and y_i coordinates should not be out of the boundaries of the given FPGA i.e., $x_i + w_i <= W$ and $y_i + h_i <= H$. Here w_i and h_i represent the width and height of the current gene and W, H represent the width and height of FPGA.

2. The gene should be placed in such a way that it also does not overlap with any other genes in the layout i.e., $(x_i + w_i <= x_j)OR(x_j + w_j <= x_i)OR(y_i + h_i <= y_j)OR(y_j + h_j <= y_i)$ where i and j are the indices of two distinct blocks, and the OR operator indicates that only one of the conditions needs to be true to satisfy the constraint. This constraint ensures that the bounding boxes of the two blocks do not overlap in the x or y direction, which means that the blocks themselves do not overlap.

Figure 3.8 depicts the layout after applying the "change the position of the gene" modification. In Figure 3.8 that the gene3 which is Block3 has changed its location from (7,4) to (3,4)

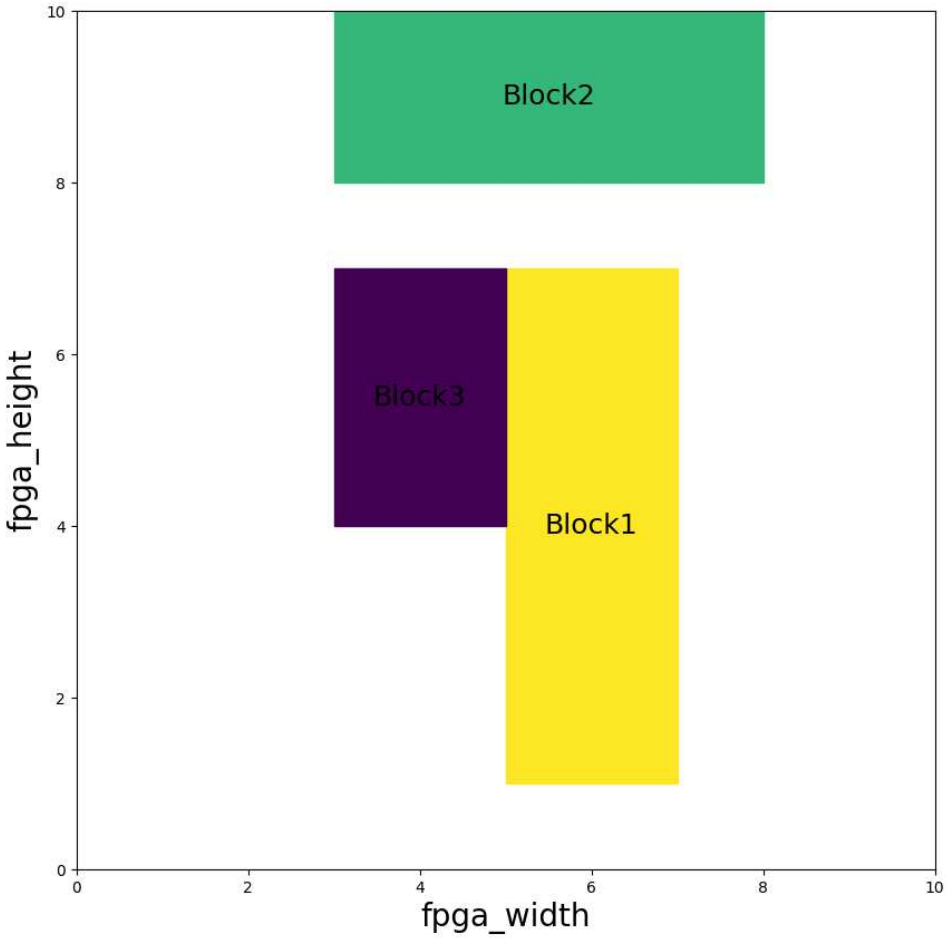


Figure 3.8: Initial chromosome after applying the "change the position of the gene" operator

On the other hand, changing the shape of the gene involves changing the w and h values of the gene. Since all the genes are considered as soft modules. We can change the width and height of the gene while also maintaining the same module area. Basically, it means

$$w_i * h_i = Area(gene_i) \tag{8}$$

the algorithm perturbs the w_i and h_i values of each block in the layout to find a better solution. To ensure that the new width and height values fall within a given aspect ratio range, the optimization algorithm imposes minimum and maximum aspect ratio bounds on each block. The minimum and maximum aspect ratio bounds are obtained by finding the minimum and maximum dimensions that can be possible with the given block area i.e., $minimum(w_i, h_i)$ and $maximum(w_i, h_i)$. For example, if a gene has an area of 100 micro slots, its minimum possible dimensions could be $(1, 100)$, and its maximum possible

dimensions could be $(10, 10)$.

By imposing these minimum and maximum aspect ratio bounds on each block, the optimization algorithm ensures that the new width and height values generated during the perturbation process are within the specified aspect ratio range. This helps to maintain the desired block aspect ratios in the final layout.

Figure 3.9 depicts the layout after applying the "change the shape of the gene" modification. In the figure, gene1 that is Block1 has changed shape from 2×6 to 3×4

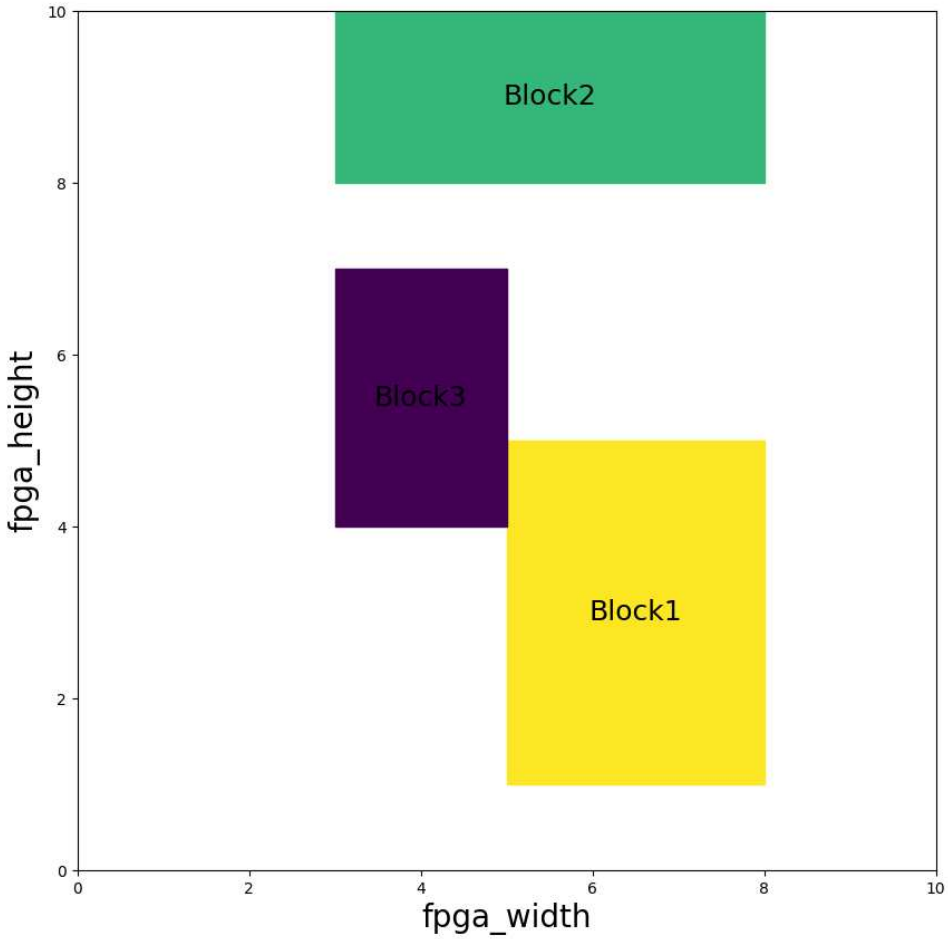


Figure 3.9: Initial chromosome after applying the "change the shape of the gene" operator

The mutation function also has a mutation rate that controls the frequency at which mutations are applied to the individuals in the population. Higher mutation rates encourage more exploratory searching because there is a greater likelihood that solutions will emerge at random that are new and varied. However, if the mutation rate is too low

then it might affect the algorithm to converge very slowly or become trapped in local optima. So, the mutation rate needs to be chosen by experimenting depending on the goal of the problem.

3.5.3 Algorithmic details

In this work, there are several key parameters that were carefully selected to ensure the effective performance of layout generation of reconfigurable slots.

- For the population size in this work is chosen as similar to the work [26]. In this work, the population size is set to 4 times to the total number of reconfigurable slots that need to be placed on FPGA. This was chosen because it was computationally feasible and provided enough diversity in the population to prevent early convergence
- The size of the tournament was set at twice the total number of reconfigurable slots or half the size of the population. While still allowing for some exploration of the potential solutions, this tournament selection method makes sure that the fittest individuals are more likely to be chosen for reproduction
- The number of generations was set to 150, which provided enough time for the algorithm to converge to a reasonable solution
- Finally, the mutation rate is set between 0.03 to 0.08. This mutation rate makes sure that the population has enough diversity to explore the solution space effectively while still allowing for a sufficient amount of convergence toward a sound solution

After applying all these above-mentioned functions with the given algorithm details the minimum bounding area for the mentioned problem 3.1 is 36 micro slots. Figure 3.10 depicts the final layout for the above-considered problem 3.1.

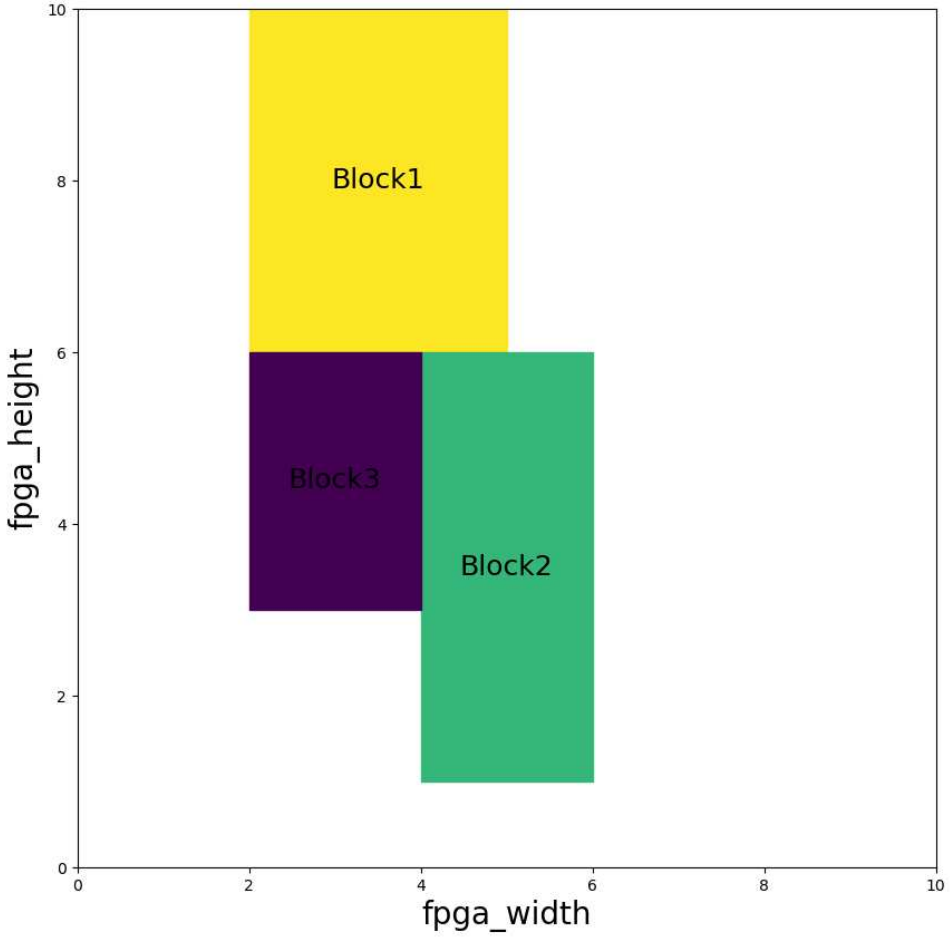


Figure 3.10: Final Layout from ES algorithm for the example

3.5.4 Final Evolutionary Strategy Algorithm for Optimization Problem

Below you can find the final Evolutionary Strategy algorithm that has been used in this work to solve the optimization problem.

Algorithm 4 Final Evolutionary Strategy Algorithm for Optimization Problem

Require: Set the $population\_size$ to 4 * the total number of blocks in layout, Set the $tournament\_size$ as 2 * the total number of blocks in layout. Set the $num\_offsprings$ to 90% of the $population\_size$, and Set $num\_generations$ to 150. Set the $mutation\_rate$ to 0.5.

Ensure: Return the chromosome with the best fitness value

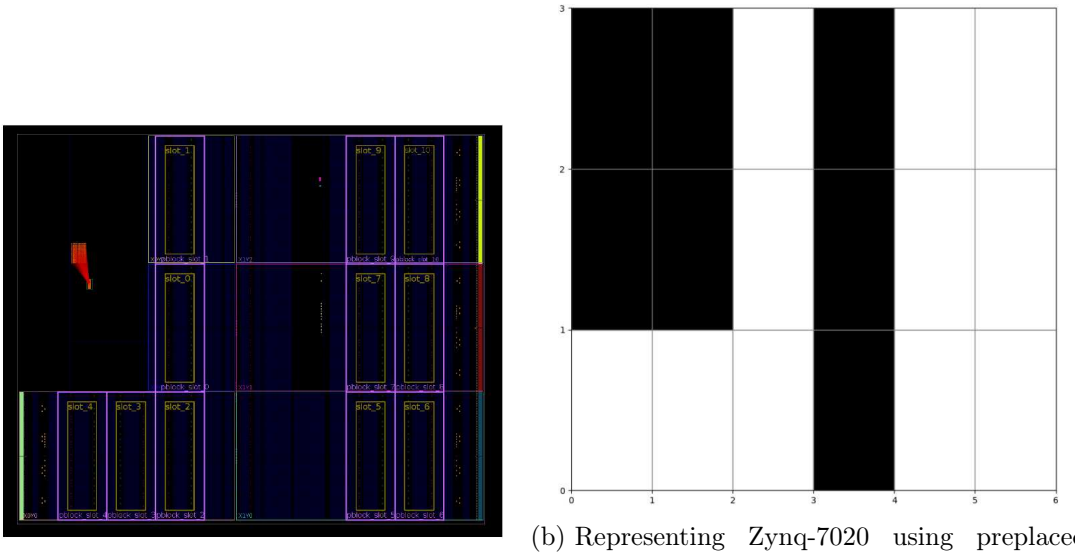
Initialize the population of $population\_size$ chromosomes with random values.
Evaluate the fitness of each chromosome in the population.
for $ite = 1$ to $num\_generations$ **do**
    Sort Initial population and Fitness scores in ascending order of fitness values.
    Select the top **10%** of sorted population as elites.
    for $ite = 1$ to $population\_size - elites$ **do**
        Sample a parent using tournament selection with $tournament\_size$
        Mutate the selected parent with probability $mutation\_rate$ to create offsprings
    end for
    Add Elites to the Offsprings
    Replace current population with new offsprings.
    Evaluate the fitness of the new population
end for
Return the chromosome in the population with the best fitness value

4 Decision Problem

In this chapter, our primary focus will be on solving the decision problem. We will begin with an overview of the decision problem, followed by an examination of floorplan representation and the factors influencing placement quality. To solve the decision problem we use SA and ES algorithms. This chapter will delve into the complexities of these algorithms, highlighting their main functions and how they are used in our work.

4.1 Overview

In the decision problem, there are preplaced blocks and reconfigurable slots from the slot creation and task assignment step from the 2D slot-based reconfiguration model are considered. The preplaced blocks are used to block the region where there are no micro slots, e.g., areas where FPGA's processing system is located. Figure 4.1 shows how preplaced blocks are used to represent the restricted area on FPGA which in turn represents the specific FPGA. The white space represents the available micro slots on the Xilinx Zynq-7020 FPGA and the black blocks represent the preplaced blocks.



(a) Micro slots for Zynq-7020

(b) Representing Zynq-7020 using preplaced blocks (black blocks)

Figure 4.1: How Zynq-7020 FPGA is represented in this work

The floorplan decision problem can be stated as follows: Let $B = B_1, B_2, ..., B_m$ be the preplaced blocks with specific coordinates (x_i, y_i) and specific dimensions (w_i, h_i) and $S = S_1, S_2, ..., S_n$ be a set of reconfigurable slots. These reconfigurable slots are associated with respective area a_i, $1 \leq i \leq n$ and have no connection between the slots. These reconfigurable slots will be converted into rectangular modules whose width w_i and

height h_i are not fixed. This kind of rectangular module can also be called as soft module which means the height h_i and width w_i can be changed while keeping the same module area. Let (x_i, y_i) be the coordinates of the bottom left corner of the reconfigurable slot S_i, $1 \leq i \leq n$, on the chip. The decision problem is to state whether the algorithm can find a feasible layout with reconfigurable slots where there is no overlapping between the modules and there is also no overlapping between the preplaced blocks. The goal of the decision problem is to reduce the overlaps between the slots. This can be stated as:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \max\left(0, S_i \cap S_j\right). \tag{1}$$

where $\mathbf{x}$ represents the layout, n is the total number of slots, S_i is the bounding box for block i, $S_i \cap S_j$ represents the intersection of bounding boxes S_i and S_j.

Let us consider an example data as shown in Table 4.1. Here, Block1 requires 3 micro slots, Block2 has 2 micro slot requirements, Block3 has 3 and Block4 has 2 micro slot requirements. Here the goal of the decision problem is to say whether these given reconfigurable slots can be placed on the given Xilinx Zynq 7020 FPGA or not.

Figure 4.2is the visualization of how these blocks can be placed on the FPGA. If all the given blocks are able to be placed on the given FPGA then the algorithm returns TRUE if not FALSE.

Before going through the SA and ES approach for the decision problem. First, we will discuss what is the floorplan representation used for both algorithms and what is the cost and fitness function used in the approaches.

Block name	Area
Block1	3
Block2	2
Block3	3
Block4	2

Table 4.1: sample decision data

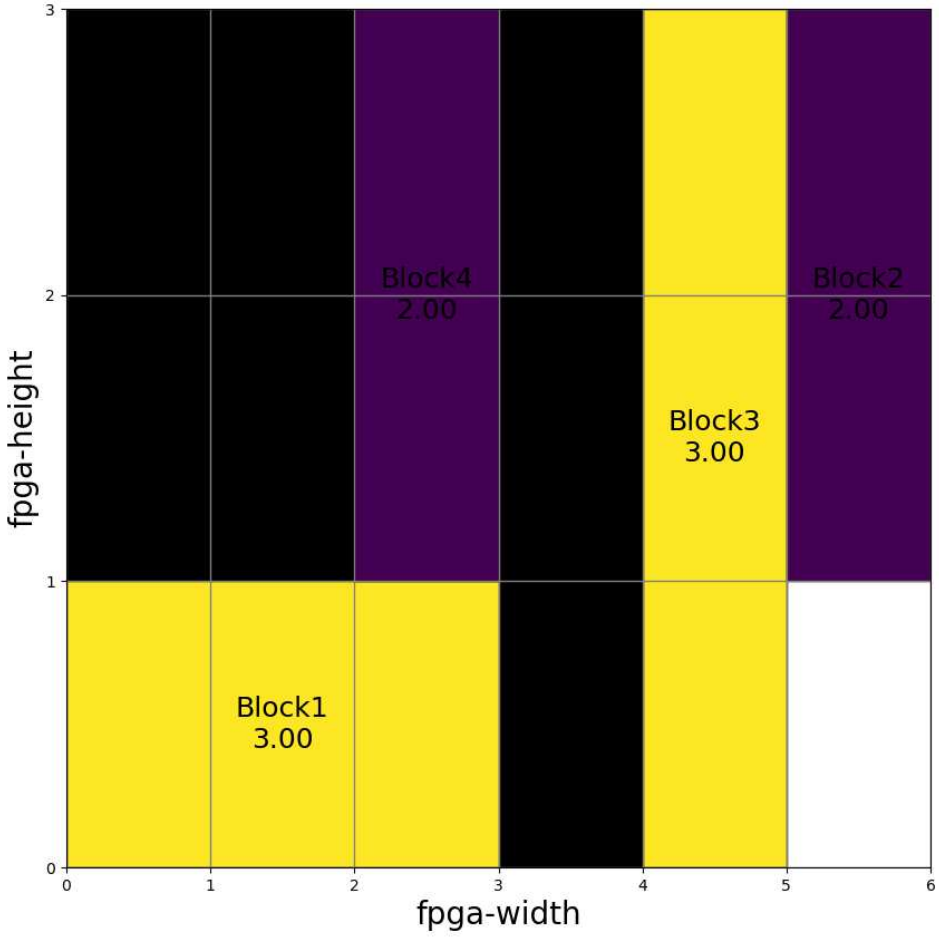


Figure 4.2: After placing reconfigurable slots on Xilinx Zynq-7020 FPGA device.

4.2 Floorplan Representation

The floorplan representation for the SA and ES algorithms to solve the decision problem is the same as the technique we used in solving the optimization problem. The concept regarding the floorplan representation can be found in subsection 3.2.

In SA the floorplan representation consists of all rectangular modules that are placed on the layout with the bottom left coordinate (x, y) and width w and height h. The algorithm ensures there is no overlap between the module and the preplaced blocks and also not placed outside the boundaries of FPGA.

The floorplan representation of SA for figure 4.2 looks like this:
Rectangle(xy=(0, 0), width=3, height=1)
Rectangle(xy=(5, 1), width=1, height=2)
Rectangle(xy=(4, 0), width=1, height=3)
Rectangle(xy=(2, 1), width=1, height=2)

Here each rectangle refers to a reconfigurable slot in the layout.

In ES each layout is represented as a chromosome. Each chromosome contains a number of genes. Here each gene represents the reconfigurable slot in the layout. A gene is represented using a tuple containing 5 elements of the slot. They are the name of the block, the x and y coordinates of the gene, and finally the width and height of the gene.

The floorplan representation of ES for figure 4.2 looks like this:
$[('Block1', 0, 0, 3, 1), ('Block4', 2, 1, 1, 2), ('Block3', 4, 0, 1, 3), ('Block2', 5, 1, 1, 2)]$

Here each tuple in the list refers to a reconfigurable slot in the layout.

4.3 Quality of Placement

In this section, we will go through the cost and fitness functions used in the SA and ES algorithms. In SA and ES algorithms, the cost or fitness function is a measure of how well a particular layout of blocks satisfies the constraints of the problem. Both functions are used to assign a numerical value to the given layout, which indicates how well the solution satisfies the given criteria. Prior researches [31], [17] suggested that the number of overlaps between the blocks can help in achieving a desired layout when placing the blocks on an FPGA.

In this work, the cost and fitness functions have been designed to penalize layouts with intersecting or overlapping blocks, by counting the number of overlaps between the reconfigurable slots.

The cost function can be expressed as follows for both the SA and ES algorithms:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \max\left(0, A_i \cap A_j\right). \tag{2}$$

where $\mathbf{x}$ represents the layout, n is the total number of blocks, A_i is the bounding box for block i, $A_i \cap A_j$ represents the intersection of bounding boxes A_i and A_j. The goal of both the SA and ES algorithms is to determine whether a feasible layout exists with the given reconfigurable slots and on selected Xilinx FPGA.

4.4 Simulated Annealing Approach

The SA starts the process by encoding a floorplan as a solution called initial layout. The initial layout can be layout with random placements of the reconfigurable slots on the FPGA layout. While placing the reconfigurable slots we have considered that they should not be overlapping with the preplaced blocks and also should not be placed outside the bounds of FPGA. But, there can be overlapping between the reconfigurable slots.

Solution space, Neighborhood structure, and Annealing schedule are the main function that is required while running an SA algorithm. All these functions are explained in the below sections.

4.4.1 Solution Space

As we discussed the solution space in subsection 3.4.1. This solution space represents the set of all possible solutions that can be derived from the given blocks. By referring to the insights and strategies discussed in that subsection, we can effectively explore the solution space and optimize our search for the best solution.

For the decision problem, we have also used the same trick we implemented in the optimization problem that increases the number of shapes for the given area. For example, if we consider 3 as the required area. Then there are only two rectangular shapes that can possible to make i.e., $(1, 3)$ and$(3, 1)$. In that situation, we have used a functionality where we had a $+1$ to the area which is prime and that has only two rectangular shapes. In the above case, this functionality will increase the area of 3 to 4 micro slots. Now, there are more than 2 shapes for the block, and also gets a square shape which is better than before shapes.

Solution space can differ with different types of blocks. If there are larger blocks or a large number of blocks can potentially increase the solution space since there will be many possible ways of arranging these blocks and also each block can change its shape which makes the solution space bigger and bigger. On the other hand, a smaller number of blocks or smaller block sizes can limit the solution space, potentially leading to fewer possible layouts. Since there are also constraints like the blocks cannot be placed on the preplaced blocks which is the area where FPGA processing units are located and also the blocks cannot be placed outside the FPGA. These two constraints can also limit the solution space eliminating many possible layouts. So, here we can say that a larger solution space does not necessarily mean that it is easier to find a good solution, as it may also contain a higher number of infeasible or poor-quality layouts.

4.4.2 Neighborhood Structure

The SA algorithm starts with encoding the initial solution or initial layout and then the algorithm tries to make changes to the initial layout to get better layouts. This process will be done using the neighborhood function. Finding the desired solution using perturbation is very important in SA [7]. In this decision problem of this work, there are two different types of moves that will generate the new neighborhood solution. They are

1. Changing the shape of the rectangular module
2. Changing the position of the rectangular module

These moves allow the algorithm to explore the solution space in a more diverse manner. These techniques are similar to the techniques we discussed in subsection 3.4.2. Below we will go through some of the important rules in the given moves.

In the first move, Changing the shape of the rectangular module, the algorithm randomly selects a reconfigurable slot S_i from the layout and then tries to change its shape (width and height). The new shape of the block should be in the given aspect ratio bounds. The min and max aspect ratio of the block is calculated using the minimum and maximum dimensions that are possible with the block area it also makes sure that the reconfigurable slot has the same module area as before i.e., $width_i * height_i = Area(S_i)$.

We can define the minimum and maximum possible dimensions of a block S_i as $w_{i_{min}}$ and $h_{i_{min}}$ for the minimum width and height, respectively. Similarly, $w_{i_{max}}$ and $h_{i_{max}}$

denote the maximum width and height, respectively. Based on these values, we can determine the aspect ratio bounds of the block as follows:

1. Find the aspect ratio of the minimum dimensions: $AR_{min} = w_{i_{min}}/h_{i_{min}}$.
2. Find the aspect ratio of the maximum dimensions: $AR_{max} = w_{i_{max}}/h_{i_{max}}$.
3. Set the aspect ratio bounds to be a range between AR_{min} and AR_{max}. We can represent this as: $(AR_{min} <= w_i/h_i <= AR_{max})$

Here in the work, we have preferred that if the shape of the block is similar to a square then it is a well-suited shape for that block since the square blocks mostly don't create dead space in the FPGA layout and have a more uniform layout which can be easily placed on the FPGA layout. The rotation of the blocks hasn't been considered as a separate move since changing the shape of the block already includes the rotated block shape too. The new shape of the block is selected such that the block fits in the given FPGA and does not overlap with the preplaced blocks. The visualization for the change of shape move can be found in figure3.5.

In the second move of the algorithm, a rectangular module, denoted as S_i, is randomly selected from the current layout. Its position, represented by the coordinates (x_i, y_i), is then changed to a new position (x_j, y_j) that satisfies two conditions. Firstly, the new position must not overlap with any preplaced blocks. Secondly, the new position must remain within the boundaries of the FPGA. It is possible for the module to overlap with other modules after the move. The visualization for the change of position can be found in figure 3.6

The chosen neighborhood structure allows the algorithm to explore a wide range of possible layouts by changing the shape and position of the rectangular modules.

4.4.3 Annealing Schedule

In the subsection referenced (3.4.3), the annealing schedule settings used in the SA algorithm for solving optimization problems are described. These settings include the choice of cooling parameter, the determination of the acceptance ratio, and the calculation of the initial temperature. The settings are also used for solving the decision problem.

The cooling parameter r plays a crucial role in controlling the rate at which the system transitions from exploring a wide range of solutions to exploiting the local search space. By keeping the cooling parameter adaptive, the algorithm dynamically adjusts the cooling schedule based on the acceptance ratio of moves.

The acceptance ratio, which is the ratio of accepted moves to attempted moves, helps in fine-tuning the exploration and exploitation balance during the annealing process. By monitoring this ratio, the algorithm can make informed decisions about the acceptance of uphill moves, thereby guiding the search towards better solutions.

Additionally, the initial temperature is set by perturbing the initial solution for a constant amount of time and observing the average of all positive cost changes. If every perturb gives a downhill then we set the initial temperature T_0 to high manually. This observation allows for the determination of an appropriate initial temperature that enables effective exploration of the solution space.

4.4.4 Final Simulated Annealing Algorithm for Decision Problem

The below-provided algorithm is the Final SA algorithm used in this work to solve the Decision problem.

Algorithm 5 Final Simulated Annealing Algorithm For Decision Problem

Require: Set T_{min} to the lowest temperature, Set k to 100, Set Predefined cutoff as 0.4, Set the Random range to [0, 1)

Ensure: Return the best layout found S_{Best}

Get initial floorplan S; $S_{Best} = S$
Set initial temperature $T_0 = -\Delta_{avg} / \ln P$
while $T > T_{min}$ **do**
    Set accepted attempts = 0
    Set total attempts = 0
    for *ite* = 1 to k **do**
        Perturb the initial floorplan S to generate neighboring floorplan S' by choosing between changing the shape of some blocks in the layout or changing the location of some blocks in the layout.
        Compute $\Delta C = cost(S') - cost(S)$
        if $\Delta C \leq 0$ or $Random < e^{\Delta C/T}$ **then**
            Set $S = S'$
            if $cost(S) == 0$ **then**
                return S
            end if
            accepted attempts += 1
        end if
        total attempts += 1
    end for
    if $cost(S_{Best}) > cost(S)$ **then**
        Set $S_{Best} = S$
        if $cost(S_{Best} == 0)$ **then**
            return S_{Best}
        end if
    end if
    Compute Acceptance rate = accepted attempts / total attempts
    if Acceptance rate < Predefined value **then**
        Increase r (Cooling rate)
    else
        Decrease r
    end if
    $T = rT$; //reduce temperature
end while
return S_{Best}

4.5 Evolutionary Strategy Approach

ES is a stochastic optimization algorithm that is commonly used to solve complex optimization problems. Many researchers have utilized ES for solving the FPGA layout generation problem, as it produces fast and efficient solutions. ES is based on the natural process of evolution and uses a mutation operator to generate new candidate solutions.

In this work, a simple $(1 + \lambda)$ ES is used which is similar to the approach we used while solving the optimization problem using ES that is mentioned in section 3.5, where only one parent is selected in every generation, and λ represents the number of offspring that will be produced in each generation through the mutation operator. We have also used the technique of elitism similar to the ES optimization approach.

Below the main functions like the Selection process and Evolutionary operators are discussed.

4.5.1 Selection Process

The selection process is a crucial aspect of any evolutionary algorithm. The selection process determines which individual will be selected to become a parent and contribute to the next generation [22]. In this work, a tournament selection approach is employed to choose a single parent for the next generation. In the tournament selection approach, a small subset of individuals from the population is randomly selected, and the individuals with the highest fitness is chosen as the parent. The fitness function in this work defines the total number of overlaps between the reconfigurable slots. So, here the individuals with less number of overlaps is selected as the parent for the next generation. Tournament selection has been shown to be an effective and efficient selection method in many optimization problems [1]. The tournament selection technique is more robust than other selection methods since it eliminates the noise in the fitness landscape by evaluating fitness in a local context rather than globally and also reduces the impact of outliers [1]. One of the main advantages of tournament selection is that it provides a balance between exploration and exploitation. Since the code chooses individuals randomly for the tournament, the algorithm can explore the search space, while selecting the fittest individual ensures that the algorithm exploits the best solutions found so far.

4.5.2 Evolutionary Operators

Evolutionary operators play a very crucial role in Evolutionary algorithms. These operators are used for generating new candidate solutions for the next generations by mutating the selected parent from the selection process. In this work, only the mutation operator is used to generate the new off-springs. The crossover operator is not considered since it does not give many efficient solutions in this case. On the other hand, The mutation operator helps to explore the search space and generate diverse candidate solutions. It also helps the algorithm not to stuck at the local minima [26]. The mutation function includes two different operations. While generating the new candidate solutions one of these operations is used on some of the genes in a chromosome to produce more diverse chromosomes. Below are the two operators that are used in the mutation function. They are:

1. Change the position of the gene
2. Change the shape of the gene

Each generation involves selecting a parent through a selection process, which is then subjected to mutation to generate new candidate solutions. The mutation process typically involves altering a percentage of the genes in the chromosome.

In changing the position of the gene operation, it tries to randomly perturb the new x and y coordinates for the gene inside the given FPGA layout boundaries. The new x and y coordinates are assigned such that the gene should not be overlapping with any preplaced blocks in the FPGA and also the gene should not be outside the FPGA boundaries.

Let PPB be the set of preplaced blocks and S_i be the block whose coordinates are being changed from (x_i, y_i) to (x_j, y_j). We can define a function $overlap(S_i, S_k)$ which returns true if blocks S_i and S_k overlap and false otherwise.

Then, the condition for the new coordinates (x_j, y_j) to not overlap with preplaced blocks can be written as:

$$\forall S_k \in PPB, overlap(S_i, S_k) = False \tag{3}$$

This means that for all preplaced blocks S_k, the overlap function should return false when comparing the new position of S_i with S_k.

On the other hand, Changing the shape of the gene involves changing the width and height of the gene. As mentioned before all the genes are soft modules. It means all the modules can change their shape in the given aspect ratio bounds. The algorithm applies minimum and maximum aspect ratio bounds to each block to guarantee that the new width and height values are contained within a specified range of aspect ratios. The minimum and maximum aspect ratio bounds are obtained by finding the minimum and maximum dimensions that can be possible with the given block area i.e., $minimum(w_i, h_i)$ and $maximum(w_i, h_i)$. Here aspect ratio of each module is considered as height over the width of the module. To maintain the gene area the new width and height should follow a rule. That is

$$w_i * h_i = Area(gene_i) \tag{4}$$

The mutation function also has a mutation rate that controls the frequency at which mutations are applied to the given chromosome in the population. The mutation rate determines the probability of any gene in the chromosome being randomly changed in each generation. If the mutation rate is too low, the algorithm may not explore the solution space enough, leading to premature convergence and possibly getting stuck in local optima. On the other hand, a higher mutation rate can lead to more exploration of the solution space, which can help the algorithm to avoid getting stuck in local optima and find better solutions.

4.5.3 Algorithmic Details

In this work, there are several key parameters that were carefully selected to ensure the effective performance of layout generation of reconfigurable slots. The algorithm settings are same as the settings we used in the ES while solving the optimization problem. The algorithmic details can be found in subsection 3.5.3.

- For the population size in this work is chosen as similar to the work [26]. Here, in this work, the population size is set to 4 times the total number of reconfigurable slots that need to be placed on FPGA.

- The size of the tournament was set at twice the total number of reconfigurable slots or half the size of the population.
- The number of generations was set to 150, which provided enough time for the algorithm to converge to a reasonable solution
- Finally, the mutation rate is set between 0.03 to 0.08. This mutation rate makes sure that the population has enough diversity to explore the solution space effectively while still allowing for a sufficient amount of convergence toward a sound solution.

4.5.4 Final Evolutionary Strategy Algorithm for Decision Problem

Below provided the final Evolutionary Strategy algorithm used for solving the Decision problem.

Algorithm 6 Final Evolutionary Strategy Algorithm for Decision Problem

Require: Set the $population\_size$ to 4 * the total number of blocks in layout, Set the $tournament\_size$ as 2 * the total number of blocks in layout. Set the $num\_offsprings$ to 90% of the $population\_size$, and Set $num\_generations$ to 150. Set the $mutation\_rate$ to 0.5.

Ensure: Return the chromosome with the best fitness value

1: Initialize the population of $population\_size$ chromosomes with random values.
2: Evaluate the fitness of each chromosome in the population.
3: Check if any chromosome fitness is 0. If yes, return that chromosome as the best solution.
4: **for** $ite = 1$ to $num\_generations$ **do**
5: Sort Initial population and Fitness scores in ascending order of fitness values.
6: Check if any chromosome fitness is 0. If yes, return that chromosome as the best solution.
7: Select the top **10%** of sorted population as elites.
8: **for** $ite = 1$ to $population\_size - elites$ **do**
9: Sample a parent using tournament selection with $tournament\_size$
10: Mutate the selected parent with probability $mutation\_rate$ to create offsprings
11: **end for**
12: Add Elites to the Offsprings
13: Replace current population with new offsprings.
14: Evaluate the fitness of the new population
15: Check if any chromosome fitness is 0. If yes, return that chromosome as the best solution.
16: **end for**
17: Return the chromosome in the population with the best fitness value

5 Results and Experimentation

In this chapter, we will discuss the results of both the SA and ES algorithm for the optimization and decision problem. Prior to discussing the results, we will examine the test data used to evaluate the algorithms and the process used to generate this data.

5.1 Test Data

To demonstrate the functionality of the algorithms we have implemented a simulation framework in Python and executed a number of simulation experiments. For the experimentation, we randomly generated the slots and grouped them into two different categories for the optimization problem. They are More Blocks Less Area (MBLA) and Less Blocks More Area(LBMA) for Optimization problem.

As the name suggests in MBLA dataset there will be more number of reconfigurable slots with the less required area. Table 5.1 presents the factors we have considered while generating the test data. The dataset MBLA is generated by keeping the width and height of the FPGA in the range of 15 to 25. So, the total required area in each test file can be between 225 to 625 micro slots. We have also used the area utilization factor while generating the datasets. The area utilization factor represents the ratio of the total area occupied by reconfigurable slots to the total available area on the FPGA. It is a measure of how efficiently the FPGA's resources are being utilized by the generated test data. The area utilization for MBLA is set in the range of 0.40 to 0.90. This means that, on average, the generated test data will occupy 40% to 90% of the available area on the FPGA. While generating the MBLA datasets we made sure that we get more blocks with less area. For this, we have kept the area requirement of each block to be between 1 to 15 micro slots. With these settings, we have generated the MBLA dataset.

Parameters	Range
Width and Height of FPGA	[15,25]
Area utilization factor	[0.40,0.90]
Area for each block	[1,15]

Table 5.1: Parameter Ranges for MBLA Data Generation

We wanted each task file in the LBMA dataset to have fewer blocks but a higher area requirement. The factors that we took into consideration when producing the test data are shown in Table 5.2. The dataset LBMA is created by keeping the FPGA's width and height between 15 and 25. Therefore, each test file total area can range from 225 to 625 micro slots. While creating the datasets, we also used the area utilization factor. The range of the LBMA's area utilization is 0.10 to 0.40. We kept the area requirement of each block at between 15 and 30 micro slots to generate datasets with fewer blocks but higher area requirements.

Parameters	Range
Width and Height of FPGA	[15,25]
Area utilization factor	[0.10,0.40]
Area for each block	[15,30]

Table 5.2: Parameter Ranges for LBMA Data Generation

For creating the data for the decision problem, we have considered the actual number of micro slots present in the Xilinx Zynq 7020 FPGA device. Specifically, this device has a total of 14 slots, but only 11 of them are eligible to be micro slots. Therefore, when creating the datasets, we ensured that each dataset contained no more than 11 micro slots. We also considered area utilization when generating the data, setting it to a range of 0.50 to 1.0. With these parameters, we were able to create the data for the decision problem.

We have also generated a small dataset for experimentation where we have considered the actual available number of micro slots in the Xilinx Zynq family devices. Table 5.3 shows the device name and number of micro slots available in that device. We took the available micro slots value as the target value to generate the task sets and each reconfigurable slot in the task set has an area requirement between 1 to 6 micro slots. Area utilization is also considered while generating the task sets. Table 5.4 shows the parameters that are considered during the data generation.

Zynq device name	Total number of available micro slots
Zynq-7010	4
Zynq-7015	7
Zynq-7020	11
Zynq-7030	19
Zynq-7035	39
Zynq-7045	44
Zynq-7100	71

Table 5.3: Xilinx Zynq devices with respective available micro slots

Parameters	Values
Target Area	[4, 7, 11, 19, 39, 44, 71]
Area utilization factor	[0.5, 0.8, 1.0]
Area for each block	[1 - 6]

Table 5.4: Parameter Ranges for Experiment Data Generation

All the task generation code has been implemented in Python.

5.2 Results

In this section, we will see how both the SA and ES algorithms performed the optimization problem using the single test data from the MBLA dataset and LBMA dataset.

5.2.1 Result from MBLA Data

Here, let us consider one test data from the MBLA data. Table 5.5 refers to one of the test file from the MBLA dataset. Here each block represents a reconfigurable slot with the required area on the FPGA. Table 5.5 shows that all the reconfigurable slots have an area of at most 15 micro slots.

Block Name	Area
Block1	10
Block2	12
Block3	4
Block4	10
Block5	6
Block6	3
Block7	14
Block8	13
Block9	10
Block10	10
Block11	4
Block12	12
Block13	5
Block14	10
Block15	1
Block16	3
Block17	5
Block18	8
Block19	3
Block20	15
Block21	1
Block22	8
Block23	1
Block24	4
Block25	2
Block26	14
Block27	12
Block28	7

Table 5.5: Sample test data from MBLA dataset for optimization problem

Further, we will see the visualization of final layouts generated by SA and ES algorithms for the considered data.

Figure 5.1 is the resulting layout given by the SA algorithm for the considered test data. Here, in Figure 5.1, all the reconfigurable slots are placed on the given FPGA. As per the optimization problem, all the reconfigurable slots need to be placed without overlapping with any other reconfigurable slot and also should not be placed outside the boundaries of the given FPGA. These two rules were followed by the algorithm.

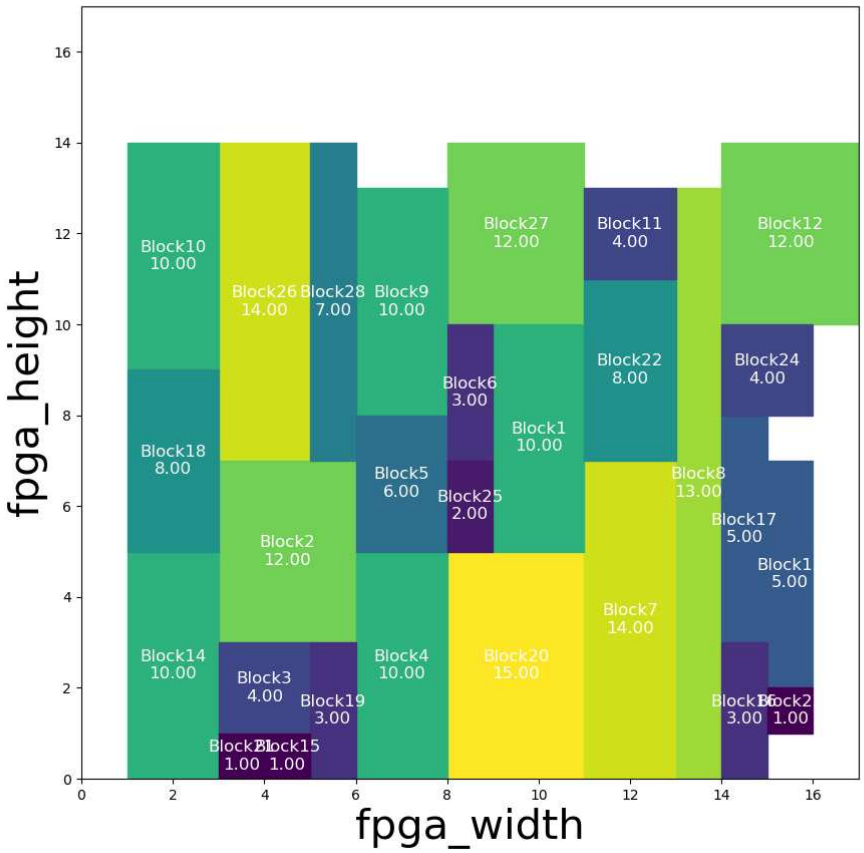


Figure 5.1: Resulting floorplan from SA algorithm for the optimization problem using MBLA data.

In the optimization problem, we want to find out the MBR area for this final layout. As mentioned earlier, MBR refers to the smallest rectangular area that encloses all the reconfigurable slots placed on an FPGA. To find the MBR area, we have taken the minimum over all of x_i as $x_0 = 1$ and the minimum over all of y_i as $y_0 = 0$ which means the bottom left corner of MBR is (1,0) and maximum over all of x_i as $x_1 = 17$ and maximum over all of y_i as $y_1 = 14$ so the top right corner of the MBR is (17,14). From this, we can find the MBR area using $(x_1 - x_0) * (y_1 - y_0)$. So, the final bounding rectangle area for this data given by SA is $(17 - 1) * (14 - 0) = 224$ micro slots.

Figure 5.2 displays the resulting layout generated by the ES algorithm for the given data. The algorithm ensures that all the reconfigurable slots are placed on the given FPGA, without any overlapping or placement outside the boundaries of the FPGA, as per the optimization problem requirements.

The optimization problem aims to determine the MBR area for the given data, Figure 5.2 has the bottom left corner at $x_0 = 0$ and $y_0 = 0$ and the top right corner is at $x_1 = 16$ and $y_1 = 16$. MBR area can be found using the formula $(x_1 - x_0) * (y_1 - y_0)$, which

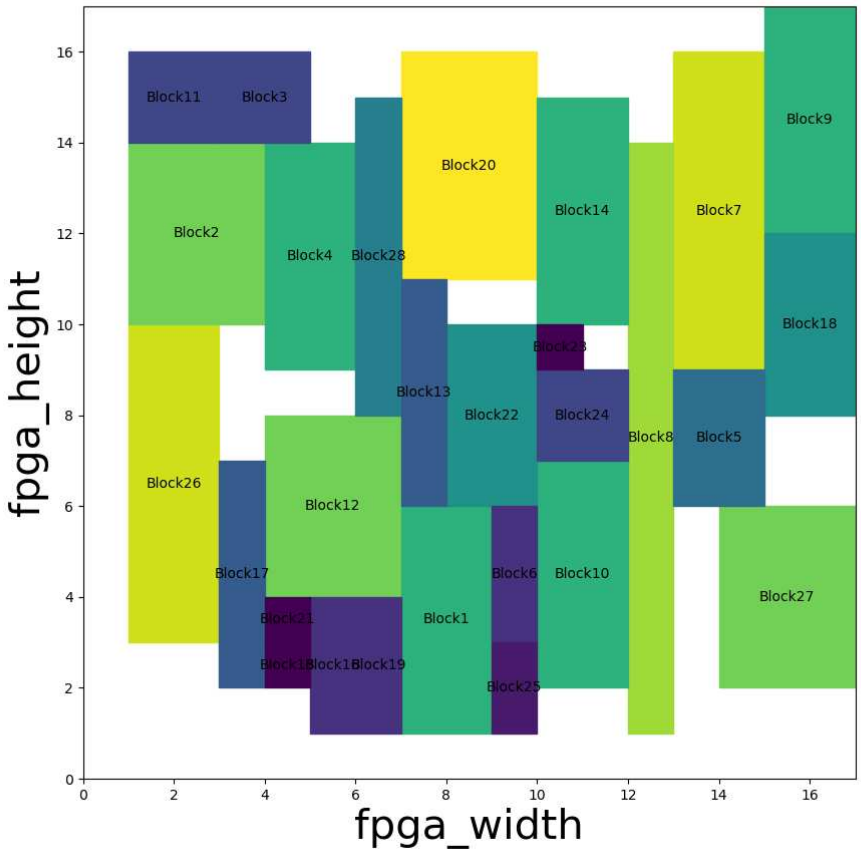


Figure 5.2: Resulting floorplan from ES algorithm for the optimization problem using MBLA.

yields a MBR area of $(16 - 0) * (16 - 0) = 256$ micro slots, as determined by the ES algorithm.

By considering the above MBR from both algorithms. We can say that SA has placed the given reconfigurable slots in less number of micro slots than ES.

5.2.2 Result from LBMA Data

Now, let us consider the single test data from the LBMA dataset. Table 5.6 is one of the test data from the LBMA dataset. Here, each block represents a reconfigurable slot with the required area. As mentioned in the above section each reconfigurable slot will have an area between 15 to 30 but there will be fewer blocks in each test file.

Below we will see the visualization of final layouts generated by SA and ES algorithms for the considered data.

Figure 5.3 is the output from the SA algorithm for the above-considered data. To find

Block Name	Area
Block1	28
Block2	17
Block3	15
Block4	21
Block5	17

Table 5.6: Sample test data from LBMA Dataset for Optimization problem

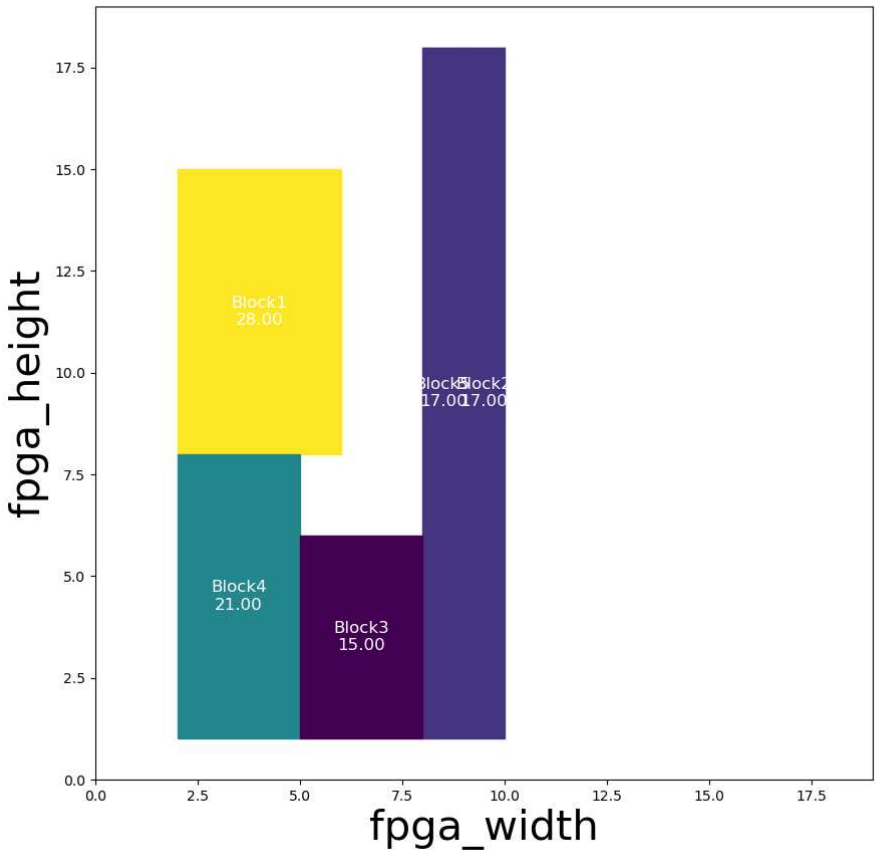


Figure 5.3: Resulting floorplan from SA algorithm for the Optimization problem using LBMA data.

the MBR area for this data, we need the bottom left corner and the top right corner of the MBR. We can obtain the bottom left corner by taking the minimum of all x_i values and the minimum of all y_i values as x_0 and y_0 respectively. Similarly, we can obtain the top right corner by taking the maximum of all x_i values as x_1 and the maximum of all y_i values as y_1. In this case, the bottom left corner is (2,1) and the top right corner is (10,18), as shown in Figure 5.3. Using this data the MBR area can be found using $(x_1 - x_0) * (y_1 - y_0)$. In this case, $(10 - 2) * (18 - 1) = 136$. SA has placed the given

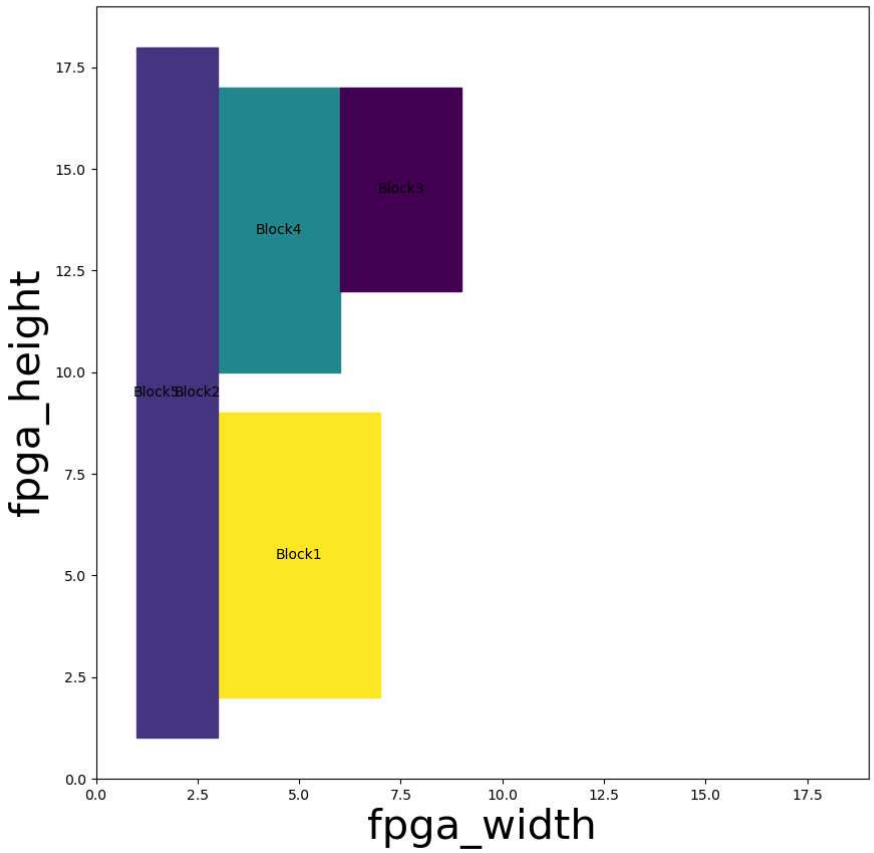


Figure 5.4: Resulting floorplan from ES algorithm for the Optimization problem using LBMA data.

reconfigurable slots in 136 micro slots.

Figure 5.4 is the output from the ES algorithm for the optimization problem. To obtain the MBR area we need the bottom left corner and the top right corner of the bounding rectangle. In Figure 5.4 the bottom left corner is $(x_0, y_0) = (1,1)$ and the top right corner is $(x_1, y_1) = (9,18)$. From these points, we can find the minimum bounding rectangle area using $(x_1 - x_0) * (y_1 - y_0)$. So, the final bounding rectangle area for this data given by ES is $(9 - 1) * (18 - 1) = 136$. Here, we can say ES requires 136 micro slots to place the reconfigurable slots on an FPGA.

For the considered data, both the SA and ES algorithms have performed equally well in optimizing the problem and finding the minimum bounding rectangle area for the given data. However, it is important to note that this conclusion may not generalize to other test files.

5.3 Experimentation and Comparison

In this section, we delve into the realm of experimentation and comparison among four different algorithms: Simulated Annealing, Evolutionary Strategy, Heuristic algorithm, and Optimal algorithm. To ensure efficient execution, all experiments were conducted on High-Performance Computing (HPC) clusters, leveraging the parallel processing capabilities of nodes and cores. A node refers to an individual computational unit within the HPC cluster. It typically consists of several components, including CPUs, RAM, clock rates, and other relevant specifications. We have considered 5 nodes. Each node was equipped with multiple CPUs. Additionally, the nodes were equipped with a memory capacity of 2GB per CPU.

5.3.1 Optimization Problem

This section involves the comparison and experimentation regarding the optimization problem. The newly implemented algorithms SA and ES are compared with heuristic and optimal algorithms using runtime and quality of the output as the metric. Both the runtime and quality experiments are performed on MBLA and LBMA datasets. Further, we will see what is the quality experiment and runtime experiment.

1. **Runtime Experiment:** To evaluate the runtime performance of the SA and ES algorithms, we ran the algorithms until the MBR area of each file from the test data reaches the MBR area produced by the heuristic algorithm. Specifically, we measured the time it took for each algorithm to converge to the heuristic output and compared the quality of the solutions generated by each algorithm. By comparing the performance of the SA and ES algorithms against the heuristic and optimal algorithms, we were able to assess their effectiveness in solving the problem at hand.

2. **Quality Experiment:** To evaluate the quality of the SA and ES algorithms, we have conducted a quality experiment where we run each test data for a constant amount of time to see how the quality of the output changes over time. We also included heuristic and optimal algorithms as a baseline for comparison.

We have made significant changes to the optimal algorithms in order to compare their results with the SA, ES, and Heuristic algorithms in our study. Initially, the Optimal solution that is defined in subsection 2.2.3 aims to optimize the height of the floorplan However, for the purpose of our research, we decided to focus on the MBR area as the metric for comparing the results across different algorithms.

During experimentation with the Optimal algorithms using the test data, we discovered that the floorplan with the minimum height generated by the Optimal algorithm might not necessarily have the optimal MBR area. So, we introduced a modification by setting the FPGA width to the width of the MBR obtained from the SA algorithms in order to guarantee that we obtain the best MBR area from the Optimal algorithms for each test case. After these adjustments, we named it as Optimal* algorithm. This adjustment allows us to observe how effectively the reconfigurable slots can be placed by aligning the FPGA width with the MBR width.

Runtime Experiment with MBLA Data

Figure 5.5 illustrates a comparison of the MBR areas between SA, ES, Heuristic, and Optimal algorithms for all the MBLA data. The X-axis shows the file names while the Y-axis displays the final bounding areas. As evident from Figure 5.5, Both the SA and ES have reached the results of heuristic algorithms for most files. In many instances, the SA algorithm has outperformed both the ES and heuristic algorithms. On the other hand, optimal algorithms can generate a better layout than all the 3 other algorithms.

The comparison of the performance of the SA and ES algorithms reveals some interesting results. SA outperformed the ES algorithm in approximately 48% of the cases, demonstrating its ability to find superior solutions. Furthermore, SA and ES produced identical results in 25% of the cases, implying comparable performance in those scenarios. When comparing SA with the Heuristic algorithm, SA showcased superior performance in nearly 56% of the instances. This demonstrates SA's ability to outperform Heuristic algorithms in a majority of the cases.

ES, on the other hand, demonstrated its strength by outperforming SA in approximately 27% of the cases. Furthermore, in approximately 55% of the files, ES outperformed the Heuristic algorithm, indicating its effectiveness in those cases.

It is worth noting that the Heuristic algorithm outperformed SA in 25% of the cases, demonstrating its ability to excel in specific scenarios. Furthermore, when compared to ES, the Heuristic algorithm outperformed it in 40% of the files, indicating its favorable performance in a significant portion of the experiments. This is because the heuristic algorithm is specifically designed to solve the problem at hand.

In addition to the comparisons between SA, ES, and the Heuristic algorithm, it is worth noting that the Optimal* algorithm consistently outperformed the others in the majority of cases. The Optimal* algorithm demonstrated its effectiveness in solving complex problems as an algorithm designed to achieve the globally optimal solution through mathematical optimization techniques.

It is important to note that Optimal* algorithms typically require significant computational resources due to their computationally intensive nature. Nonetheless, the results of both SA and ES were found to be close to the optimal* solutions, showing their strong performance in approximating the best possible outcomes.

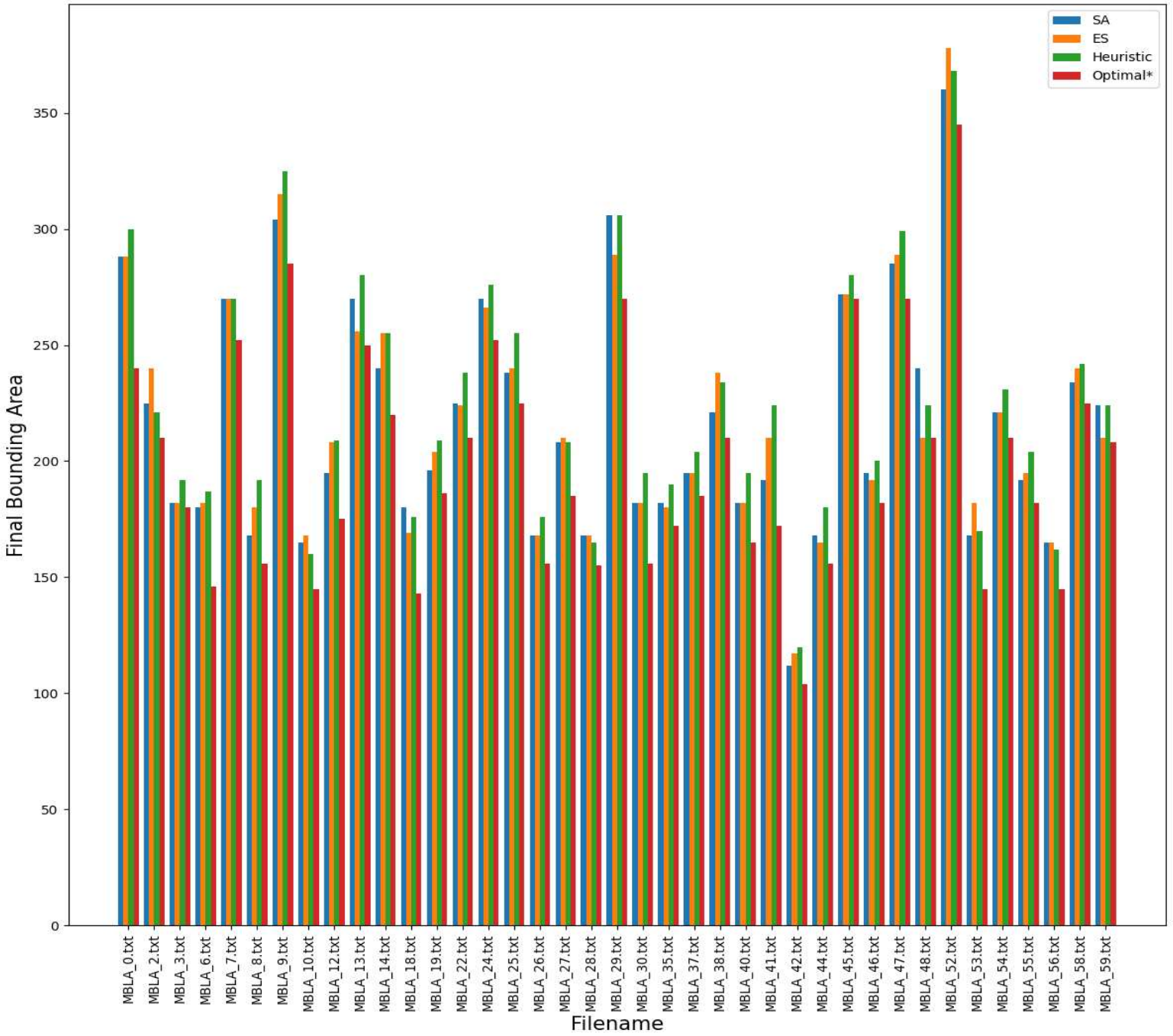


Figure 5.5: Comparison of MBR area between SA, ES, Heuristic, and Optimal algorithms for quality experiment with MBLA data

In Figure 5.6, we present a runtime comparison between the SA, ES, and Heuristic algorithms specifically for the quality experiment. Because of their inherent computational intensity, the runtimes of the Optimal* algorithms have been excluded from Figure 5.6. It is worth mentioning that the Optimal* algorithms often necessitated several hours to reach the optimal results. When compared to the Optimal* algorithms, the SA, ES, and Heuristic algorithms produced significantly faster results. Their runtimes were much shorter, allowing for more efficient exploration of the solution space.

The graph in 5.6 shows that both the SA and ES algorithms took more time to produce results than the heuristic algorithm. This outcome was expected, given that the SA and ES algorithms are known to take more time to solve problems. The reason for the long runtime of SA and ES algorithms is that they perturb the initial layout and population multiple times to find the desired solution. In contrast, the heuristic algorithm is tailored to solve the specific problem and does not require a specific number of generations or inner iterations to generate the desired output. The runtime of the heuristic algorithm is significantly lower than that of SA and ES algorithms almost 0, as it is designed to be more efficient.

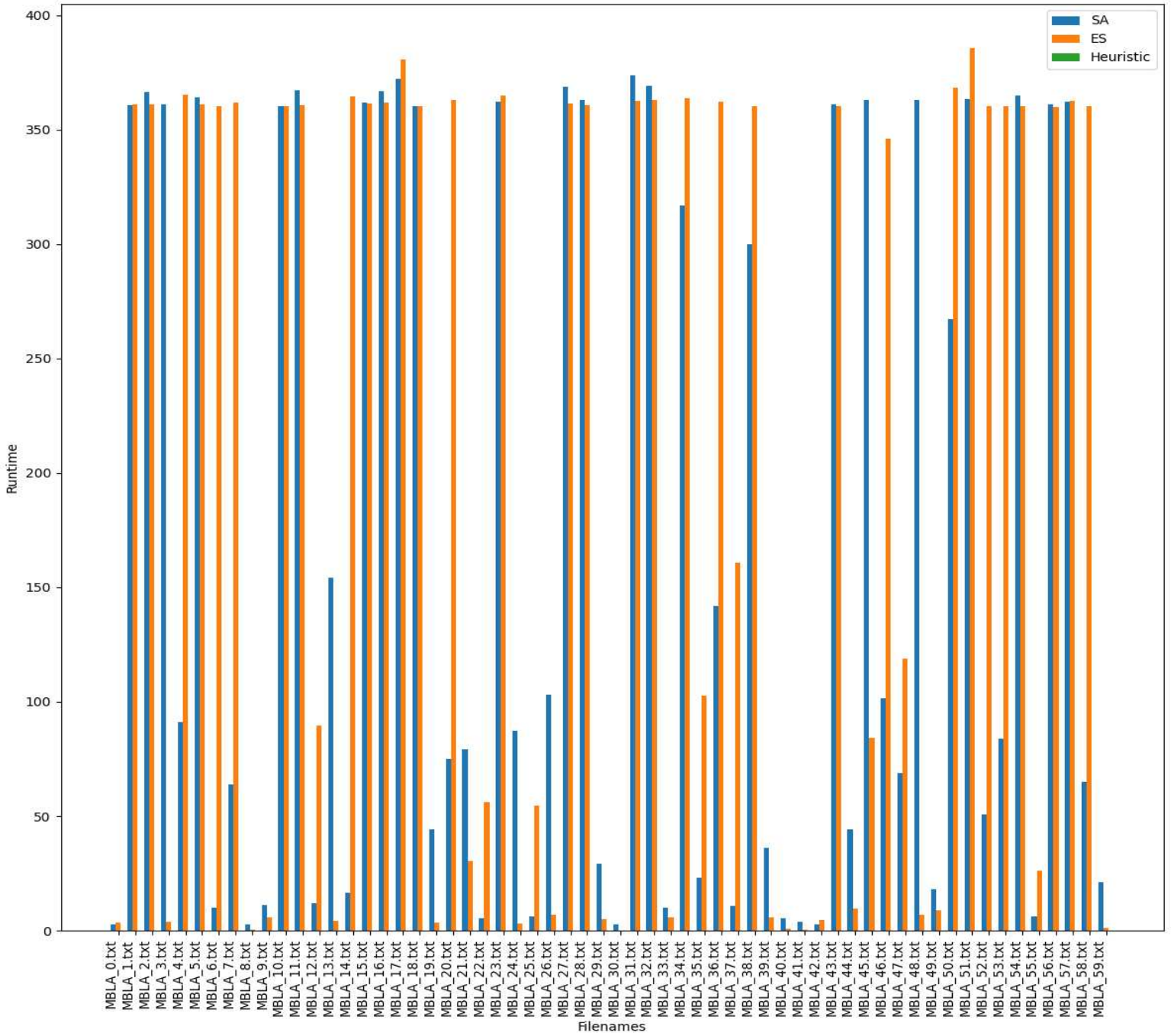


Figure 5.6: Comparison of runtime between SA, ES, and Heuristic algorithms for quality experiment with MBLA data

However, it is worth noting that the SA and ES algorithms have the potential to produce more promising layouts than the heuristic algorithm. The algorithms can generate a diverse set of solutions that can lead to better outcomes, although this may come at the cost of longer runtimes. Overall, this experiment highlights the trade-off between runtime and solution quality. While the heuristic algorithm offers a quick solution, it may not always produce the best results. In contrast, the SA and ES algorithms may take longer to solve the problem but can potentially generate more promising layouts. The choice of an algorithm should be based on the specific problem and the importance of runtime and solution quality.

Quality Experiment with MBLA Data

To evaluate the performance of the SA and ES algorithms, we have conducted a runtime experiment where we ran each test data for 15 mins to see how the quality of the output changes over time. We also included Heuristic and Optimal* algorithm results as a baseline for comparison.

After analyzing the result, in Figure 5.7 we found that SA outperformed ES, and Heuristic algorithms in most cases, with a higher success rate in finding a better solution within the given time frame. SA emerged as the dominant algorithm, outperforming ES in roughly 70% of the cases. ES, on the other hand, outperformed SA in 15% of the cases, demonstrating its ability to outperform in specific scenarios. When compared to the Heuristic algorithm, SA proved to be more effective, outperforming it in nearly 70% of the files. However, it is worth noting that the Heuristic algorithm outperformed the SA algorithm in 20% of the cases, highlighting the contextual dependency of algorithm performance. Furthermore, when compared to the Optimal* algorithm, SA achieved identical results in 20% of the cases.

ES, on the other hand, demonstrated its superiority by outperforming the Heuristic algorithm in roughly 54% of the cases. When compared to the Optimal* algorithm, ES produced layouts that achieved the same results in 10% of the cases.

One explanation might be SA's capacity to explore a larger solution space, which might enable it to discover better solutions than Heuristics and ES. SA is able to find better solutions because it can avoid becoming trapped in local optima. Another factor might be SA's adaptive nature, which allows for a better balance between exploration and exploitation by adjusting the cooling parameter in accordance with the acceptance rate. This allows SA to better adapt to different problem instances and find better solutions.

Additionally, the SA algorithm supports a probabilistic solution acceptance mechanism, allowing it to accept solutions that are worse than the current solution with a certain probability. This feature allows the algorithm to avoid getting trapped in a suboptimal solution and potentially find a better solution later in the search process. On the other hand, the ES algorithm has demonstrated good performance only on a limited number of test cases, even after running for 15 minutes but with enough runtime ES can escape local minima. One possible reason could be the lack of sufficient diversity in the initial population. If the initial population of solutions is not diverse, the algorithm may converge prematurely to a local maximum or settle for a suboptimal solution. To overcome this we may need to run the algorithm multiple times to check the quality of the solution.

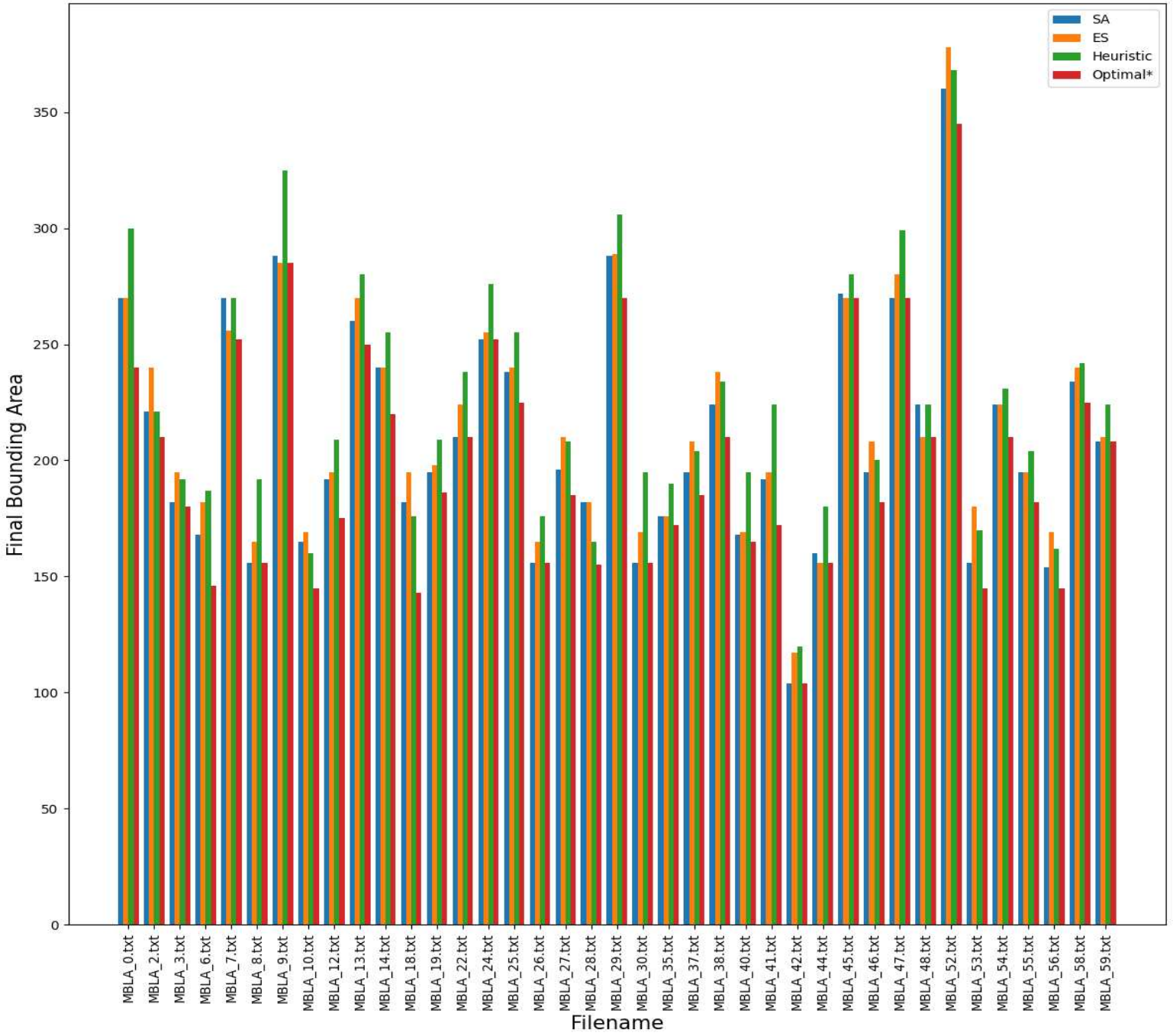


Figure 5.7: Comparison of the bounding area between SA, ES, Heuristic, and Optimal algorithms for runtime experiment with MBLA data

Based on the results of the experiment, it can be inferred that heuristic algorithms have the advantage of quickly providing a feasible solution. On the other hand, SA and ES algorithms require more time to converge to an optimal solution as they involve iterative improvements. However, SA has proven to be a better performer than ES in most cases and can provide better solutions if given enough time to converge.
In summary, the choice of algorithm depends on the problem at hand and the trade-off between solution quality and execution time. If a quick feasible solution is sufficient, a heuristic algorithm can be used. However, if a higher-quality solution is required and time is not a constraint, SA can be a better option.

Runtime Experiment with LBMA Data

Figure 5.8 depicts the final bounding area comparison between SA, ES, Heuristic, and Optimal* algorithms for LBMA data. The X-axis shows the filenames and the Y-axis shows the final bounding area. As evident from Figure 5.8 that SA and ES have reached the MBR area is the same as the minimum bounding rectangle area returned by the Heuristic algorithms. There are a few instances where SA and ES could not reach the heuristic results. In that case, we have depicted the minimum MBR area given by the SA and ES algorithms. As Figure 5.8 depicts that in most of the instances, SA has reached better solutions than ES and Heuristic results.

SA outperformed ES in nearly 60% of the cases, demonstrating its ability to achieve better results in the majority of scenarios. Furthermore, when compared to Heuristic algorithms, SA outperformed them in approximately 75% of the cases, highlighting its effectiveness in outperforming Heuristic-based solutions.

When evaluated against the Optimal* algorithms, SA achieved the same results as the Optimal* algorithms in 32% of the files, demonstrating its ability to generate solutions on the level with the globally optimal ones.

In contrast to SA, the ES algorithm demonstrated distinct performance characteristics when compared to the SA, Heuristic, and Optimal algorithms.

In approximately 11% of the cases, ES outperformed SA, demonstrating its ability to produce superior results in specific scenarios. Furthermore, in nearly 28% of the cases, ES achieved the same results as SA, indicating that the two algorithms performed similarly in those cases.

When compared to Heuristic results, ES outperformed Heuristic by providing better layouts in nearly 53% of the cases. Furthermore, in 18% of the cases, ES produced identical results to the Heuristic algorithm.

Evaluating ES against Optimal* algorithms revealed for almost 19% of the files, ES achieved the same results as the Optimal* algorithms.

However, the Heuristic and Optimal* algorithms perform better than both the SA and ES algorithms for some of the files. Therefore, the effectiveness of the algorithms can vary depending on the problem and the data set being used.

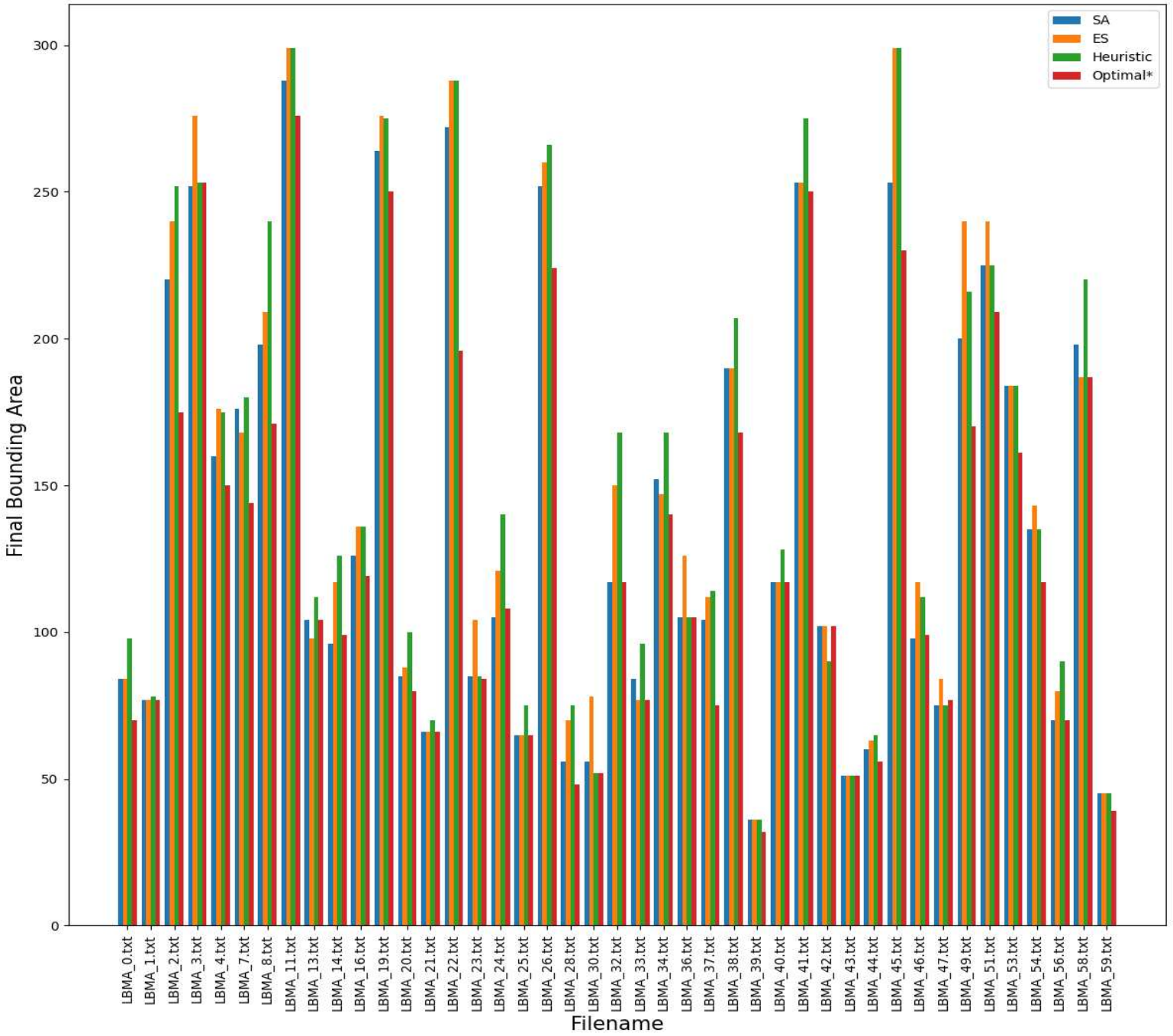


Figure 5.8: Final bounding area comparison between SA, ES, Heuristic, Optimal results for quality experiment with LBMA data

In Figure 5.9, we compare the runtimes of the SA, ES, and Heuristic algorithms for quality experiments on LBMA data. Because of their longer computation times, the runtime data for the Optimal* algorithms have been excluded for clarity, with some files taking several hours to reach the optimal solution.

The plot clearly demonstrates that the SA algorithm achieved results comparable to the Heuristic algorithm significantly faster than the ES algorithm when applied to LBMA data SA consistently achieved heuristic results in the vast majority of cases, with only a few exceptions. Surprisingly, the runtime for obtaining heuristic results using SA was nearly zero.

ES, on the other hand, achieved heuristic results in the majority of cases but required longer runtimes than both SA and the Heuristic algorithm. Almost 30% of the instances using ES failed to achieve heuristic results, indicating some limitations in its effectiveness. However, there were times when ES outperformed SA in terms of achieving heuristic results faster.

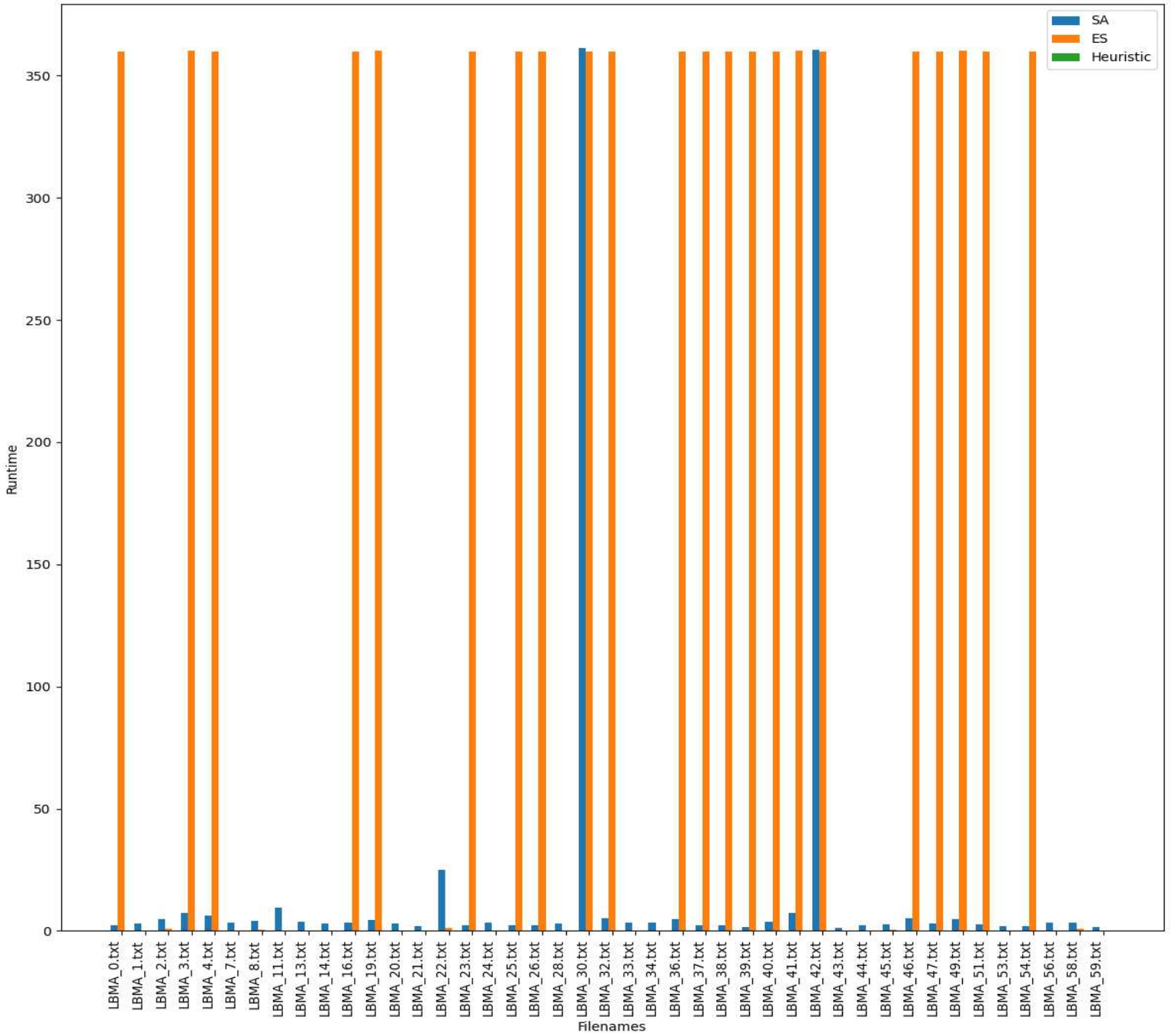


Figure 5.9: Runtime comparison between SA, ES, Heuristic, and Optimal results for quality experiment with LBMA data

Quality Experiment with LBMA Data

Figure 5.10 compares the MBR area obtained by the SA, ES, Heuristic, and Optimal* algorithms. The SA and ES algorithms were run for a duration of 10 minutes to observe changes in output within this time frame.

The results shown in Figure 5.10 demonstrate that SA and ES outperform the Heuristic algorithm in the majority of cases. SA outperformed the Heuristic algorithm in approximately 77% of the files, while ES outperformed the Heuristic algorithm in 31% of the cases. It's worth noting that there were times when both SA and ES produced the same results as the Heuristic algorithm. In most cases, SA outperformed both ES and the Heuristic algorithm, with SA outperforming ES in nearly 78% of the files and ES outperforming SA in only 7% of the cases.

When comparing the performance of SA and ES to the Optimal* algorithms, it is clear that in 53% of the cases, SA produced the same results as the Optimal algorithms. ES, on the other hand, achieved the same results as Optimal* in 20% of the cases.

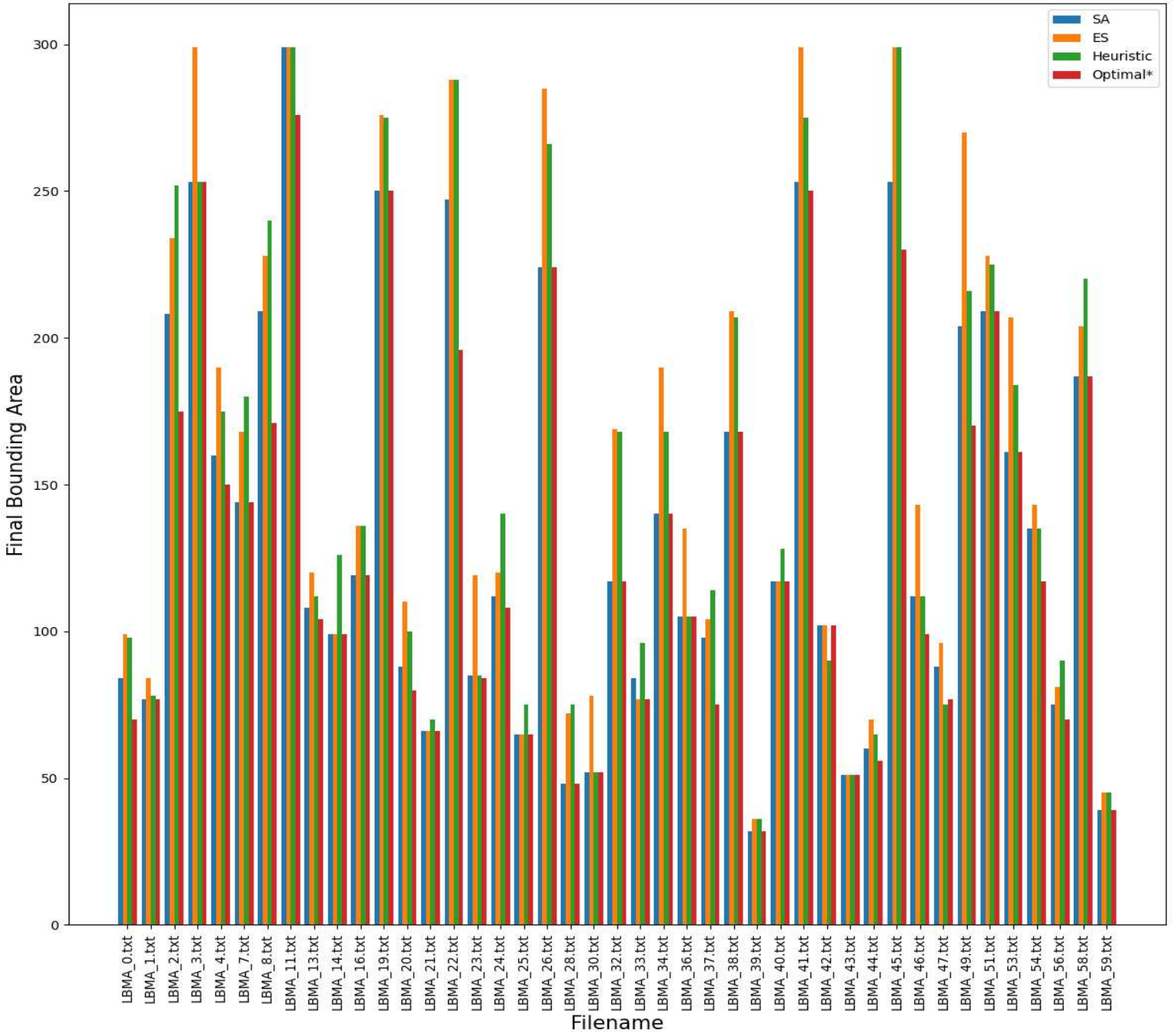


Figure 5.10: Comparison of the minimum bounding area between SA, ES, Heuristic, and Optimal algorithms for runtime experiment with LBMA data

These findings underscore the effectiveness of SA and ES in surpassing the performance of the Heuristic algorithm in terms of minimum bounding rectangle area over time. Furthermore, SA demonstrates a notable advantage over ES in terms of outperforming both the Heuristic and Optimal algorithms in a majority of cases.

Enhancing Shape Flexibility: The 'Area Adjustment' Technique for Improved Block Placement on FPGA

Previously, in the subsection 3.4.1 we referred to the technique as a "trick," but we have now formalized it as the "Area Adjustment" technique. In this section, we present the results of experiments with the SA and ES algorithms that used the Area Adjustment technique to increase the possible shapes of certain blocks in the test data.

The Area Adjustment technique's main goal is to produce better layouts by increasing the block area of particular blocks. With the help of this method, we aim to achieve a more even distribution of blocks throughout the FPGA, which will enhance routing effectiveness, lessen congestion, and increase overall performance.

We ran the SA and ES algorithms twice, once without the area adjustment technique and once with it, in order to assess the technique's effectiveness. We can evaluate the impact of the area adjustment technique on the quality of the generated layouts and its potential advantages for FPGA design by comparing the results of these two runs.

We have performed runtime experiments with SA and ES algorithms including the technique to reach the heuristics. Figure 5.11 presents a bar plot comparing SA without the technique and SA with the technique for MBLA data. The plot displays the bounding area values produced by both algorithms.

Upon analysis, we observed that in most cases, the SA algorithm with the technique resulted in a larger bounding area compared to the SA algorithm without the technique. This outcome is expected because the technique increased the block area of certain blocks by 1. The primary objective of employing this technique was to achieve a more uniform layout. Approximately 15% of the files exhibited a better bounding area when the technique was used and 25% of the files reached the same bounding area as SA without the trick.

Figure 5.12 illustrates a bar plot comparing the performance of the SA algorithm without the technique and the SA algorithm with the technique, specifically for LBMA data. The plot showcases the resulting bounding area values generated by both algorithms.

Upon careful examination, it is evident that a significant number of files in the LBMA dataset required less bounding area when the technique was employed. The effectiveness of the technique in reducing the bounding area can be clearly observed in the plot. While there are 32% of total instances where SA without the technique outperformed SA with the technique, the majority of cases almost 44% of instances demonstrate that SA with the trick generated superior layouts.

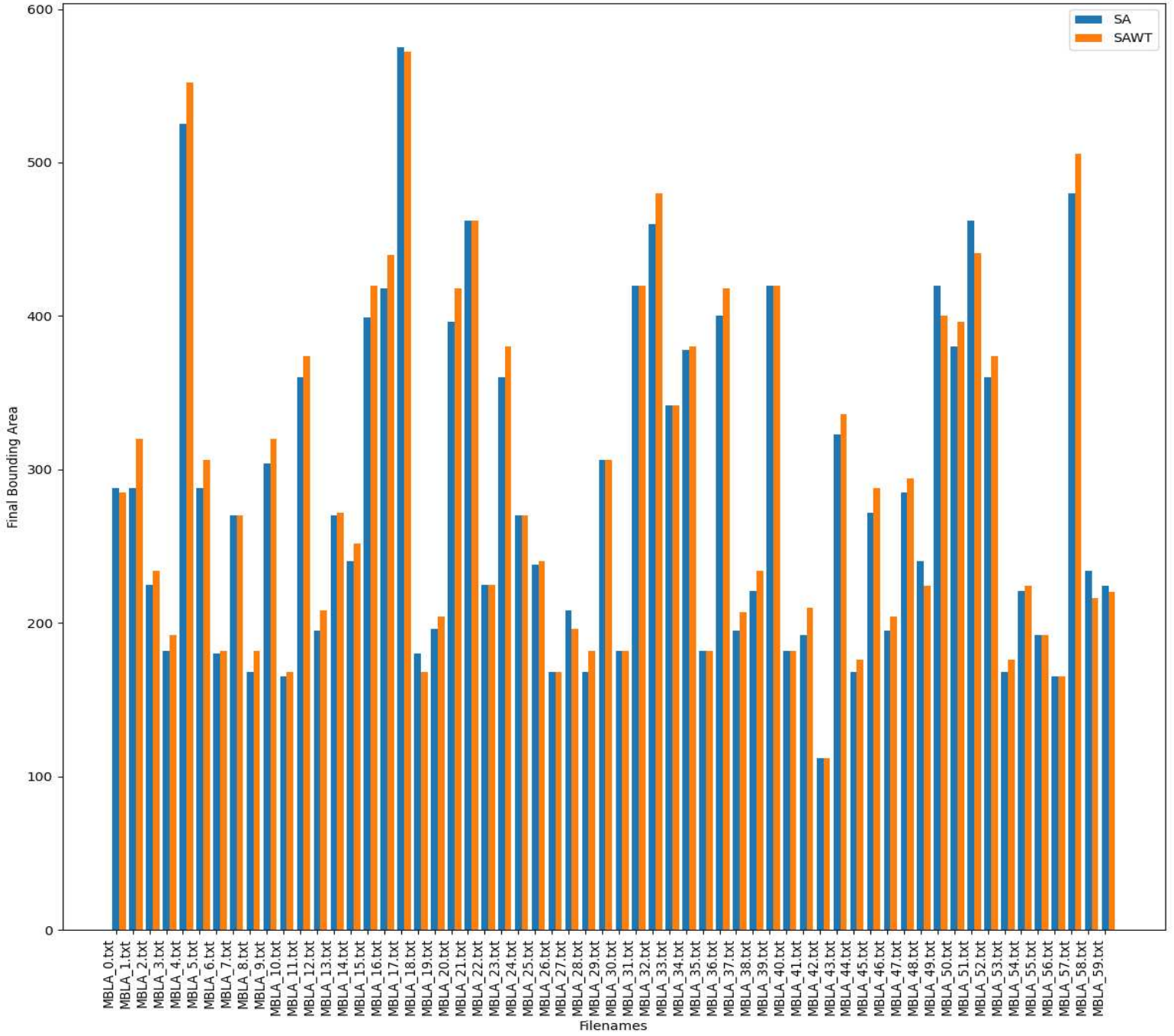


Figure 5.11: Comparison of the bounding area for MBLA data from SA with the Area Adjustment technique and SA without the Area Adjustment technique

Figure 5.12: Comparison of the bounding area for LBMA data from SA with the Area Adjustment technique and SA without the Area Adjustment technique

Figure 5.13 shows the bar plot comparing the performance of the ES algorithm without the technique and the ES algorithm with the technique, specifically for the MBLA dataset. The bar plot depicts the bounding area achieved by both algorithms.

Upon analysis, it is evident that the integration of the area bounding technique in the ES algorithm led to an increase in the required bounding area for placing the blocks on the FPGA. This increase can be attributed to the technique's effect of expanding the block area by 1 for specific blocks within the dataset. However, it is crucial to note that despite the larger bounding area, the layout achieved with the trick can offer distinct advantages such as Enhanced Routing Efficiency, Improved Timing and Performance, and Design Flexibility and Reusability.

Remarkably, the area adjustment technique resulted in a better bounding area for approximately 41% of the dataset. This considerable portion shows how significantly the technique affects layout optimization. Although the technique does result in increased space requirements overall, it is essential to consider the trade-off between increased area utilization and the benefits of a more uniform layout. While the trick may increase the overall area requirements, the resulting layout is often more optimal in terms of routing and performance in FPGA design.

Figure 5.14 depicts the bar plot comparing the performance of the ES algorithm without the technique (ES) and the ES algorithm with the technique (ESWT), Specially for the LBMA dataset.

Upon careful observation, we can say that there are 30% of instances from LBMA data have got a better layout by using the area adjustment technique, and 27% of the instances have reached the same result as SA without the area adjustment technique.

By the results of both SA and ES algorithms, we can say that both SA and ES with the technique have shown superior performance than SA and ES without the technique. For the MBLA data ES algorithm including the area adjustment technique has shown better results than the SA algorithm including the area adjustment technique. For the LBMA data SA algorithm including the area adjustment technique has shown better results than the ES algorithm including the area adjustment technique.

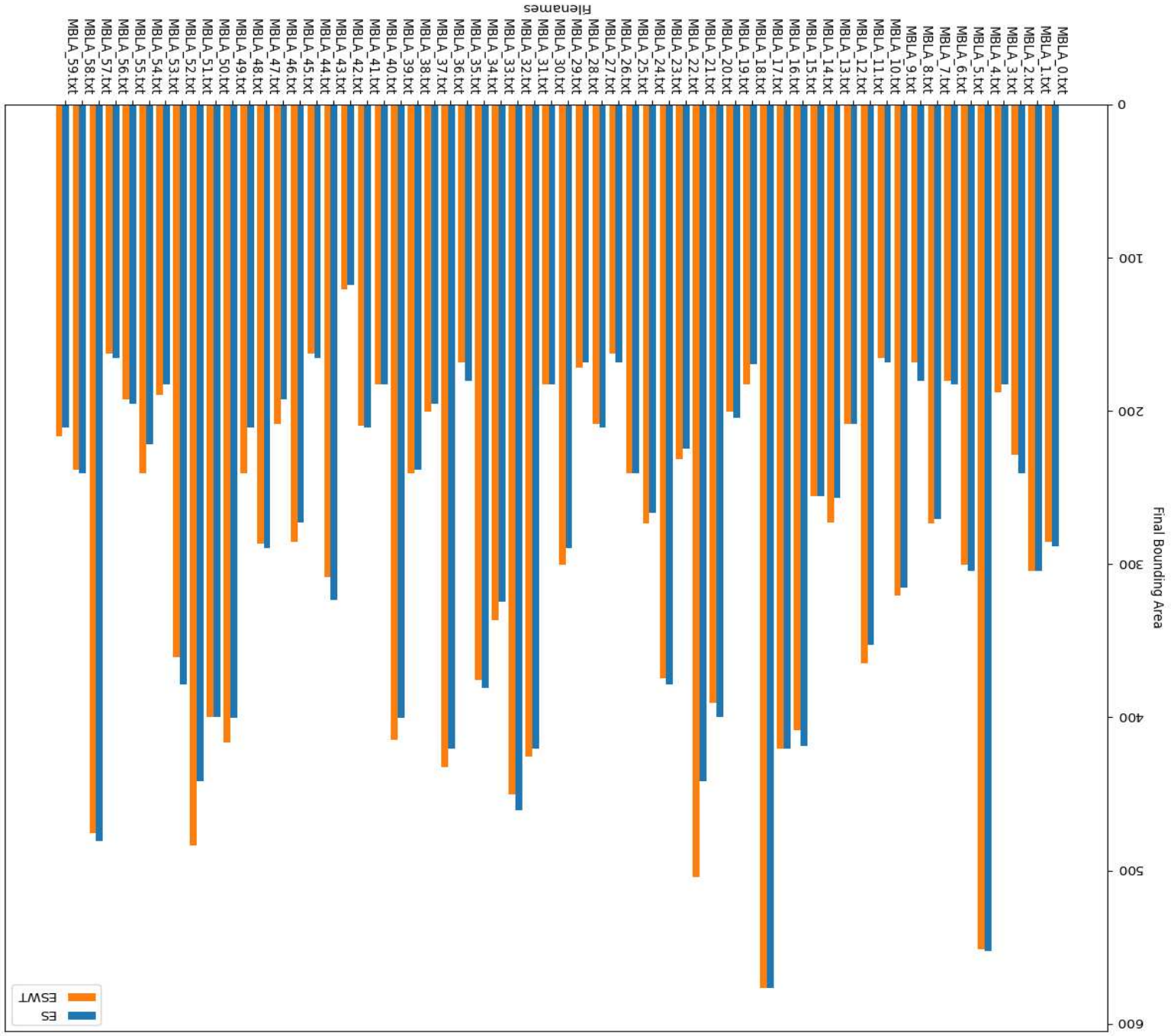


Figure 5.13: Comparison of the bounding area for MBLA data from ES with the Area Adjustment technique and ES without the Area Adjustment technique

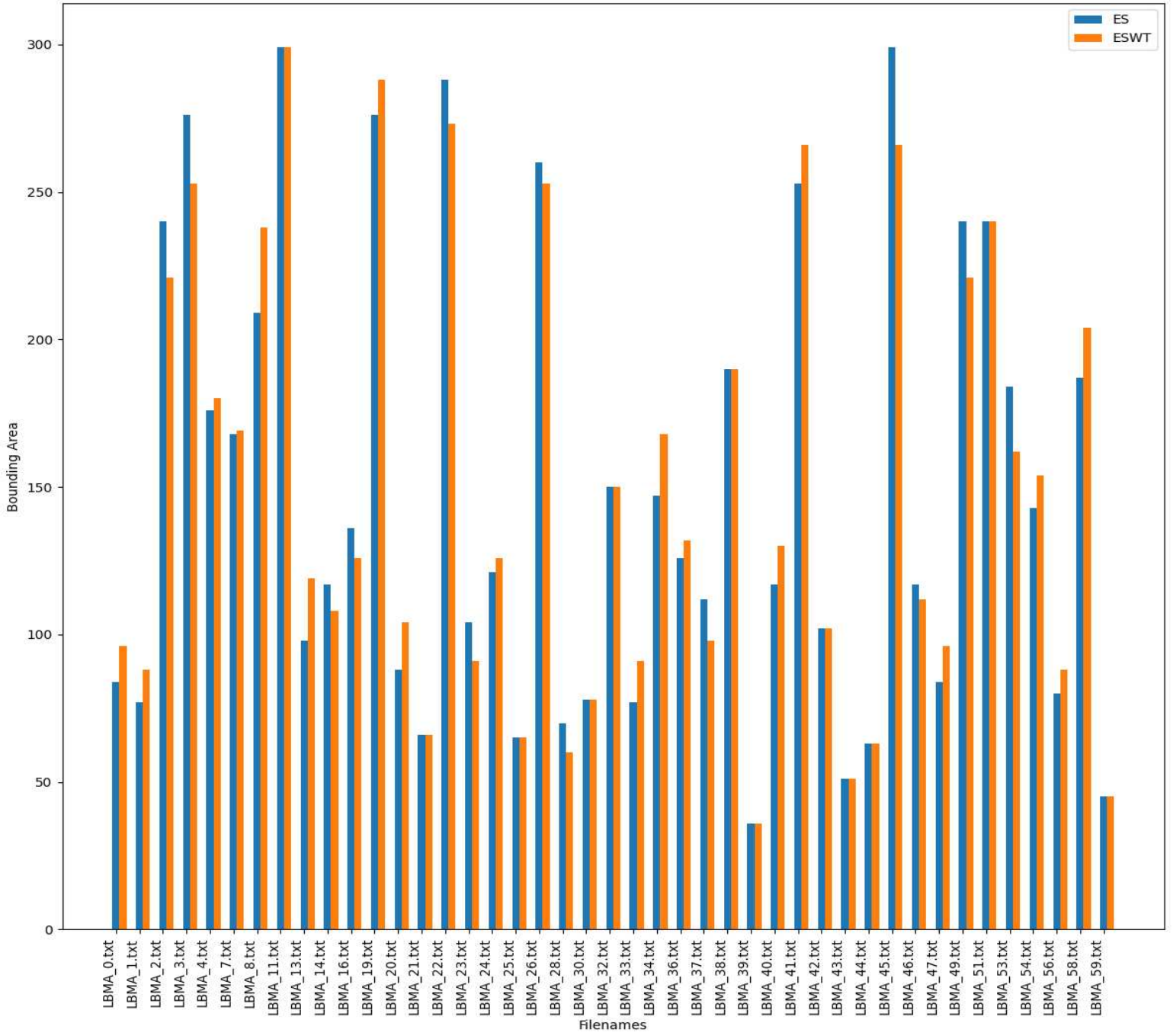


Figure 5.14: Comparison of the bounding area for LBMA data from ES with the Area Adjustment technique and ES without the Area Adjustment technique

Comparative Analysis of Requested vs Required Areas for Block Placement on FPGA

In order to compare the efficiency of the SA and ES algorithms, an experiment was conducted to plot the requested number of micro slots in the dataset versus the required number of micro slots to place the requested number of micro slots on FPGA surface, using SA and ES algorithms. The data used for this experiment is created considering the available number of micro slots in different Xilinx Zynq devices.

Figure 5.15 depicts that in most cases, SA has placed the requested number of micro slots in a less bounding area when compared to ES. It also indicates that the test data which has the requested number of micro slots up to 10 micro slots stay below the capacity of Xilinx Zynq 7020 which has 11 micro slots. However, this does not mean that all the test data with the requested number of micro slots below 10 have a feasible arrangement on the selected FPGA.

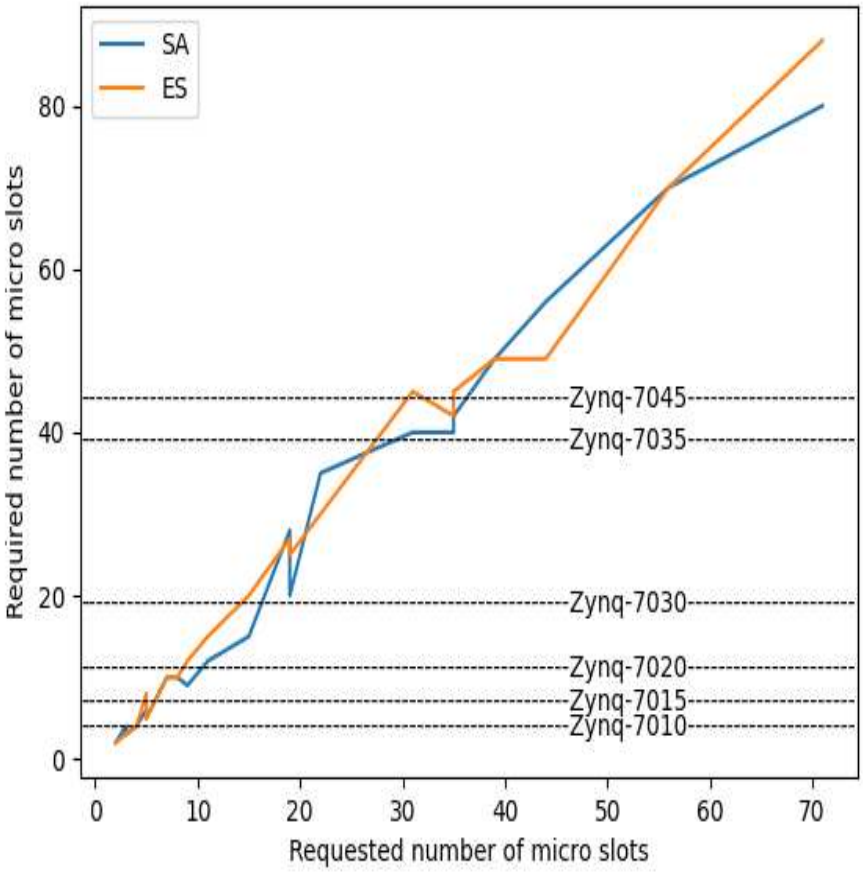


Figure 5.15: Requested number of micro slots Vs Required number of micro slots

Overall, SA algorithm has performed better than the ES algorithm throughout the

test data, with a few instances where ES has outperformed SA. Figure 5.15 shows that SA algorithm has outperformed ES algorithm until 22 requested micro slots. By this, it states that SA can perform really well when there are fewer blocks or less number of micro slots are requested. On the other hand, ES has outperformed SA from the test data which contains the total area between 42 to 58 micro slots. According to this, we can say that ES algorithm can perform better when there is more number of micro slots.

5.3.2 Decision Problem

In this section, we compare the runtime of SA and ES algorithms for solving the decision problem presented above. The goal of this comparison is to determine which algorithm is more efficient in terms of computational time for solving this particular problem.

The final output from SA and ES algorithms for the decision problem is TRUE when the algorithms have found a feasible arrangement on the selected FPGA device and FALSE when they could not find the feasible arrangement in the given time limit. Here, we considered 3 minutes for each file. For the decision problem, we have generated 60 test files. Both the SA and ES algorithm returned TRUE for 98% of the files and 2% of total files returned FALSE. We created the decision problem dataset considering the maximum number of available micro slots on Xilinx Zynq 7020 devices. With the above-mentioned results, we can say SA and ES may not be able to find a feasible arrangement for all test files which has a less or equal requirement of micro slots as the Zynq 7020 device.

Now both the SA and ES algorithms for the decision problem are compared using the runtime that algorithm took to place the given reconfigurable slots on the specific FPGA. Figure 5.16 depicts the runtime comparison between SA and ES for the decision data. It depicts that SA takes less time to place the given reconfigurable slots. On the other hand, ES takes more time to place the given reconfigurable slots.

However, for the files which do not have a feasible arrangement of blocks, SA and ES both had reached the maximum time given. We have ignored the runtime for 2% of the files in Figure 5.16 since both algorithms could not find the feasible arrangement and including this runtime with other files affect to see the insights between the runtimes for other files.

Exploration of a larger solution space by SA, which might allow it to find better solutions than ES. On the other hand, ES has taken more time to find an acceptable arrangement that is because there might be less diversity in the initial population.

It is worth noting that when there are fewer reconfigurable slots to be placed, SA may be able to explore the search space quickly and find the desired solution faster than the ES algorithm. However, there are instances where ES has performed better than SA, which could be due to the parameter settings used for the ES algorithm being better suited to the problem than those used for the SA algorithm.

Based on the results of the comparison, it is depicted that SA outperformed EA in terms of runtime for most of the instances. However, there were some instances where ES performed better than SA. The difference in performance can be attributed to factors such as the size of the solution space, the diversity of the initial population, and the parameter settings used for each algorithm.

Figure 5.16: Comparison of the runtime between SA, ES for the decision problem

6 Conclusion and Future Work

In this paper, we have addressed the layout generation problem of a 2D slot-based reconfiguration area model using optimization algorithms, especially SA and ES algorithms. We have approached the layout generation problem as two separate problems: optimization and decision problems. To evaluate the effectiveness of SA and ES algorithms, we compared their performance with heuristic and optimal algorithms through simulation experiments, considering metrics such as MBR area and runtime.

To facilitate fair comparisons, we made modifications to the optimal algorithms and referred to them as "optimal*" algorithms. Additionally, we created two different datasets, MBLA and LBMA, for the optimization problem. For the decision problem, we considered the total available micro slots on the Xilinx Zynq 7020 device. All experiments were conducted on HPC clusters, focusing on runtime and quality evaluations of SA and ES algorithms for the optimization problem.

We introduced a novel area adjustment technique aimed at generating more uniform layouts by increasing the area for specific blocks in the given data. By incorporating this technique into SA and ES algorithms, we achieved improved MBR area results for most instances.

The results showed that SA outperformed both ES and heuristic algorithms in many cases. When comparing with optimal results, SA consistently provided solutions closer to optimality compared to ES. it should be noted that both SA and ES algorithms required significant computational time to reach desirable solutions. Moreover, the runtime for these algorithms is also influenced by the number of reconfigurable slots being placed on the FPGA.

In future work, we propose exploring the combination of SA and ES algorithms to leverage their individual strengths and achieve better overall performance. This could involve developing hybrid approaches that integrate the advantages of both algorithms, leading to more efficient and effective solutions. Investigating other optimization algorithms or metaheuristic approaches may also be worthwhile for comparison and potential improvements.

Bibliography

[1] Cai-Juan Rahman Rosshairy Abd Ramli Razamin Manaf Mohammed Suhaimee Abd Ahmadieh Khanesar, Soong and Chek-Choon Ting. An evolutionary algorithm: An enhancement of binary tournament selection for fish feed formulation. 2022.

[2] Kiarash Bazargan, Ryan Kastner, Majid Sarrafzadeh, et al. Fast template placement for reconfigurable computing systems. *IEEE design & Test of Computers*, 17(1):68–83, 2000.

[3] Jürgen Becker and Reiner Hartenstein. Configware and morphware going mainstream. *Journal of Systems Architecture*, 49(4-6):127–142, 2003.

[4] E Burke, G Kendall, and G Whitwell. Meta heuristic enhances of the best-fit heuristic for the orthogonal stocking-cutting problem. Technical report, Technical Report, University of Nottingham, NOTTCS-TR-2006-3, 2006.

[5] Pei Cao, Zhaoyan Fan, Robert X. Gao, and Jiong Tang. Harnessing multi-objective simulated annealing toward configuration optimization within compact space for additive manufacturing. volume 57, pages 29–45, 2019.

[6] Bernard Chazelle. The bottomn-left bin-packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 32(08):697–707, 1983.

[7] Tung-Chieh Chen and Yao-Wen Chang. Chapter 10 - floorplanning. In Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting (Tim) Cheng, editors, *Electronic Design Automation*, pages 575–634, Boston, 2009. Morgan Kaufmann.

[8] Marcel Eckert, Dominik Meyer, Jan Haase, Bernd Klauer, et al. Operating system concepts for reconfigurable computing: review and survey. *International Journal of Reconfigurable Computing*, 2016, 2016.

[9] Ken Eguro, Scott Hauck, and Akshay Sharma. Architecture-adaptive range limit windowing for simulated annealing fpga placement. In *Proceedings of the 42nd Annual Design Automation Conference*, DAC '05, page 439–444, New York, NY, USA, 2005. Association for Computing Machinery.

[10] Abdellatif El Afia, Mohamed Lalaoui, and Raddouane Chiheb. Fuzzy logic controller for an adaptive huang cooling of simulated annealing. In *Proceedings of the 2nd International Conference on Big Data, Cloud and Applications*, BDCA'17, New York, NY, USA, 2017. Association for Computing Machinery.

[11] David E Goldberg. Genetic algorithms in search, optimization and machine learning addison welssey publishing company. *Reading, MA*, 1989.

[12] Zakarya Guettatfi, Paul Kaufmann, and Marco Platzner. Optimal and greedy heuristic approaches for scheduling and mapping of hardware tasks to reconfigurable computing devices. In Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 108–117, Cham, 2020. Springer International Publishing.

[13] Zakarya Guettatfi, Marco Platzner, Omar Kermia, and Abdelhakim Khouas. An approach for mapping periodic real-time tasks to reconfigurable hardware. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 99–106, 2019.

[14] EBCH Hopper and Brian CH Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1):34–57, 2001.

[15] Peter Eggenberger Hotz. 16 - combining developmental processes and their physics in an artificial evolutionary system to evolve shapes. In Sanjeev Kumar and Peter J. Bentley, editors, *On Growth, Form and Computers*, pages 302–318, London, 2003. Academic Press.

[16] Stefan Jakobs. On genetic algorithms for the packing of polygons. *European journal of operational research*, 88(1):165–181, 1996.

[17] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*, volume 312. Springer, 2011.

[18] Zhu Lichen, Yang Runping, Chen Meixue, Jia Xiaomin, Li Xuanxiang, and Du Shimin. An efficient simulated annealing based vlsi floorplanning algorithm for slicing structure. In *2012 International Conference on Computer Science and Service System*, pages 326–330. IEEE, 2012.

[19] Hiroshi Murata, Kunihiro Fujiyoshi, Shigetoshi Nakatake, and Yoji Kajitani. Rectangle-packing-based module placement. *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*, pages 535–548, 2003.

[20] Hiroshi Murata and Ernest S. Kuh. Sequence-pair based placement method for hard/soft/pre-placed modules. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, page 167–172, New York, NY, USA, 1998. Association for Computing Machinery.

[21] Frank G Ortmann, Nthabiseng Ntene, and Jan H Van Vuuren. New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems. *European Journal of Operational Research*, 203(2):306–315, 2010.

[22] S Prayudani, A Hizriadi, EB Nababan, and S Suwilo. Analysis effect of tournament selection on genetic algorithm performance in traveling salesman problem (tsp). In *Journal of Physics: Conference Series*, volume 1566, page 012131. IOP Publishing, 2020.

[23] Y. Rao and Q. Luo. Intelligent algorithms for packing and cutting problem.

[24] Suphachai Sutanthavibul, Eugene Shragowitz, and J. Ben Rosen. An analytical approach to floorplan design and optimization. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, page 187–192, New York, NY, USA, 1991. Association for Computing Machinery.

[25] Jesús Tabero, Julio Septién, Hortensia Mecha, and Daniel Mozos. Task placement heuristic based on 3d-adjacency and look-ahead in reconfigurable systems. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 396–401, 2006.

[26] R. Venkatraman and Lalit M. Patnaik. An evolutionary approach to timing driven fpga placement. GLSVLSI '00, page 81–85, New York, NY, USA, 2000. Association for Computing Machinery.

[27] Kristofer Vorwerk, Andrew Kennings, and Jonathan W. Greene. Improving simulated annealing-based fpga placement with directed moves. volume 28, pages 179–192, 2009.

[28] Herbert Walder, Christoph Steiger, and Marco Platzner. Fast online task placement on fpgas: free space partitioning and 2d-hashing. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2003.

[29] Meng Yang, AEA Almaini, Lun Wang, and Pengjun Wang. Fpga placement using genetic algorithm with simulated annealing. In *2005 6th International Conference on ASIC*, volume 2, pages 806–810. IEEE, 2005.

[30] Defu Zhang, Yan Kang, and Ansheng Deng. A new heuristic recursive algorithm for the strip rectangular packing problem. *Computers & Operations Research*, 33(8):2209–2217, 2006.

[31] Hao ZHENG and Yue REN. Architectural layout design through simulated annealing algorithm. In Dominik Holzer, Walaiporn Nakapan, Anastasia Globa, and Immanuel Koh, editors, *RE: Anthropocene, Design in the Age of Humans*, volume 1, pages 275–284. The Association for Computer-Aided Architectural Design Research in Asia (CAADRIA), August 2020. 25th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA 2020) : RE: Anthropocene, Design in the Age of Humans, CAADRIA2020 ; Conference date: 05-08-2020 Through 06-08-2020.

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen als Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Paderborn, 21.06.2023
Ort, Datum

Unterschrift

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Eidesstattliche Versicherung

Nachname Tadakamalla Vorname Yashwanth

Matrikelnr. 6896890 Studiengang Masters in Computer Science

○ Bachelorarbeit ◉ Masterarbeit

Titel der Arbeit

☐ Die elektronische Fassung ist der Abschlussarbeit beigefügt.

☒ Die elektronische Fassung sende ich an die/den erste/n Prüfenden bzw. habe ich an die/den erste/n Prüfenden gesendet.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit (Ausarbeitung inkl. Tabellen, Zeichnungen, etc.) selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Abschlussarbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung entspricht der gedruckten und gebundenen Fassung.

Belehrung

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist die Vizepräsidentin / der Vizepräsident für Wirtschafts- und Personalverwaltung der Universität Paderborn. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz NRW in der aktuellen Fassung).

Die Universität Paderborn wird ggf. eine elektronische Überprüfung der Abschlussarbeit durchführen, um eine Täuschung festzustellen.

Ich habe die oben genannten Belehrungen gelesen und verstanden und bestätige dieses mit meiner Unterschrift.

Ort Paderborn Datum 21.06.2023

Unterschrift 

Datenschutzhinweis:

*Die o.g. Daten werden aufgrund der geltenden Prüfungsordnung (Paragraph zur Abschlussarbeit) i.V.m. § 63 Abs. 5 Hochschulgesetz NRW erhoben. Auf Grundlage der übermittelten Daten (Name, Vorname, Matrikelnummer, Studiengang, Art und Thema der Abschlussarbeit) wird bei Plagiaten bzw. Täuschung der*die Prüfende und der Prüfungsausschuss Ihres Studienganges über Konsequenzen gemäß Prüfungsordnung i.V.m. Hochschulgesetz NRW entscheiden. Die Daten werden nach Abschluss des Prüfungsverfahrens gelöscht. Eine Weiterleitung der Daten kann an die*den Prüfende*n und den Prüfungsausschuss erfolgen. Falls der Prüfungsausschuss entscheidet, eine Geldbuße zu verhängen, werden die Daten an die Vizepräsidentin für Wirtschafts- und Personalverwaltung weitergeleitet. Verantwortlich für die Verarbeitung im regulären Verfahren ist der Prüfungsausschuss Ihres Studiengangs der Universität Paderborn, für die Verfolgung und Ahndung der Geldbuße ist die Vizepräsidentin für Wirtschafts- und Personalverwaltung.*