Hierarchical FPGA Placement

S. Areibi, G. Grewal, D. Banerji, P. Du
Computing & Information Science
University of Guelph
Guelph, Ontario
Canada N1G 2W1
January 31, 2006

Abstract

Field-Programmable Gate Arrays (FPGAs) are semiconductor chips that can realize most digital circuits on site by specifying programmable logic and their interconnections. The use of FPGAs has grown almost exponentially because they dramatically reduce design turn-around time and start-up cost for electronic products compared with traditional Application-Specific Integrated Circuits (ASICs). Efficient Computer Aided Design tools are required to compile hardware descriptions into bit-stream files that are used to configure the target FPGA to implement the desired circuits. Currently, the compile time, which is dominated by placement and routing time, can easily take hours or even days to complete for large (8-million gate) FPGAs. With 40-million gate FPGAs on the horizon, these prohibitively long compile times may nullify the time-to-market advantage of FPGAs.

This paper presents two novel placement heuristics that significantly reduce the amount of computation time required to achieve high-quality placements, compared with VPR [4]. The first algorithm is an enhancement of Simulated Annealing (SA) and attempts to solve the placement problem top-down by considering all modules at the flat level. The second algorithm is a hierarchical approach, which is based on a two-step procedure that first proceeds bottom-up (grouping highly connected modules together) and then top-down (de-clustering). The overall effect is to reduce the number of entities that need to be considered at each level, making time-consuming methods, like SA, feasible for very large problems. Our experimental results show a 70% - 80% reduction in runtime, while still achieving very high-quality placements.

Keywords — FPGA Placement, Simulated Annealing, Greedy Simulated Annealing, Multilevel Clustering.

1 Introduction

One key advantage of Field Programmable Gate Arrays (FPGAs) over full-custom and semi-custom devices is that they provide relatively quick implementation from concept to physical realization [7] (i.e., manufacturing). In addition to logic design, implementation of a design on an FPGA requires a set of integrated Computer Aided Design (CAD) tools to compile a design from its hardware description into a bit-stream that is down-loaded to the target device in order to configure it. It is important to notice that the placement phase which might dominate the compile phase time depends on several factors such as the size of the circuit, the algorithm used and the architecture of

*Is currently at the School of Engineering, University of Guelph, Canada.
the FPGA. The quality\(^1\) of the compilation greatly influences the performance of the final product. However, the compilation times for designs, which are dominated by placement and routing, \cite{27} are growing much more rapidly than the available computation power. While current CAD algorithms provide high-quality solutions, they often require great amounts of CPU time. It can easily take hours or even days to complete the compilation for large (8-million gate) chips. On the other hand, as we move to sub-micron designs, circuit delay, as well as power dissipation are mostly caused by the interconnections among logic elements\cite{15}. This problem is especially severe for FPGAs, which employ programmable switches and connections instead of metal lines to implement a net-list. Poor solutions, even derived quickly, are often not acceptable in industry. With 40-million gate FPGAs on the horizon, these prohibitively long compilations may nullify the time-to-market advantage of FPGAs. Therefore, there is a great need for CAD tools that execute in a reasonable amount of CPU time, while still generating high-quality solutions.

In this paper, we focus on the placement phase of the FPGA-based design process, with the goal of reducing the compile time to achieve high-quality placements. The most basic objective for FPGA placement, minimizing the total wire-length required to complete the routing, is used as our placement objective. Two adaptive placement algorithms are provided which dramatically reduce compile times, while still achieving high-quality placements compared with one of the state-of-the-art placement and routing packages for FPGAs, VPR \cite{4,6,18}. Our first adaptive placement algorithm, called Greedy Simulated Annealing (GSA), employs a short-term memory to record recent search history. This information is used in an unique way to improve the convergence rate of the traditionally slow SA algorithm. The overall effect is a significant reduction in time to obtain high-quality placements compared with other SA-based techniques, such as VPR. To further reduce runtime, we also use GSA as part of a hierarchical approach based on a two-step procedure that first proceeds bottom-up then top-down. The bottom-up technique uses clustering, and involves grouping of highly connected blocks into clusters and then combines clusters into larger clusters. The goal of the top-down method is to determine the locations for all the clusters, and eventually the locations of all blocks within those clusters. To date, very little work in the area of hierarchical placement for FPGA placement has been done. In this paper, we show that by using different algorithms (including GSA) at different levels, and by updating the parameters that control the search behaviour of these algorithms in a sensible way, a further reduction in runtime (compared with GSA and VPR) can be obtained without sacrificing solution quality. Another important contribution included in this paper is the introduction of a novel method to automatically determine the initial values of the starting parameters for SA, when a good initial placement is constructed.

The remainder of this paper is organized as follows: Section 2 discusses previous work in the area of FPGA placement, as well as in the area of hierarchical clustering. Section 3 describes our overall methodology including our new placement algorithm, GSA embedded within a hierarchical approach. In Section 4, we compare the results obtained using both of our new heuristics with those obtained by VPR. Finally, conclusions and future work are presented in Section 5.

2 Background

There is a wide variety of architectures for FPGAs from different vendors including Xilinx, Altera, Lattice, Actel and QuickLogic. Although the exact structure of these FPGAs varies from each other, all FPGAs consist of three fundamental components as seen in Figure 1:

1. Logic blocks that are capable of implementing multiple logic functions;

\(^1\)The quality of compilation usually refers to the total wire length or longest delay encountered.
2. I/O blocks or I/O pads for communication with the outside world, and,

3. Fixed, as well as programmable, routing resources used to realize all required inter-connections between the blocks.

The Island Style FPGA architecture is employed by many vendors. Logic blocks in this architecture are referred to as Configurable Logic Blocks (CLBs) and arranged as a symmetrical array. Routing tracks have Manhattan geometry; that is, they are either horizontal or vertical. Figure 1 shows a generic model [7] of this type of FPGA, an architecture that we assume throughout the rest of this paper.

The detailed routing structure consists of three components: connection blocks, switch blocks, and routing channels. A connection block is used to connect a CLB to the routing channels via programmable connections. The pins of each CLB pass uninterrupted through the connection block and have the option of “fusing” to some channel segments. The switch block is a switch matrix that is used to connect wires in one channel segment to other wires. Depending on the topology [8], each wiring segment on one side of a switch block may be connected to some or all of the wiring segments on the other three sides. This flexible routing structure enables every CLB to have connections with any other CLB or I/O pad, depending on the number of tracks in the routing channels. A CLB in most commercial FPGAs consists of one or more Basic Logic Elements (BLE). Each BLE usually consists of a Look Up Table (LUT) and a flip-flop, as shown in Figure 1.

2.1 CAD for FPGA Design

Implementing a design on an FPGA involves a sequence of steps, each assisted by a CAD tool. A typical design procedure employed by most commercial FPGA tools is shown in Figure 2.

A circuit hardware description is first entered to the CAD system and is converted to a netlist of basic gates using a procedure called Synthesis. The synthesized circuit is passed through
an optimizer to remove redundant logic, while maintaining its functionality. Once the design is optimized, a technology-dependent mapping tool is used to transform the basic functions into k-LUT-sized groups. In clustered FPGA architectures, a CLB consists of more than one BLE and hence Packing becomes necessary. Packing tools take advantage of the fast internal routing resources by balancing constraints including the maximum number of inputs and BLEs per logic block, aiming to combine LUTs into BLEs and then group the BLEs into logic blocks. Each logic block is then assigned a physical location on the FPGA by a placement tool. Required connections between the blocks are then realized during the following routing phase. A design is acceptable only if all connections between CLBs can be routed. Next, the implemented design is simulated to ensure its functionality, and the timing constraints are verified. Once all of the previous steps are completed successfully, the CAD tool creates a bit-stream file that is downloaded onto the target FPGA to configure the logic and interconnections.

2.2 FPGA Placement

FPGA placement usually begins with a net-list of logic blocks, which includes CLBs and I/O pads, and their interconnections. The result of placement is the physical assignment of all blocks and pads on the target FPGA, as shown in Figure 3, that minimizes one or more specific objective cost functions (wire-length, delay, power dissipation, etc.). In order for an FPGA to accommodate the design, some void (unused) CLBs and pads are usually present.

The most basic objective for FPGA placement is to minimize the routing cost, which is the total wire-length required to complete the routing. Routing cost [9] is used because reducing it...
actually reduces a number of associated design parameters. By reducing the routing length, the routing resources required by all interconnections are also reduced. This results in an increase in circuit speed due to the reduction in connection capacitance and resistance. Power consumption, which is another very important parameter to measure the quality of an FPGA implementation, is reduced too [20]. If the objective of the placement tool is to minimize the routing cost, the process is referred to as wirelength-driven placement. There are other objective terms that can be added to the original cost function to directly optimize the various design goals [22]. For example, placement can be performed to minimize the length of a critical path to meet timing constraints, referred to as timing-driven placement [17] [26], or to balance the wire density across the FPGA device, referred to as routability-driven placement [19]. In this paper, we use the wirelength-driven approach as a metric in our FPGA placement approach. In the placement phase, it is computationally expensive to determine the exact configurations of routing resources to realize physical connections for CLBs and I/O pads, which actually is another NP-hard problem. For this reason, the routing cost is approximated during placement. The speed and accuracy of estimation have a significant effect on the performance of any placement tool. The Half-Perimeter WireLength (HPWL) model is the most widely used method to estimate the wirelength of a net [24]. The wirelength is approximated by half the perimeter of the smallest bounding rectangle that encloses all terminals in the net, as shown in Figure 4.

In a Manhattan routing structure, the HPWL of a net approximates the length of a Steiner tree, which is the lowest bound on the final routing cost on a net. Given a block \( b \) with coordinates \((x_b, y_b)\), the half-perimeter of net \( i \) is calculated as follows:

\[
HPWL_i = (\max_{b \in i}\{x_b\} - \min_{b \in i}\{x_b\} + 1) + (\max_{b \in i}\{y_b\} - \min_{b \in i}\{y_b\} + 1)
\]  

(1)

For a net with two or three terminals, the routing cost obtained by HPWL model is accurate. When there are more than three terminals in a net, a \( q(i) \) factor [10] is introduced to compensate for the fact that the HPWL model under-estimates the wirelength necessary to connect all blocks. The value of \( q(i) \) depends on the number of terminals in net \( i \). The parameter \( q(i) \) is 1 for nets with 3 or fewer terminals, and gradually increases to 2.79 for nets with 50 terminals. For exceptionally heavy fanout nets that have more than 50 terminals, the value of \( q(i) \) increases linearly [5] at the rate of:

\[
q(i) = 2.7933 + 0.02616 \times (\text{Terminal Number} - 50)
\]  

(2)

Therefore, the final cost function, called the bounding box cost, takes the following form:
\[ \text{Cost}_{\text{bounding box}} = \sum_{i=1}^{N_{\text{nets}}} q(i) \ast HPWL_i \]  

Consequently, the FPGA placement pertaining to this paper is equivalent to the problem of minimizing the bounding box cost.

### 2.3 Previous Techniques for FPGA Placement

FPGA placement is an \( NP \)-hard combinatorial optimization problem, hence no polynomial-time algorithm is known to produce an exact solution [24]. In recent years, many heuristic techniques have been developed in an attempt to obtain (sub-optimal) solutions in a reasonable amount of time.

Historically, these methods have been divided into two classes: **partitioning-based** placement [14, 13] and **iterative improvement** [4, 6]. In partitioning-based placement, a circuit is recursively bi-sected, minimizing the number of *cuts* of the nets that connect components between partitions, while leaving highly-connected blocks in one partition. Eventually, the partition size reaches a few blocks to obtain improvement by grouping the highly-connected blocks together. These kind of methods are good from a “global” perspective, but they do not directly attempt to minimize wirelength. Therefore, the solutions obtained are sub-optimal in terms of wirelength. However, they run fast, and are normally used in conjunction with other search techniques, such as local search, for further quality improvement.

Iterative improvement methods start with an initial placement and seek improvements by searching for small perturbations in the *neighbourhood* of the placement that results in better solutions. For FPGA placement, these perturbations are location swaps (pairwise move) between two blocks (either two CLBs or two I/O pads).

In local search methods [1], only moves that tend to improve the current solution are accepted. Placement heuristics in this category can run fast if well implemented. The weakness of these methods is related to the fact that they can easily get trapped in local minima. When the number of local minima is large, which actually seems to be a generic feature of many of the classic \( NP \)-hard problems including FPGA placement [1], the probability that these heuristics converge to a global minima is extremely small.

Meta-Heuristics, in the form of Simulated Annealing (SA), improve the performance of basic local search by allowing *hill-climbing* (i.e., accepting moves that would deteriorate the objective function) to escape local optima, which usually cause the latter heuristic to terminate. Besides accepting beneficial swaps, moves that deteriorate the solution will be accepted in SA with a probability of \( e^{-\Delta C/T} \), where \( \Delta C \) is the change in cost, and \( T \) is analogous to temperature in the metal crystallization process. The change of \( T \) is referred to as an *annealing schedule*. Initially, \( T \) is set to a high value such that most inferior solutions can be accepted. As the *annealing* process continues, \( T \) gradually decreases (cooling), reducing the probability of accepting poor solutions. In the final stage, \( T \) usually is only a small fraction of its original value and almost only improving solutions are allowed. Currently, SA-based placement heuristics, like VPR [4, 6], have achieved similar or higher quality solutions, compared to other types of placement tools. However, this improvement often comes at the expense of longer runtime [18].

A more recent approach is to reduce the complexity of large circuits by *clustering* them into less complicated and easily solvable forms, which helps to decrease the time required to obtain good solutions for the overall problem. This approach has been applied successfully to circuit partitioning [16] and VLSI standard-cell placement [3, 25]. In many cases, a decrease in computation time by
an order of magnitude compared to manipulating the flat net-list is reported [3, 16, 25]. Only recently has this approach been applied to the FPGA placement problem [23], and then only in a limited way. Early methods of clustering were applied only to a single level. As the circuit sizes have grown larger, recent research [2, 3] has shown that adding extra levels of clustering, referred to as multilevel clustering, is more manipulable and produces superior results.

3 Overall Methodology

The overall placement approach is a two-step procedure, as shown in Figure 5, first proceeding bottom-up then top-down. The bottom-up technique involves grouping highly-connected blocks into clusters. Ideally, a clustering method will convert the original circuit into a similar but much smaller structure, with many entities combined into one large cluster. Next, a top-down method is applied to globally determine the locations for all the clusters. The simplified problem makes the use of a traditionally time-consuming method, like the SA, more feasible. A de-clustering process proceeds to restore the original FPGA layout according to the previous placement result of clusters. In this procedure, the flattened blocks should be placed as close to their center-of-gravity [3, 23] as possible. Finally, a localized improvement heuristic is executed to move blocks in small regions to achieve the final placement.

With multilevel clustering, compaction of the original net-list can proceed more gradually and produce more gentle clusters at each level. In the coarse (high) level, a top-down method places blocks in the appropriate regions that they should belong to. As the refinement moves from coarse to fine (low) level, the small sized clusters and more detailed steps enable a localized heuristic to find a good final solution [2]. Furthermore, during de-clustering the difference between the positions of clustered blocks and the flat circuit blocks can be substantial, and significant future refinement is thus necessary. In a multilevel approach, smaller blocks usually result in a much smaller difference, which assists superior quality solutions to be achieved in a shorter amount of time [3].
A large portion of improvement should be done at the highest level of the hierarchy, taking advantage of the reduced solution space facilitated by clustering. Therefore a high quality technique, such as the new proposed Greedy Stochastic Algorithm (GSA), will be able to place clusters to obtain a globally good placement in a short amount of time. With proper top level placement, blocks should have been assigned to their proper locations in the intermediate levels. The major objective in these levels is to gradually improve the solution quality. A simple yet effective stochastic method is used to complete this task. Finally, fine-tuned search is performed at the flat level of the hierarchy. Utilizing appropriate improvements at the previous levels, blocks in this level should be placed close to their final location; a localized placement method is therefore preferred.

3.1 Greedy Stochastic Algorithm (GSA)

The inspiration for GSA comes mainly from meta heuristic techniques such as Simulated Annealing and Tabu Search. A short-term memory, which records recent search history, is employed to help the new placement algorithm [11] converge more quickly than conventional Simulated Annealing. We employ a “first improving or least non-improving among $D_{greedy}$ neighbouring moves” search strategy, where the parameter $D_{greedy}$ is an integer. The detailed methodology of the algorithm is illustrated in Figure 6. First, an initial (legal) placement is constructed (randomly) within the confinement of the target FPGA. Next, two logic blocks (two CLBs or two I/O pads) are selected (randomly) to create a new solution in the neighbourhood as shown by Figure 7.

If the swap results in a decrease in cost, the move is always accepted and the locations of these two blocks are switched. However, if the move results in an increase in cost, the two blocks together with the change in cost are recorded in a short-term memory. Otherwise, the change in cost of this move is compared with corresponding value already recorded in the memory to preserve the better or less deteriorating move. A large number of such moves are made and evaluated to gradually obtain improvements. The number of continuous non-improving moves is counted until the number reaches the value specified by the parameter $D_{greedy}$. Next, the move recorded in the history is mandatorily accepted even though it makes the placement worse. Additionally, the contents of the memory are erased once a move is accepted (including the acceptance of improving moves).

The parameter $D_{greedy}$ which is used to select the best move from the set of non-improving moves is a key parameter in controlling the aggressiveness of our placement tool. Initially, $D_{greedy}$ is set to a small value such that non-improving moves are easily accepted; however, $D_{greedy}$ is gradually increased as the placement is refined so that eventually the probability of accepting a move that makes the placement worse is very low. The ability to accept non-improving moves allows GSA to escape local minima, while still converging quickly.

The update of $D_{greedy}$, the exit criterion to terminate the search, the number of moves attempted at each $D_{greedy}$ and a method to restrict the generation of potential moves are defined by an update schema, which maintains a good balance between the search strategies of exploration and exploitation.

3.1.1 Adaptive Update Schema

An FPGA placement algorithm should perform well over a wide range of circuits. “Fixed” update schemas usually work well within the narrow application range for which they are developed. However, they are not sufficiently generalized to adapt to different problems [12]. It is thus imperative to explore adaptive update strategies.

Let $N_{blocks}$ be the total number of CLBs and I/O pads in the circuit. The number of pairwise
moves attempted at each value of $D_{\text{greedy}}$ in GSA is set to:

$$\text{MovesPer}D_{\text{greedy}} = \text{innerNum} \times (N_{\text{blocks}})^{4/3}$$  \hspace{1cm} (4)$$

where the parameter $\text{innerNum}$ can be overridden on the command line to allow different trade-offs between runtime and placement quality. Reducing the value of $\text{innerNum}$ has the effect of reducing the runtime at the expense of quality. In contrast, increasing $\text{innerNum}$ would consume more CPU time to obtain higher quality solutions. In addition, adaptively limiting the scope of moves ($R_{\text{limit}}$) within the region of the original block positions, has been shown to give superior results compared to allowing unrestricted moves [4, 6]. A simple example is given in Figure 8 where $R_{\text{limit}}$ is set to one. The net effect is that the source block can only swap positions with candidate blocks inside the designated square window.

Figure 6: Pseudo-code for GSA
CLB4 and CLB5 are (randomly) selected to perform an attempted swap. FPGA layout after accepting the swap between CLB4 and CLB5.

![Diagram of FPGA layout](image)

Yes

Accept this swap?

No

Discard this attempted swap.

![Diagram of FPGA layout](image)

Figure 7: Create a Neighbouring Solution by Swapping two CLBs.

The update of $D_{greedy}$ and $R_{limit}$ is determined by the parameter $S_{best}$, which records the best solution obtained so far during the search procedure. The value of $S_{best}$ is set to the initial placement cost. Next, a new placement cost $S_{current}$ is obtained after each $MovesPerD_{greedy}$ moves have been evaluated. The difference in cost, $\Delta S = S_{current} - S_{best}$ is calculated. If $\Delta S$ is negative, it means that a new lower cost placement has been found, indicating that the current values of $D_{greedy}$ and $R_{limit}$ are still fruitful. Therefore, no changes are made to the values of $D_{greedy}$ and $R_{limit}$; these values are used to proceed with another round of search. Only the parameter $S_{best}$ is updated in this case. A non-negative value of $\Delta S$ indicates that the current parameters are no longer yielding improving solutions and need to be updated. Guided by the value of $\Delta S$, GSA adaptively adjusts parameters according to each circuit.

$D_{greedy}$ and $R_{limit}$ in GSA are two interactive parameters used to control the balance between exploration and exploitation during a search. Initially, $D_{greedy}$, which determines the aggressiveness of the search procedure, is set to two to ensure that the maximum number of attempted evaluations are accepted. At the same time, $R_{limit}$, which limits the movement of blocks within specific regions, is set to the whole span of the chip to allow for maximum exploration. Whenever an update of $D_{greedy}$ and $R_{limit}$ is necessary ($\Delta S \geq 0$), a new value of $D_{greedy}$ is computed as $D_{greedy\_new} = [\alpha \times D_{greedy\_old}]$, where the value of $\alpha$ depends on:

10
\[ \alpha = \begin{cases} 
1.5, & \text{if } D_{\text{greedy}} \leq 10 \\
1.3, & \text{else if } R_{\text{limit}} = 1 \\
1.05, & \text{otherwise.} 
\end{cases} \tag{5} \]

At the same time, the value of \( R_{\text{limit}} \) is revised as follows: \( R_{\text{limit}} \new = [\beta \times R_{\text{limit}}\old] \) and limited to the range \((1 \leq R_{\text{limit}} \leq \text{maximum FPGA span})\). If the value of \( R_{\text{limit}} \) becomes less than 1, it is set to 1; otherwise. The value of \( \beta \) is determined as follows:

\[ \beta = \begin{cases} 
1, & \text{if } D_{\text{greedy}} \leq 10 \\
0.9, & \text{otherwise.} 
\end{cases} \tag{6} \]

This strategy causes \( R_{\text{limit}} \) to cover the entire chip for the first phase of the search, shrinking gradually during the middle stages of the search, and finally settling at “1” (logic block distance) during the final phase of the search.

When \( D_{\text{greedy}} \) is very small \((\leq 10)\), non-improving moves are easily accepted. In this phase, the algorithm mainly explores the solution space in search of the region containing the global optimum. At the same time, \( R_{\text{limit}} \) is maximized to enable free block movements. At the end of placement stage, very few attempted moves can be accepted. This is because the value of \( D_{\text{greedy}} \) is very large and the current placement is of fairly high quality. This is a pure exploitation stage and can be identified when \( R_{\text{limit}} \) equals 1. With this in mind, we create an update schedule to relatively increase the amount of CPU time (small \( \alpha \)) spent in the more “productive” intermediate stage of the search, which represents a combination of exploration and exploitation.

The performance of GSA with different combinations of \( \alpha \) and \( \beta \) \((\text{innerNum} = 5)\) is shown in Figure 9. The exact values of \( \alpha \) and \( \beta \) listed here were determined imperically (through experimentation). Even though the values of \( \alpha \) and \( \beta \) lead to the best trade-offs between CPU time and placement quality, GSA is not overly sensitive to slight variations in their values. We termi-
nate the placement process adaptively by making use of $S_{\text{best}}$. When the value of $R_{\text{limit}}$ equals 1, the placement algorithm terminates if $S_{\text{best}}$ cannot be updated in 5 consecutive $\text{MovesPer} D_{\text{greedy}}$ evaluation rounds. In this pure exploitation stage, little improvement can be achieved, and it is unlikely any further benefit can be obtained.

### 3.2 Hierarchical Approach

The framework of the hierarchical algorithm is shown in Figure 10. The first stage is a multi-level bottom-up clustering of the logic blocks, as illustrated by Figure 11. This stage tends to group highly connected blocks into clusters. These clusters can be grouped again to create a second level of clustering, and so on. In this example, $\text{net}_1$ is absorbed (eliminated) and terminals of $\text{net}_2$ are reduced from 3 to 2. Optimized clustering not only reduces the number of modules in the circuit but also reduces the complexity of interconnections.

![Figure 10: Framework of Hierarchical Placement](image)

![Figure 11: Net Absorption and Terminal Reduction](image)

Once the clustering stage is complete, placement is performed at each level of the hierarchy. The placement heuristics performed at each level tend to place blocks in the appropriate regions.
that they should belong to. A de-clustering process (which decomposes clusters into logic blocks or clusters of lower levels) follows when the top-down optimization is performed to each level in the hierarchy. Additionally, a local improvement is performed during de-clustering to place blocks close to their “center-of-gravity” [3, 23] (as shown in Figure 12) to achieve further improvement. The main objective of this phase is to optimize the location of blocks in clusters to minimize wirelength during de-clustering. Finally, a localized heuristic, which only moves blocks in small regions, will be performed at the bottom (flat) level to achieve a final high-quality solution.

![Figure 12: De-clustering Technique Used in Overall Approach.](image)

3.2.1 Improvement Techniques

The circuit at the top level of clustering is considered to be the least complex representation of the original circuit, and therefore an advanced fast converging heuristic technique, GSA (innerNum = 1), is used to place clusters efficiently in a short period of time. Similar to a global placement tool in standard-cell design, the GSA placement tool roughly determines the best position of each block. Intermediate and bottom level improvements follow to find exactly where each block should be for the final high-quality placement.

When the total number of hierarchical levels is more than two, extra improvement should be performed at the intermediate levels. The major objective at these levels is to gradually improve the solution quality. This is achieved by using a simple yet effective stochastic local search algorithm, which is equivalent to Simulated Annealing with initial temperature set to 0 [23].

Fine-tuned search should be performed at the flat level of the hierarchy, since it is an accurate representation of the actual circuit. A low temperature Simulated Annealing Algorithm (innerNumsa = 5), together with proper Rlimit, is utilized on the flattened circuit to optimize the final local ordering of the logic blocks. With the help of the parameter Rlimit, which shrinks when temperature T becomes low, the algorithm tends to place blocks in fine defined regions.

3.2.2 Choice of Starting Parameters for SA

One crucial feature of a dynamic adaptive annealing schedule for a variety of circuits is the choice of starting parameters for a given good configuration, which is the placement resulting from the previous improvement in higher levels. Parameter tuning, in VPR SA, involves the start temperature T0 and the initial size of window limiter Rlimit. If T0 is set too high or Rlimit is set too large, subsequent annealing will destroy the existing good placement structure, which makes any previous work toward placing the circuit useless. Conversely, if T0 is set too low or Rlimit is set too small, the annealer is unlikely to improve upon the existing placement significantly, as it will be unable to escape local minima.
In [21], a concept, which is proposed to compute a good starting temperature \( T_0 \) for simulated-annealing placements, is introduced. The idea evolves around an initial temperature where the placement is in a state of “equilibrium”. In this state, there is no expected net change in the cost function after a large number of moves, which implies that the expected change in placement cost is zero. Unfortunately, the parameter \( R_{\text{limit}} \) is not considered in their work which is so important and should not be neglected.

In this paper, we propose a new method to calculate the equilibrium state defined in the work of [21], in which there is no expected net change in the cost function after a large number of moves. It resembles the annealing schedule used by VPR SA with the exception of having a fewer number of moves attempted at each temperature.

Starting from a good initial placement, this method performs \( N \) evaluations at each temperature, none of which are actually permitted to change the placement. The criteria of accepting or rejecting a move is based on the same criteria of VPR. If a move is “accepted”, the change in cost associated with this move is recorded and accumulated in the parameter \( V_{\text{addup}} \). Accepting a bad move increases the value of \( V_{\text{addup}} \), while an improvement move, which is always “accepted” would reduce it. At the end of each \( N \) evaluations, the value of \( V_{\text{addup}} \) is assessed.

When temperature \( T \) is high enough, bad moves are accepted with high probability leading to a high value of \( V_{\text{addup}} \). Temperature \( T \) and window limiter \( R_{\text{limit}} \) are updated with the same method as VPR to ensure the best combination. As the value of \( T \) decreases, the heuristic becomes more and more greedy: favouring moves that would decrease the objective function. Once negative overall changes in placement cost \( V_{\text{addup}} \) is reached, the present temperature \( T \) and \( R_{\text{limit}} \), which would render the current placement in a state of equilibrium, are set to a suitable value of initial parameters, for the following actual annealing.

It is important to ensure that enough moves per temperature are made to obtain an accurate probability distribution. In our implementation, the value of \( N \) is set equal to the value of \( N_{\text{block}} \), which is the total number of logic blocks plus the number of I/O pads in a circuit. Detailed implementations of this work can be found in [11].

### 3.3 Clustering Parameters

We performed experiments for clustering depth \( L = 1, 2, 3 \) and 4 levels\(^2\) with different sizes \( S \), where cluster size \( S \) denotes the number of blocks per cluster. Additionally, once \( S \) is set to \( n \), then \( n \) is the clustering size for every clustering level.

The curves of the average final placement cost as well as average CPU time over ten MCNC benchmark circuits [28] versus clustering size and clustering depth are shown in Figure 13 and Figure 14, respectively.

We can observe from Figure 13 that two schedules \( (L = 2, S = 4) \) and \( (L = 3, S = 2) \) produce the best final quality. However, Figure 14 shows that the first schedule \( (L = 2, S = 4) \) consumes less CPU time than the latter \( (L = 3, S = 2) \). Consequently, it is the best selection for our hierarchical approach. Nevertheless, the performance of the placement tool is not extremely sensitive to this specific schedule. As long as we keep the clustering depth \( L \) to 2 and \( S \) between 2 and 7, the placement quality and CPU time curve appear quite even.

---

\( ^2 \) \( L = 0 \) denotes the flat level.
Figure 13: Average “Bounding Box Cost” vs. Clustering Size with Different Clustering Depth.

Figure 14: Average “CPU time” vs. Clustering Size with Different Clustering Depth.
4 Computational Results

We make a fair evaluation of how well our FPGA placement tool performs with respect to both runtime and placement quality compared to VPR [5], by running all algorithms on the same platform with the same suite of benchmark circuits and the same FPGA architecture. VPR is considered to be the state-of-the-art academic tool for FPGA placement and routing. It is used extensively by almost all researchers to validate their results [9]. However, it is important to notice that we are not comparing our results with the latest revision of VPR due to late releases of this tool (i.e. current revisions are used by commercial products in industry).

In our approach, an island style FPGA model, with each CLB containing a single 4-input lookup table (4-LUT) and a single D flip-flop is used. The I/O pad pitch-to-logic block ratio, which is the number of pads available at each marginal block location, is set to 2. Each CLB has 6 pins: 4 inputs, 1 output, and 1 clock (global), and we assume the FPGA has dedicated resources for routing the clock, reset and other global nets.

Several MCNC [28] benchmarks, shown in Table 1, are used to measure the performance of every heuristic. This suite consists of circuits ranging from a few hundred CLBs to nearly ten thousand CLBs and is organized into three groups: small, medium and large. The “FPGA matrix” column is the actual CLB matrix of target FPGA, which is the minimum size to hold the design.

<table>
<thead>
<tr>
<th>Circuit name</th>
<th>FPGA matrix</th>
<th>Number of CLBs</th>
<th>Number of I/O Pads</th>
<th>Number of nets</th>
<th>Average fanout</th>
</tr>
</thead>
<tbody>
<tr>
<td>e64</td>
<td>17x17</td>
<td>274</td>
<td>130</td>
<td>290</td>
<td>3.94</td>
</tr>
<tr>
<td>tseng</td>
<td>33x33</td>
<td>1047</td>
<td>174</td>
<td>1099</td>
<td>4.28</td>
</tr>
<tr>
<td>ex5p</td>
<td>33x33</td>
<td>1064</td>
<td>71</td>
<td>1072</td>
<td>4.73</td>
</tr>
<tr>
<td>alu4</td>
<td>40x40</td>
<td>1522</td>
<td>22</td>
<td>1536</td>
<td>4.52</td>
</tr>
<tr>
<td>seq</td>
<td>42x42</td>
<td>1750</td>
<td>76</td>
<td>1791</td>
<td>4.46</td>
</tr>
<tr>
<td>frisc</td>
<td>60x60</td>
<td>3556</td>
<td>136</td>
<td>3576</td>
<td>4.48</td>
</tr>
<tr>
<td>spla</td>
<td>61x61</td>
<td>3690</td>
<td>62</td>
<td>3706</td>
<td>4.73</td>
</tr>
<tr>
<td>ex1010</td>
<td>68x68</td>
<td>4598</td>
<td>20</td>
<td>4608</td>
<td>4.49</td>
</tr>
<tr>
<td>s38584.1</td>
<td>81x81</td>
<td>6447</td>
<td>342</td>
<td>6485</td>
<td>4.18</td>
</tr>
<tr>
<td>clma</td>
<td>92x92</td>
<td>8383</td>
<td>144</td>
<td>8445</td>
<td>4.61</td>
</tr>
</tbody>
</table>

Table 1: MCNC Benchmark circuit suite used as test cases

Our hierarchical methodology is implemented in C++ using the GNU g++ compiler, while VPR is implemented in C using the GNU gcc compiler. All experiments were conducted on a Sun Sparc 10 dual CPU workstation with Solaris 8.0 operating system. Furthermore, our results are obtained by running both placement tools 5 times with different initial (random) placements to produce a more meaningful comparison.

Table 2 shows the cumulative improvement of placement from no improvement (random start) to our hierarchical approach, with the cluster parameter \((L = 2, S = 4)\). The column, “Rand.clustering” stands for random grouping of blocks to clusters without optimization, and “Rand.de-clustering” indicates that clusters were randomly de-clustered without optimization. “Stochastic simple local search” stands for using this method (the same as the local search in intermediate levels) in all clustering levels. The cumulative value is obtained by the comparison between two adjacent techniques, listed in the columns. For example, the cumulative improvement of the hierarchical

---

3The original VPR “linear congestion” cost function, which is used when routing channel is irregular, is changed to pure bounding box cost function.
approach (43.3%) is based on the already optimized placement done by implementing stochastic simple local search at each hierarchical level. The table clearly illustrates the “contribution” of each technique in our overall methodology. Furthermore, the overall improvements over initial random placement by each techniques are also given in Table 2.

<table>
<thead>
<tr>
<th>Circuit name</th>
<th>Rand. clustering &amp; rand. de-clustering</th>
<th>Optimized clustering &amp; rand. de-clustering</th>
<th>Optimized clustering &amp; optimized de-clustering</th>
<th>Stochastic simple local search</th>
<th>Hierarchical approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ave. cost</td>
<td>Ave. CPU time(s)</td>
<td>Ave. cost</td>
<td>Ave. CPU time(s)</td>
<td>Ave. cost</td>
</tr>
<tr>
<td>et4</td>
<td>742</td>
<td>0</td>
<td>6271</td>
<td>0.01</td>
<td>5443</td>
</tr>
<tr>
<td>tseng</td>
<td>41285</td>
<td>0.02</td>
<td>25431</td>
<td>0.08</td>
<td>23378</td>
</tr>
<tr>
<td>ex5p</td>
<td>42301</td>
<td>0.02</td>
<td>32653</td>
<td>0.09</td>
<td>29856</td>
</tr>
<tr>
<td>alu4</td>
<td>61504</td>
<td>0.08</td>
<td>42550</td>
<td>0.22</td>
<td>39624</td>
</tr>
<tr>
<td>seq</td>
<td>79903</td>
<td>0.03</td>
<td>59268</td>
<td>0.13</td>
<td>55571</td>
</tr>
<tr>
<td>M.avg.</td>
<td>56248</td>
<td>0.04</td>
<td>39975</td>
<td>0.14</td>
<td>37107</td>
</tr>
<tr>
<td>frisc</td>
<td>229152</td>
<td>0.19</td>
<td>132999</td>
<td>0.38</td>
<td>144279</td>
</tr>
<tr>
<td>spla</td>
<td>236521</td>
<td>0.11</td>
<td>169741</td>
<td>0.35</td>
<td>159422</td>
</tr>
<tr>
<td>ex1010</td>
<td>328664</td>
<td>0.21</td>
<td>250872</td>
<td>0.67</td>
<td>240335</td>
</tr>
<tr>
<td>s8954.1</td>
<td>598570</td>
<td>0.38</td>
<td>335429</td>
<td>3.40</td>
<td>320052</td>
</tr>
<tr>
<td>clma</td>
<td>796991</td>
<td>0.52</td>
<td>548932</td>
<td>2.79</td>
<td>528772</td>
</tr>
<tr>
<td>L.avg.</td>
<td>439090</td>
<td>0.28</td>
<td>291594</td>
<td>1.56</td>
<td>278572</td>
</tr>
<tr>
<td>Avg.</td>
<td>238697</td>
<td>0.15</td>
<td>162424</td>
<td>0.83</td>
<td>154703</td>
</tr>
<tr>
<td>Avg. cumulative impro.</td>
<td>-</td>
<td>-</td>
<td>+32.0%</td>
<td>-43.0%</td>
<td>+4.5%</td>
</tr>
<tr>
<td>All impro.</td>
<td>-</td>
<td>-</td>
<td>+32.0%</td>
<td>-35.2%</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Contribution of each improvement technique of our overall methodology ($L = 2, S = 4$)

4.1 Analysis

We plot one search curve (placement quality vs. number of attempted evaluations) of VPR and GSA for a large circuit in Figure 15.

The search curve of GSA converges much faster than that for VPR, while the quality of final placement is very close to that obtained by VPR. Therefore, by terminating GSA much sooner than VPR, we can obtain high-quality placements in significantly less time. It should be emphasized that although the graph is for a single circuit, the picture it gives is very typical of other circuits. Figure 16 illustrates a typical overall behaviour of a two-level hierarchical placement over a large MCNC [28] circuit.

This example starts from a random placement where all blocks have already been collapsed into each cluster. The top level placement is completed by the GSA placement tool, which quickly discovers a good local optimum at this level. However, this bounding box cost is an approximation of the flat counterpart and they are “equalized” by the following de-clustering procedure, which results in a cost increase. The VPR simulated-annealing algorithm is employed at the flat level with proper initial parameters to ensure a “smooth” transfer between different heuristics.

The innerNum of GSA can be overridden through the command line, allowing user-controlled trade-offs between placement quality and CPU time. Our suggested innerNum for GSA is 5; for comparison we increase it to 10 (exhaustive version) to obtain better placement quality with longer runtime.

Table 3 shows that GSA with innerNum = 5 and GSA with innerNum = 10, as well as our hierarchical approach produced almost the same placement quality over the test suite as VPR, while our tools are significantly faster.
Figure 15: Comparison between GSA and VPR using a Large Size Circuit “clma”.

Figure 16: Hierarchical Approach Behaviour Based on a Large Circuit “spla”.
<table>
<thead>
<tr>
<th>Circuit name</th>
<th>VPR default</th>
<th>GSA InnerNum = 5</th>
<th>GSA InnerNum = 10</th>
<th>Hierarchical approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ave. cost</td>
<td>Ave. CPU time(s)</td>
<td>Ave. cost</td>
<td>Ave. CPU time(s)</td>
</tr>
<tr>
<td>e64</td>
<td>2858</td>
<td>13</td>
<td>2872</td>
<td>3</td>
</tr>
<tr>
<td>tseng</td>
<td>9394</td>
<td>71</td>
<td>9444</td>
<td>19</td>
</tr>
<tr>
<td>ex5p</td>
<td>16227</td>
<td>70</td>
<td>16280</td>
<td>18</td>
</tr>
<tr>
<td>alu4</td>
<td>19161</td>
<td>104</td>
<td>19434</td>
<td>30</td>
</tr>
<tr>
<td>seq</td>
<td>24736</td>
<td>142</td>
<td>24866</td>
<td>39</td>
</tr>
<tr>
<td>M.avg.</td>
<td>17380</td>
<td>97</td>
<td>17506</td>
<td>27</td>
</tr>
<tr>
<td>frisc</td>
<td>52156</td>
<td>392</td>
<td>52368</td>
<td>125</td>
</tr>
<tr>
<td>spla</td>
<td>61046</td>
<td>414</td>
<td>60722</td>
<td>135</td>
</tr>
<tr>
<td>ex1010</td>
<td>65493</td>
<td>554</td>
<td>65898</td>
<td>197</td>
</tr>
<tr>
<td>s38584.1</td>
<td>64925</td>
<td>905</td>
<td>65674</td>
<td>325</td>
</tr>
<tr>
<td>clma</td>
<td>140391</td>
<td>1332</td>
<td>140878</td>
<td>574</td>
</tr>
<tr>
<td>L.avg.</td>
<td>76802</td>
<td>720</td>
<td>77108</td>
<td>271</td>
</tr>
<tr>
<td>Avg.</td>
<td>45639</td>
<td>400</td>
<td>45844</td>
<td>146</td>
</tr>
<tr>
<td>Avg.impro.</td>
<td>-</td>
<td>-</td>
<td>-0.53%</td>
<td>+69.3%</td>
</tr>
</tbody>
</table>

Table 3: Performance of VPR, GSA and Hierarchical Approach

When innerNum is set to 5, which is the default value in our implementation, on average the GSA achieves 69% reduction in CPU time (3.2x faster) compared with VPR at the cost of only a slight (0.5%) decrease in the final placement quality. When innerNum is increased to 10 to trade CPU time for better quality, GSA obtains a slight gain of 0.4% in placement quality with 41% reduction (1.7x faster) in CPU time, compared to VPR. For our hierarchical approach, on average, it achieves 79% reduction in CPU time (4.8x faster) at the cost of a slight (less than 2%) increase of the final placement cost.

5 Conclusions

The time to compile current Field Programmable Gate Arrays (FPGAs) can easily take hours or even days to complete for large chips, which may nullify the time-to-market advantages of FPGAs. In this paper, two new approaches for the FPGA placement problem have been presented and investigated. Both methods greatly reduce the execution time of FPGA placement, while achieving the same quality of placement, when compared to one of the state-of-the-art placement tools, VPR.

In our future work we plan to use a constructive technique (GRASP or based on analytical mathematical technique) to create a good initial starting point at the highest level of the hierarchy, with the hope of further improving the convergence of the algorithm. In addition, we also plan to integrate inter-cluster as well as intra-cluster routing issues into our placement tool.

References


