

Benchmarking the Independence Architecture Adaptive Placer on the Triptych FPGA Architecture

Peter Grossmann

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science
In Electrical Engineering

University of Washington

2006

Program Authorized to Offer Degree:
Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Peter Grossmann

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made

Committee Members:

Scott Hauck

W. Carl Ebeling

Date: _____

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature _____

Date _____

University of Washington

Abstract

Benchmarking the Independence Architecture Adaptive Placer on the Triptych FPGA Architecture

Peter Grossmann

Chair of the Supervisory Committee
Associate Professor Scott Hauck
Electrical Engineering

The ability to evaluate new FPGA architectures is inherently limited by the ability of software tools to support them. This motivated the development of Independence, an architecture-adaptive FPGA placement tool. Previous work demonstrated that Independence adapts to a variety of architectures without significant degradation in required area when compared to placement tools targeting these architectures. Independence achieves this by using a routing-driven cost function that includes a congestion estimate as well as wirelength. This thesis compares Independence on the Triptych architecture to a custom Triptych placement tool that uses the cost function presented in [Ebeling95]. Unlike the tools previously compared to Independence, this cost function accounts for congestion. The new custom placer is shown to produce placements of similar required area to the original [Ebeling95] tool. Compared to this custom placement tool, Independence requires 15.6% more area for 3-input RLB architectures and 17.8% more area for 4-input architectures.

TABLE OF CONTENTS

	Page
List of Figures	ii
List of Tables	v
1 Introduction.....	1
2 FPGA Background.....	3
2.1 FPGA Architectures.....	3
2.1.1 Island-Style FPGAs.....	3
2.1.2 HSRA.....	6
2.1.3 RaPiD.....	7
2.2 FPGA Place-and-Route Software.....	8
2.2.1 FPGA Placement.....	8
2.2.2 FPGA Routing.....	12
3 Independence.....	15
3.1 Independence Placer.....	15
3.2 Previous Benchmark Results.....	18
4 Triptych Architecture.....	19
4.1 Triptych Logic Resources.....	19
4.2 Triptych Interconnect.....	20
4.3 Alternate Triptych Architecture Styles.....	23
4.4 Triptych Placement Software.....	25
5 Triptych Placer Results.....	29
5.1 Determining the Triptych Placer's Parameters.....	29
5.2 Quality Versus Original Triptych Placer.....	32
5.3 Tuning Independence.....	35
5.4 Comparison to Independence.....	37
6 Conclusion.....	43
7 Future Work.....	44
References.....	45
Appendix: Data Tables for Triptych, Independence Tuning Parameters.....	47

LIST OF FIGURES

Figure Number	Page
Figure 1: Representation of an island-style FPGA [Sharma05]. The white boxes denote CLBs. Vertical and horizontal lines show routing channels. The Xs indicate points within a switchbox where data can change direction.	4
Figure 2: Block diagram of a simple island-style CLB. This CLB contains a four-input LUT whose output is connected to a D flip-flop and a multiplexer input. The flip-flop output is also connected to the mux so that the CLB output can be configured to be either combinational or sequential. Wires used to program the CLB are not shown.	4
Figure 3: A more advanced hypothetical CLB. The overall CLB accepts eight arbitrary inputs and produces two outputs. One additional input (top) and one additional output (bottom) are shown to represent dedicated carry-in/carry-out lines to be used when the CLB is configured for fast addition. The two leftmost LUTs can be configured to be used separately to compute two four-input functions or combined with the third LUT on the right to compute one eight-input function. Optional flip-flop storage is provided for each.	5
Figure 4: Close-up look of a representative island-style FPGA switchbox. The circles overlaying the array of wires each represent an intersection at which pass transistors have been arranged as shown to the right. Turning on any one of the transistors creates a connection from one side of the switchbox to another. If multiple transistors are turned on a signal may fanout to more than two sides of the switchbox.	6
Figure 5: Example HSRA architecture. Squares indicate LUTs. Xs show connectivity between the lowest level of switchboxes and LUT inputs. Here the base channel width is three, the growth rate is 0.5, and there are four levels of interconnect. The growth rate is achieved by alternating compressing switchboxes (ovals) and non-compressing switchboxes (triangles) [Sharma05].	7
Figure 6: RaPiD-Benchmark cell. Functional units are shown on top. The routing tracks below are divided between short busses and long busses. Long busses may be joined together via bus connectors, shown as small squares [Ebeling99].	8
Figure 7: Pseudo-code for the simulated annealing algorithm. After creating an initial placement randomly and computing its cost, the quality of placement is gradually adjusted by repeatedly swapping two blocks. Initially almost any swap is permitted, but as the temperature is lowered fewer and fewer moves that increase the cost are accepted.	10
Figure 8: Example decomposition of FPGA logic and interconnect into a routing graph. The top diagram shows a Triptych routing and logic block (RLB). Below is the routing graph representation of the RLB. Nodes 1, 2, and 3 represent the three input wires to the RLB. Node 4 represents the input terminals to the LUT itself; this node thus has a capacity of three, corresponding to the LUT having	

three inputs. Since the input nets terminate at the LUT, node 4 is termed a sink node. Nodes five and six are termed source nodes, representing the outputs of the LUT and D flip-flop selectable by the mux. Node 7 represents the output of the mux, and nodes 8-10 represent the three RLB outputs. Because RLB inputs may be routed directly through to RLB outputs, edges connect nodes 1-3 to nodes 8-10..... 13

Figure 9: Independence pseudo-code based upon the generic simulated annealing pseudo-code of Figure 7. The highlighted portions indicate steps Independence performs to maintain a fully routed netlist used to compute cost, and point out the dependence of the cost function on the routing results..... 16

Figure 10: Triptych RLB. Two inputs receive data from one of two neighboring RLBs. The middle input receives data either fed back from the RLB's D flip-flop or from one of seven routing tracks in a vertical routing channel. Any input may be routed to any output, as may the output of the three-input LUT or the D latch [Borellio95]..... 20

Figure 11: Triptych diagonal interconnect. In (a), data flow is shown going to the right between adjacent RLBs. (b) shows how left-going cells, shown in gray, overlap the right-going cells in a full Triptych array. (c) shows how each RLB diagonal may be configured to receive data from a neighboring right-going or left-going cell. (d) shows the complete diagonal interconnect. 21

Figure 12: Triptych vertical channel representation. There are two staggered segmented routing tracks in each of three lengths: 8, 16, and 32 RLBs. A seventh routing track is unsegmented and dedicated for top/bottom chip input pins [Borellio95]. 22

Figure 13: Vertical channel connectivity to RLBs. Right-going vertical channels connect right-going cells in adjacent rows (left). Like the RLBs themselves, the vertical channel structure is reversed and overlain to integrate with the checkerboard of RLBs, as shown on the right. 22

Figure 14: Side IO and boundary connections for Triptych RLB diagonals. Side IOs are bidirectional, able to act as output pins by receiving RLB outputs and as input pins driving RLB inputs. RLBs on the boundary have their output diagonals move laterally rather than diagonally, enabling all RLBs in the array to receive data from one of two neighbors. A single vertical channel is provided between IOs and the outer RLB columns to loop data back into the array, as shown in gray. 23

Figure 15: 4-input RLB structure. The extra input is accepted from the vertical channels. Multiplexing is provided to select which three of the four inputs will be used in the logic function. Routing paths are provided for the diagonals to all four outputs and for the two middle inputs to one diagonal and one middle output each. [Hauck 95a]. 24

Figure 16: Cells A (left-going) and B (right-going) are the only two cells driven by the output diagonals of the cells directly above and below them. These wires can thus be easily checked for conditions that result in the inputs of A and B being unroutable. 28

Figure 17: Determination of best pegs per cell threshold for the peg cost term in the Triptych placer cost function. The y axis shows the sum of the number of RLB columns needed to route the netlist, while the x axis shows the pegs per cell threshold. Values of 0.8 and 0.9 produced the best results.....	30
Figure 18: Determination of weighting parameter B. Within the range 0.05-0.2 the total number of columns remains very similar; the best choice for B found proved to be the nominal value of 0.1, corresponding to roughly equal weighting with wirelength.	31
Figure 19: Determination of weighting parameter C. Note that for C=0 that many of the netlists are not routeable. The value of C = 1.125 proved to be the best among those tried.....	32
Figure 20: Graph of minimum number of columns required to route benchmarks vs. congestion weighting parameter λ . The upper line indicates results for 3-input RLB architectures, while the lower line indicates results for 4-input RLB architectures.	36
Figure 21: An example of a local routing violation produced by an Independence placement. In this case, net <code>_90_gat_22_</code> and net <code>_423gat_155</code> are both trying to use the empty cell's upper diagonal. Because cell <code>423gat_155_</code> requires inputs to all three pins, and none of these inputs is the net being routed to output <code>out:_423gat_155</code> , it can be determined purely from this context that the netlist is not routeable.....	41
Figure 22: A second example of a local routing violation produced by Independence. In this case, six different nets need to be routed to the RLB pair [210] and [243]. This means that all four adjacent RLBs who have diagonals connecting to this pair must be used to route the six required nets. Because cell [254] has three input nets and an output net that are not required by [210] or [243], its upper diagonal cannot be used in routing and the netlist is therefore guaranteed to be unrouteable.....	42

LIST OF TABLES

Table Number	Page
Table 1: VPR Cooling Schedule Temperature Scaling Factors [Betz97].....	11
Table 2: Comparison of RLB count between [Ebeling95] synthesis and synthesis performed for this work.	33
Table 3: Performance of Triptych Placer with 4-input RLBs on netlists with 150-300 logic blocks and less than 128 I/Os.....	34
Table 4: Benchmark results for Triptych placer vs. Independence, 3-input RLBs. Netlists that failed to route were assigned a value of 17 for computing the sum.	37
Table 5: Benchmark results for Triptych placer vs. Independence, 4-input RLBs.....	38
Table 6: Independence placement cost calculations for selected benchmark test cases..	39
Table 7 Triptych placement cost calculations for selected benchmark test cases	40
Table 8: Minimum number of columns as a function of peg threshold. Netlists that failed to route were assigned a value of 17 for computing the sum.	47
Table 9: Minimum number of columns as a function of peg cost weighting parameter B. Netlists that failed to route were assigned a value of 17 for computing the sum.....	47
Table 10: Minimum number of columns as a function of routeability cost weighting parameter C. Netlists that failed to route were assigned a value of 17 for computing the sum.....	48
Table 11: Minimum number of columns as a function of congestion weighting factor, 3-input RLBs. Netlists that failed to route were assigned a value of 17 for computing the sum.....	48
Table 12: Minimum number of columns as a function of congestion weighting factor, 4-input RLBs. Netlists that failed to route were assigned a value of 17 for computing the sum.....	49

ACKNOWLEDGEMENTS

This work was supported by a grant from the National Science Foundation. The author also wishes to thank Ken Eguro for offering his experience and advice when needed, and Scott Hauck for his leadership and guidance throughout the project.

1 Introduction

Digital systems designers have a wide variety of hardware to select from to perform their required tasks. In many cases, a general-purpose microcontroller or microprocessor is an attractive option; implementing the system may simply require writing software. In others, the performance advantage of an application-specific integrated circuit (ASIC) may justify its lengthy design time and high cost. In still other cases, Field Programmable Gate Arrays (FPGAs) offer the right mix of performance and cost.

As the name implies, FPGAs are integrated circuits composed of digital logic resources and an interconnect structure linking those logic resources together. The logic function computed by the array is user-programmable with the help of software tools. The amount and type of logic resources, as well as the size and composition of the interconnect structure, vary widely between different FPGA architectures. Some FPGAs have a fine-grained architecture, comprised of a large number of relatively small logic resources. Others are coarse-grained, containing fewer, larger logic units. Still others take a hybrid approach, offering both coarse-grained and fine-grained structures on the same chip. For example, the Xilinx Virtex-4 family of commercial FPGAs boasts, among other features, up to two on-chip PowerPC CPU cores and up to 200,000 configurable logic blocks [Xilinx06]. Indeed, there is sufficient variety in size, composition, and cost of FPGAs that one must determine whether a given FPGA is a good fit for the target application.

Implementing a circuit on an FPGA follows a somewhat analogous design flow to ASIC design. Circuits are designed and verified using a hardware description language (HDL) and simulation software. Synthesis software performs logic optimization on the HDL representation and then maps the optimized circuit into blocks corresponding to the available logic resources on the FPGA. The result of this mapping is a netlist. A placement tool must then choose how to physically arrange the netlist on the FPGA. A routing tool determines what routing resources are used to connect the blocks and produces the data required to program the FPGA with the desired circuit. Finally,

programming software configures the FPGA's logic blocks and interconnect structure with the data supplied by the routing tool.

Because software tools are used in nearly every phase of the design process, the quality of these tools becomes an important factor in the quality of the final circuit. A poor synthesis tool may fail to optimize the circuit sufficiently to allow it to fit within the limited hardware resources of the target FPGA. A poor placement tool might produce a netlist arrangement that cannot be routed given the routing resources available. A poor routing tool might not find a routing solution that meets circuit delay requirements even though one exists.

In addition to tool quality, tool portability is an important issue. Many FPGA software tools target a specific architecture. As new architectures are developed, new tools must be developed as well, whereas it would be preferable to reuse existing tools if possible. Previous work to address this issue in FPGA placement led to the development of Independence [Sharma05], an FPGA placement tool that is portable across a wide range of FPGA architectures. Instead of targeting specific architectures, Independence adapts to an architecture specified by the user by performing placement in a highly routing-aware fashion. Independence's adaptability has been demonstrated on three different FPGA architectures by comparing its performance to placement tools targeting those architectures. The placement tools that Independence has been benchmarked against to date have had relatively limited routing awareness [Sharma05]. This paper evaluates Independence on a fourth architecture, Triptych, by comparing its performance to a routing-aware custom placement tool targeting that architecture.

Further background on the variety exhibited in FPGA architectures and placement software is provided the next section. Section three presents Independence and comments briefly on the results obtained to date. Section four presents the Triptych architecture and the placement tool developed for it. Finally, section five shows the performance of Independence versus the Triptych placer, and discusses the results.

2 FPGA Background

A brief overview of different FPGA architectures is presented next. This is followed by background on FPGA place-and-route software, with particular attention to algorithms used in Independence and the Triptych placer.

2.1 FPGA Architectures

FPGAs come in a wide range of sizes and architectures, but all of them contain some repeated arrangement of logic resources and routing resources. The configuration of the resources is typically programmed into small, local SRAMs. The SRAMs are used to store logic functions by implementing them as lookup tables. They may also store configuration options by driving multiplexer select bits. These options might include whether a D Flip flop is used in a given subcircuit, which routing channels drive a given logic resource, or any other configurable aspect of the FPGA. To better illustrate how this works, a popular class of FPGA architectures, island-style, is examined in some detail. Other architectures are then presented briefly to give some idea of the variety of FPGAs available today.

2.1.1 Island-Style FPGAs.

Island-style architectures are the most prevalent class of commercial FPGA. The architecture derives its name from the notion that its logic resources, referred to as Configurable Logic Blocks (CLBs), are islands in a sea of routing resources. The CLBs are arranged in a regular array structure and surrounded by both vertical and horizontal routing channels. At the intersections of the vertical and horizontal channels, switchboxes offer the option for data to change direction [Sharma01]. This structure is illustrated in Figure 1.

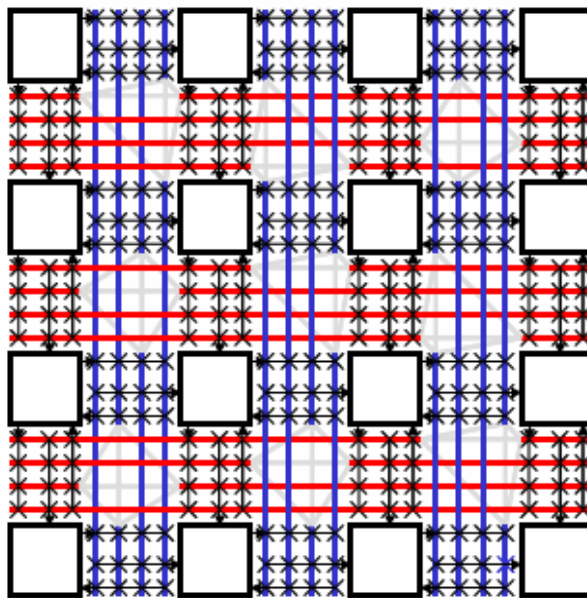


Figure 1: Representation of an island-style FPGA [Sharma05]. The white boxes denote CLBs. Vertical and horizontal lines show routing channels. The Xs indicate points within a switchbox where data can change direction.

Each CLB is a self-contained configurable device. In a simple architecture, the CLB could be nothing more than a lookup table (LUT) plus a flip-flop and a multiplexer, as in Figure 2. The lookup table may then be programmed to compute an arbitrary logic function of n inputs. A popular choice for n historically has been four.

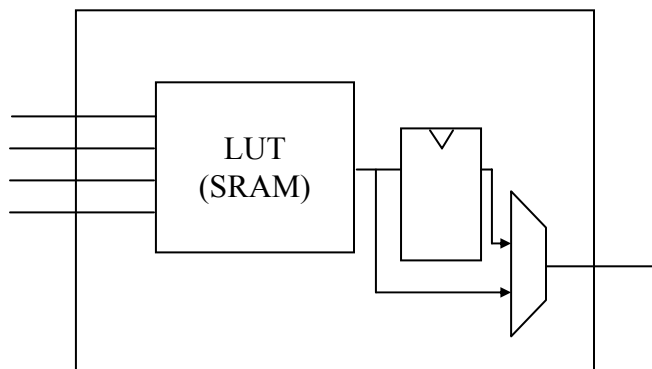


Figure 2: Block diagram of a simple island-style CLB. This CLB contains a four-input LUT whose output is connected to a D flip-flop and a multiplexer input. The flip-flop output is also connected to the mux so that the CLB output can be configured to be either combinational or sequential. Wires used to program the CLB are not shown.

As island-style architectures have grown in complexity, the CLB's complexity has often grown as well. Support for larger logic functions might be added. Special features that accelerate popular logic functions, such as paths intended specifically to chain together CLBs for fast addition, might also be included. Figure 3 shows a hypothetical example of what a more advanced CLB might contain. This style of CLB, containing multiple LUTs and programmable connectivity between them, is similar to that of the Xilinx Spartan II family [Xilinx04]. In addition to increasing size, CLB flexibility also often increases as well.

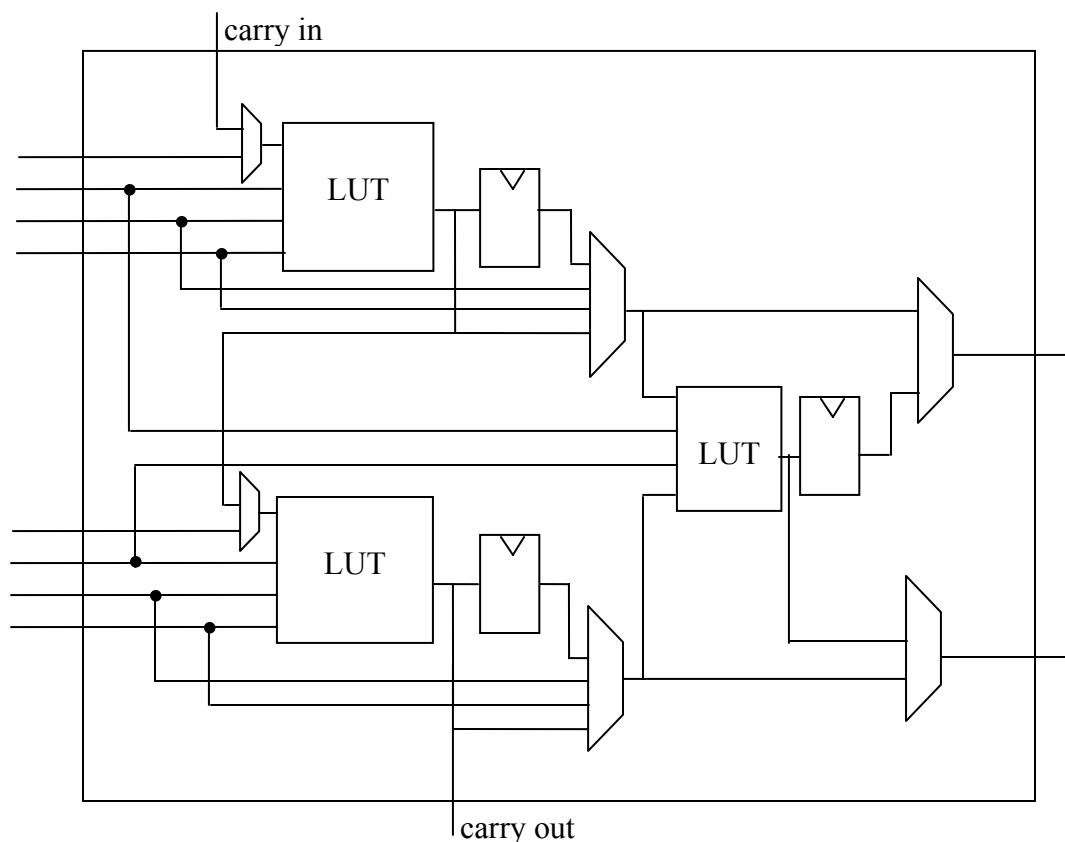


Figure 3: A more advanced hypothetical CLB. The overall CLB accepts eight arbitrary inputs and produces two outputs. One additional input (top) and one additional output (bottom) are shown to represent dedicated carry-in/carry-out lines to be used when the CLB is configured for fast addition. The two leftmost LUTs can be configured to be used separately to compute two four-input functions or combined with the third LUT on the right to compute one eight-input function. Optional flip-flop storage is provided for each.

While the vertical and horizontal routing channels surrounding the CLBs are simply wires, the switchboxes placed at their intersections must contain logic and connectivity to configure how intersecting wires may connect, or if they connect at all. This is typically implemented by linking intersecting wires via a set of six pass transistors, as shown in Figure 4. The gates of the pass transistors may then be programmed on to enable a connection between two wires or off to disable it. In order to save area, not all intersection points are populated with pass transistors, meaning not all routing channels may be connected. In general, however, there is at least one path from a routing channel on one side of the switch box to a channel on each of the three other sides of the switchbox [Chang96].

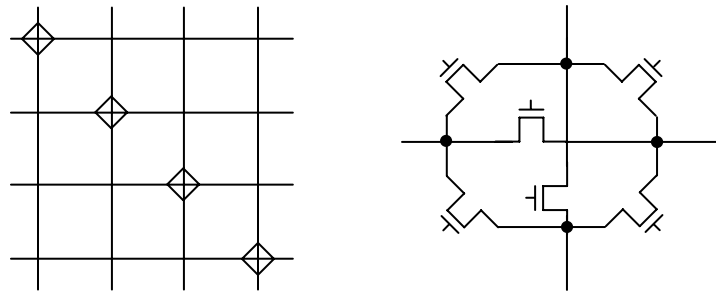


Figure 4: Close-up look of a representative island-style FPGA switchbox. The circles overlaying the array of wires each represent an intersection at which pass transistors have been arranged as shown to the right. Turning on any one of the transistors creates a connection from one side of the switchbox to another. If multiple transistors are turned on a signal may fanout to more than two sides of the switchbox.

2.1.2 HSRA

The Hierarchical Synchronous Reconfigurable Array (HSRA) architecture [DeHon99] differs from island-style architectures by organizing routing resources into a tree-based structure shown in Figure 5. Logic resources consisting of a four-input lookup-table (LUT) and associated D flip-flop are positioned at the leaves of the tree, while central routing channels comprise the root of the tree. Switchboxes provide intersections between different levels of hierarchy. The number of total routing channels available increases going from the root to the leaves according to a specified growth rate. [DeHon99].

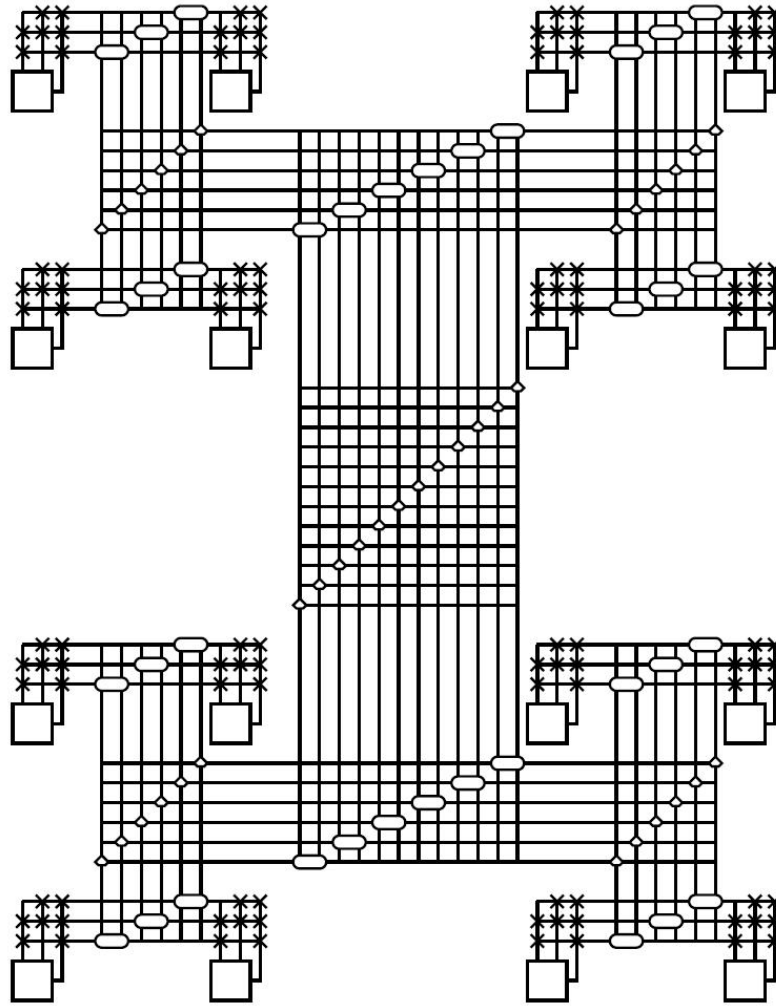


Figure 5: Example HSRA architecture. Squares indicate LUTs. Xs show connectivity between the lowest level of switchboxes and LUT inputs. Here the base channel width is three, the growth rate is 0.5, and there are four levels of interconnect. The growth rate is achieved by alternating compressing switchboxes (ovals) and non-compressing switchboxes (triangles) [Sharma05].

2.1.3 RaPiD

Whereas the island-style and HSRA architectures described previously constitute fine-grained architectures, RaPiD is coarse-grained [Ebeling99]. Figure 6 shows a RaPiD-Benchmark cell, a representative 16-bit implementation targeting digital signal processing applications. It contains a variety of high-level functional units: three ALUs, a multiplier, three 64-word RAMs, and six general-purpose registers (GPRs). A total of

14 routing tracks provide interconnect both within the cell and between adjacent cells [Ebeling99].

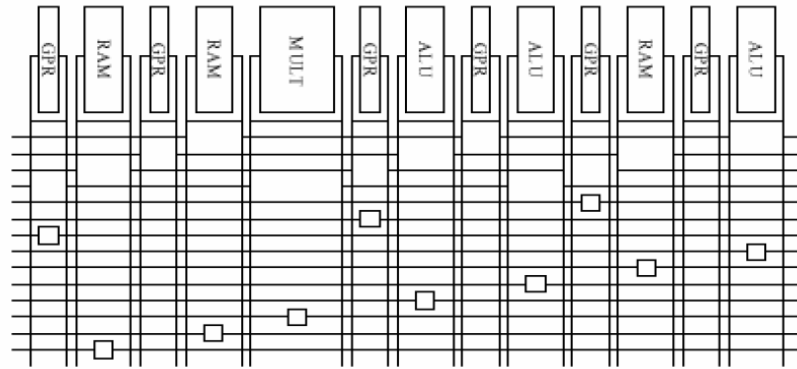


Figure 6: RaPiD-Benchmark cell. Functional units are shown on top. The routing tracks below are divided between short busses and long busses. Long busses may be joined together via bus connectors, shown as small squares [Ebeling99].

2.2 FPGA Place-and-Route Software

Placement and routing are by their nature closely related problems. A placement that yields superior routing (fewer routing resources used, smaller delay) is also a superior placement. Because the routing resources of an FPGA are fixed, FPGA placement can also involve evaluating whether a placement can be routed at all. This makes routing considerations even more important in FPGA placement than otherwise.

2.2.1 FPGA Placement

Because routing is a computationally intensive process, many place-and-route tools use heuristic estimates to evaluate the quality of the routing yielded by a given placement. This takes the form of a cost function that includes parameters for the amount of routing resources used, delay, etc. A lower-cost solution is deemed a superior solution: For example, the leading-edge placer Versatile Place-and-Route (VPR) [Betz97], uses the cost function

$$Cost = \sum_{n=1}^{N_{nets}} q(n) \left[\frac{bb_x(n)}{C_{av,x}(n)} + \frac{bb_y(n)}{C_{av,y}(n)} \right]$$

For each net in the netlist, a two-dimensional bounding box (bb_x , bb_y) for that net's terminals is computed as an estimate of the wirelength (i.e. aggregate routing resources used) required to route the net. $C_{av,x}$ and $C_{av,y}$ are the average routing channel capacities within the computed bounding box of the given net. Scaling the bounding boxes by these values penalizes a net if it is routed in an area of the FPGA with fewer resources than elsewhere. Doing so represents an attempt to reduce congestion, defined as the attempt of multiple nets to use the same routing channel. $q(n)$ is a parameter that increases gradually as the fanout of the net increases to compensate for the bounding box underestimating wirelength for nets with more than three terminals [Betz97]. Taken as a whole, the cost function attempts to evaluate routeability in terms of a combination of wirelength and congestion.

For even a small circuit placed to a small FPGA, the number of possible placements is sufficiently large that the entire solution space cannot be exhaustively searched to find the placement with the lowest cost. On the other hand, because all nets compete for the same routing resources, placing blocks incrementally is likely to produce poor solutions by failing to anticipate future congestion and/or wirelength requirement imposed by the placement of a given block. Algorithms that are able to efficiently search the solution space of complete placements for low-cost results have thus provided the best balance of runtime and performance to date.

The leader among such algorithms is simulated annealing, summarized in the pseudo-code in Figure 7. First, an initial placement is generated randomly, and the cost of the random placement is computed. Based on the initial cost, a starting temperature is calculated. A large number of random swaps of pairs of blocks in the netlist are performed, and the change in cost arising from each move is computed. If the change is positive (i.e. the move degrades the quality of the placement), then the move is randomly

accepted or rejected. The probability of acceptance is controlled by the temperature, and is initially almost one. If the change in cost is negative (i.e. the move improves the quality of the placement) then the move is always accepted. After all swaps are completed, the temperature is reduced according to formulas that collectively are referred to as the cooling schedule, and another series of swaps is performed with the new temperature controlling the probability that a bad move is accepted. When the total fraction of moves accepted dips below some threshold, or a maximum number of temperature reductions is reached, the algorithm terminates.

```

create_random_placement();
cost = compute_total_cost();
temp = compute_start_temp();
while ((frac_accepted > threshold) || (num_iterations < max) {
    for (I = 1; I < movesPerIter; I++) {
        blocksSwapped = swap_two_blocks();
        deltaC = compute_change_in_cost(blocksSwapped);
        if (deltaC > 0) { //it's a bad move
            If (accept_bad(temp) == false) {
                unswap_two_blocks(blocksSwapped);
            }
            else {
                //accept move
                cost += deltaC;
            }
        }
        else { //it's a good move
            cost += deltaC;
        }
    }
    temp = compute_new_temp();
}

```

Figure 7: Pseudo-code for the simulated annealing algorithm. After creating an initial placement randomly and computing its cost, the quality of placement is gradually adjusted by repeatedly swapping two blocks. Initially almost any swap is permitted, but as the temperature is lowered fewer and fewer moves that increase the cost are accepted.

The algorithm's effectiveness stems from several key properties. Simulated annealing evaluates full placement solutions rather than placing blocks one-by-one. This prevents the quality of the solution from depending upon the order in which blocks are placed.

Randomly accepting bad moves help ensure that a broader portion of the solution space is searched by discouraging convergence toward local minima. Trying a large number of moves at each iteration helps search the solution space thoroughly, but can be tuned to provide a good balance between performance and runtime.

VPR incorporates a number of optimizations demonstrated in previous work as well as some of its own. Other placement tools implementing simulated annealing since have followed suit, including Independence and the Triptych placer implemented here. Those common to all three include the following:

- **Initial Temperature:** VPR computes the initial temperature by performing N random moves, where N is the number of logic blocks plus IOs in the circuit, and then sets the initial temperature equal to 20 times the standard deviation of the change in cost of these moves.
- **Cooling Schedule:** VPR computes the next temperature by multiplying the current temperature by a scaling factor chosen according to the fraction of moves accepted at the current temperature. The value chosen is specified in Table 1.

Table 1: VPR Cooling Schedule Temperature Scaling Factors [Betz97]

Fraction Accepted	Temperature Scaling Factor
Less than 0.15	0.8
0.15-0.8	0.95
0.8-0.96	0.9
Greater than 0.96	0.5

- **Distance Limit:** It was shown in [Lam88,Swartz90] that it is desirable to maintain the fraction of moves accepted at 0.44 for as long as possible. VPR achieves this by gradually reducing the range of moves that can be made when the fraction of moves accepted drops below 0.44. Initially, any two blocks can be swapped. At each temperature reduction, the maximum distance between blocks that can be swapped is recalculated according to

$$dlimit_{new} = dlimit_{old} * (1 - 0.44 + frac_accepted) \text{ [Betz97]}$$

Assuming that the logic blocks are arranged in rectangular arrays (valid for most architectures), it can be measured as a vector sum of distances in logic blocks in the x and y directions. Accordingly its initial and maximum value is the maximum distance between IOs in opposite corners of the array, and its minimum value is 1. A check is thus included after computing the new distance limit to keep the value within its legal range [Betz97].

2.2.2 FPGA Routing

As in placement, routing so far has proven a problem best solved with an iterative approach. The most effective algorithm to date, and hence the most widely used, is Pathfinder [McMurchie95]. Pathfinder represents an FPGA's routing resources as a directed graph. This routing graph is formed by decomposing the architecture into nodes representing routing resources, and edges representing different ways to connect routing resources together. Edges could thus indicate switchboxes that connect two different routing channels, logic elements that receive one routing channel as input and one as output, or any other configurable circuit element that controls the FPGA's connectivity. Figure 8 shows an example decomposition of an FPGA logic block and its circuit elements into a routing graph.

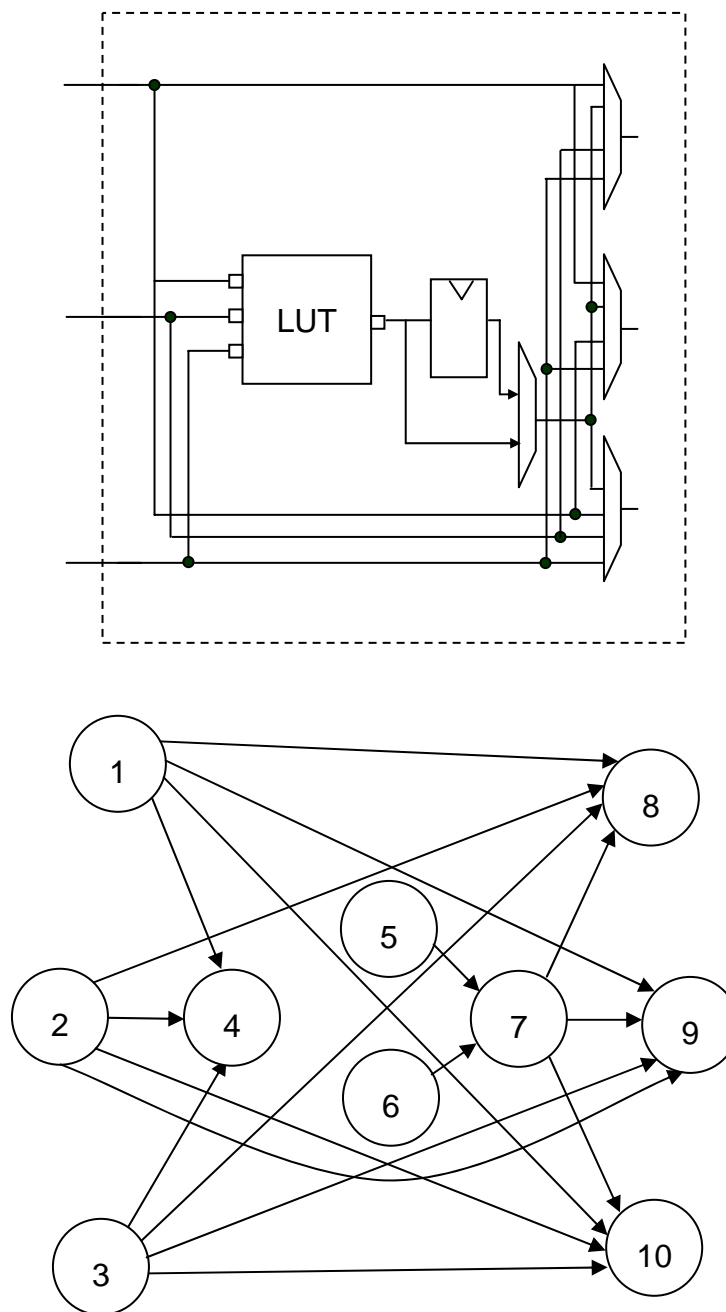


Figure 8: Example decomposition of FPGA logic and interconnect into a routing graph. The top diagram shows a Triptych routing and logic block (RLB). Below is the routing graph representation of the RLB. Nodes 1, 2, and 3 represent the three input wires to the RLB. Node 4 represents the input terminals to the LUT itself; this node thus has a capacity of three, corresponding to the LUT having three inputs. Since the input nets terminate at the LUT, node 4 is termed a sink node. Nodes five and six are termed source nodes, representing the outputs of the LUT and D flip-flop selectable by the mux. Node 7 represents the output of the mux, and nodes 8-10 represent the three RLB outputs. Because RLB inputs may be routed directly through to RLB outputs, edges connect nodes 1-3 to nodes 8-10.

Each node in the routing graph has an associated cost and a capacity. The cost value is arbitrary, provided that the relative costs of nodes are meaningful. The capacity defines how many nets may be routed through a given node. In a typical representation of a routing fabric the capacity of each node is one, but this is not strictly necessary.

An initial routing is computed for each net by traversing the routing graph using Dijkstra's algorithm. During this initial routing, previous nets are ignored when routing subsequent nets, allowing all nets to take their own preferred path and permitting any resulting congestion. Based on this result, the cost of each node is increased according to its congestion, using the equation

$$c_n = (b_n + h_n) * p_n$$

where b_n is the base cost of the node, p_n is a factor proportional to the number of nets sharing the node, and h_n is determined by the history of congestion on that node across all previous routing iterations. All nets are rerouted using Dijkstra's algorithm with the updated node costs, and the process is repeated until a maximum number of iterations is reached or the routing completes with zero congestion.

Increasing the cost of a node in a manner proportional to the number of nets attempting to use it results in popular resources becoming more expensive. As this change takes place, it prevents nets from searching for paths with less wirelength but greater congestion by assigning such paths a higher cost. On the other hand, if a net's preferred path does not increase in cost, there is no mechanism restricting it from choosing the same route on successive iterations. This ensures that nets benefiting the most from using a particular routing resource are given priority to use it. The history cost term discourages nets from returning to previous areas of high congestion by ensuring that routing resources that became expensive in a previous routing iteration remain at least somewhat expensive in subsequent routing iterations.

3 Independence

Independence [Sharma05] is an FPGA place-and-route tool that is architecture adaptive. Instead of tuning its approach to place-and-route to a particular architecture, Independence uses cost functions for placement that are directed by user-specified architecture data. This is explained in greater detail below.

3.1 Independence Placer

Because it has been demonstrated to one of the best overall algorithms for placement, Independence uses the simulated annealing algorithm, borrowing many features from VPR that have made the latter a leading-edge placement tool. The key difference between Independence and other placement tools based on annealing is that it performs routing during placement to direct its cost function.

The pseudo-code shown below in Figure 9 points out the key distinctions in Independence's implementation of simulated annealing.

```

create_random_placement();
routes = route_netlist();
cost = compute_total_cost(routes);
temp = compute_start_temp();
while ((frac_accepted > threshold) || (num_iterations < max) {
    for (I = 1; I < movesPerIter; I++) {
        blocksSwapped = swap_two_blocks();
        routes = rip_up_and_reroute(blocksSwapped);
        deltaC = compute_change_in_cost(blocksSwapped, routes);
        if (deltaC > 0) { //it's a bad move
            If (accept_bad(temp) == false) {
                unswap_two_blocks(blocksSwapped);
                routes = restore_old_routes(blocksSwapped);
            }
            else {
                //accept move
                cost += deltaC;
            }
        }
        else { //it's a good move
            cost += deltaC;
        }
    }
    temp = compute_new_temp();
    update_history_costs();
    routes = route_netlist();
}

```

Figure 9: Independence pseudo-code based upon the generic simulated annealing pseudo-code of Figure 7. The highlighted portions indicate steps Independence performs to maintain a fully routed netlist used to compute cost, and point out the dependence of the cost function on the routing results.

Like VPR, Independence is driven by a cost function that seeks to reduce wirelength and congestion. Instead of using a generic heuristic estimate, Independence infers these parameters from the target architecture's routing graph. Using the information supplied by the routing graph, Independence calculates cost by fully routing the circuit and using its routed solution to sum the wirelength and congestion for the entire circuit. To keep the amount of computation required manageable, complete routing and cost calculations are performed only after initial random placement and after each reduction in temperature. The cost is updated incrementally after each placement, according to the equation

$$\Delta C = \Delta \text{wireCost} / \text{prevWireCost} + \lambda * \Delta \text{congestionCost} / \text{congestionNorm}$$

$\Delta \text{wireCost}$ is the change in wire cost associated with the current move. prevWireCost is the total wire cost associated with the placement prior to the move. $\Delta \text{congestionCost}$ is the change in congestion associated with the current move. congestionNorm is a normalization factor for the congestion cost [Sharma05]. The total congestion cost is computed as

$$\text{congestionCost} = \sum_i^{\text{allNodes}} \max((\text{occupancy}_i - \text{capacity}_i), 0)$$

occupancy is the number of nets currently using a given node in the routing graph, and capacity is the number of nets allowed to use it. Congestion for underutilized or unused nodes is defined to be zero. Since congestionCost should converge to zero as the placement becomes fully routable, a previous congestion cost cannot be used as a normalization term; currently congestionNorm is set equal to prevWireCost . The parameter λ weights the importance of reducing wirelength against the importance of reducing congestion. Experiments have shown that the best value for λ varies from architecture to architecture [Sharma05].

$\Delta \text{wireCost}$ and $\Delta \text{congestionCost}$ are calculated by ripping up all routes of nets affected by the move and rerouting them based on the new locations of their terminals. In the case of multi-terminal nets, if the net is an input only the branches of the route affected by the move are ripped up, while for outputs the entire net is rerouted [Sharma05]. After completing this process the cost of the new routes can be calculated and subtracted from the cost of the routes that were ripped up to obtain the change in cost for the move.

The routing algorithm used during this process is based on Pathfinder. However, an adjustment is made to the history cost calculation to account for the fact that the history

of congestion on a node loses relevance as the placement changes over time [Sharma05]. This is handled in two ways. First, history costs are adjusted only when the temperature is reduced, not after each placement move. Second, the history cost of a node is allowed to decay over multiple routing iterations if it has no congestion. The exact equation is

if (shared)

$$historyCost_i = \alpha * historyCost_{i-1} + \beta$$

else

$$historyCost_i = \alpha * historyCost_{i-1}$$

Currently $\alpha = 0.9$ and $\beta = 0.5$ [Sharma05].

3.2 Previous Benchmark Results

Independence has previously been tested on island-style, HSRA, and RaPiD architectures against placers targeting these architectures. In each case Independence's performance was comparable or better to that of the custom placer [Sharma05]

4 Triptych Architecture

The Triptych architecture [Borellio95] provides an excellent contrast to island-style, HSRA, and RaPiD. Like island-style and HSRA, it is fine-grained. However, it differs from all three architectures in two major ways: its routing resources are more asymmetric, and some of them are integrated with the logic resources. The overall Triptych architecture is described in greater detail next.

4.1 Triptych Logic Resources

A Triptych Routing and Logic Block (RLB) contains a three input LUT to perform arbitrary logic functions of three or fewer inputs. The LUT's output is connected to a D flip-flop to provide data storage. A multiplexer configures whether the flip-flop is actually used.

A local routing fabric surrounds the LUT and latch and provides the interface between adjacent RLBs and shared vertical routing channels. Each RLB has two inputs that may each come from one of two adjacent RLBs, and one that may come from either the vertical channels or the output of the D-latch. Similarly, two of the three RLB outputs are connected to two adjacent RLBs each, and the third output connects to vertical routing channels. Each of the three outputs may be driven by any one of four nets--one of the three inputs or the output of the mux mentioned above. This structure is summarized in Figure 10 [Borellio95].

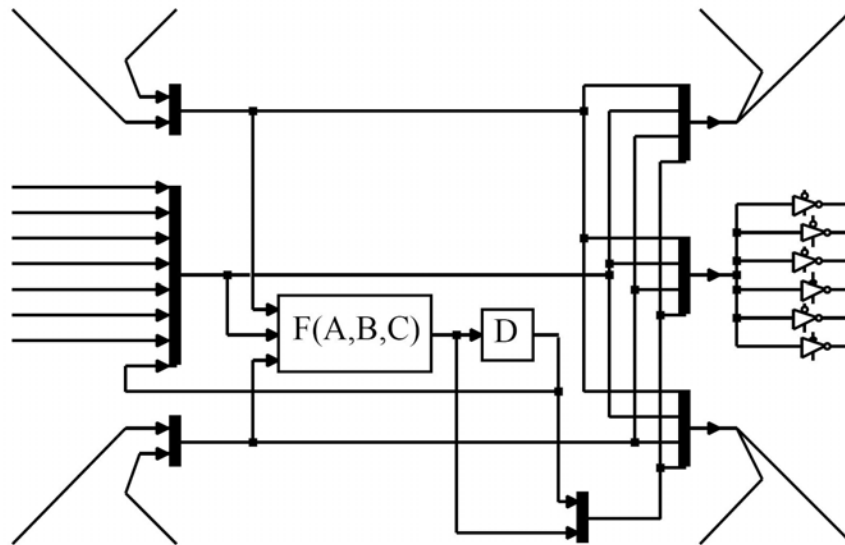


Figure 10: Triptych RLB. Two inputs receive data from one of two neighboring RLBs. The middle input receives data either fed back from the RLB's D flip-flop or from one of seven routing tracks in a vertical routing channel. Any input may be routed to any output, as may the output of the three-input LUT or the D latch [Borellio95].

4.2 Triptych Interconnect

Outside the RLBs, Triptych's routing fabric is divided between two components--diagonal wires connecting adjacent RLBs, and segmented vertical channels connecting RLBs at greater distances from one another. Figure 11(a) shows how five neighboring RLBs provide data flow diagonally and to the right through connections between diagonal inputs and outputs. To provide data flow to the left, another set of RLBs is interwoven with the right-going RLBs as shown in Figure 11 (b). The resulting RLB structure is a checkerboard-patterned array of right-going and left-going cells. Wires connecting outputs to cells directly above and directly below them provides data flow between right- and left-going cells. This option is configured via a mux as shown in Figure 11 (c), and the overall diagonal interconnect is shown in Figure 11 (d). At the top and bottom of the array, connections that would be made to RLBs above and below, respectively, are made to cells to the left and right instead, as shown in Figure 14.

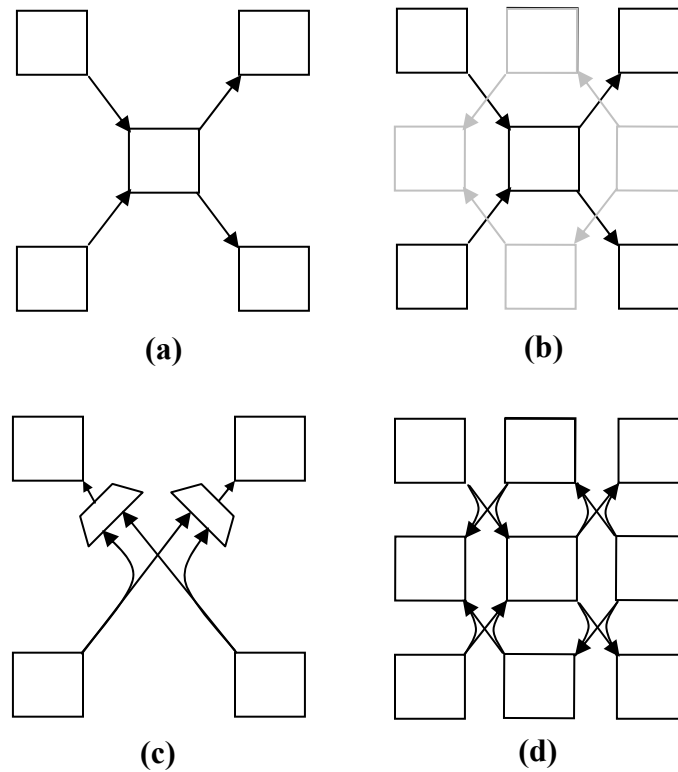


Figure 11: Triptych diagonal interconnect. In (a), data flow is shown going to the right between adjacent RLBs. (b) shows how left-going cells, shown in gray, overlap the right-going cells in a full Triptych array. (c) shows how each RLB diagonal may be configured to receive data from a neighboring right-going or left-going cell. (d) shows the complete diagonal interconnect.

In between each column of RLBs are two vertical channels--one providing data flow to the right, and one providing data flow to the left. The channels are identical structures each containing seven routing tracks. One track is dedicated to an input pin. The other six facilitate longer-range routing between RLBs. Two tracks are divided into 8 RLB-high segments, two into 16-RLB segments, and two into 32-RLB segments. This structure is shown for a single channel in

Figure 12; the overall structure is shown in Figure 13.

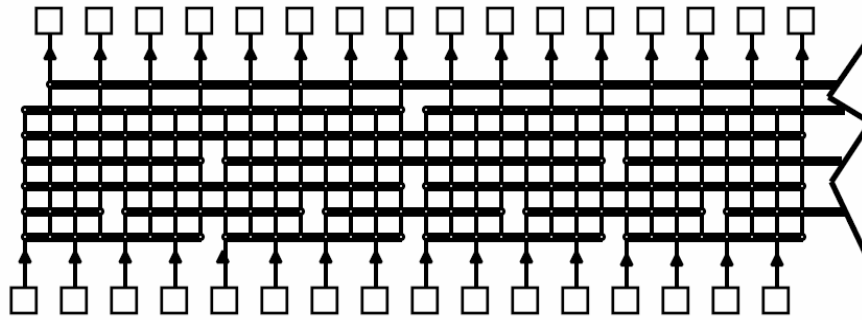


Figure 12: Triptych vertical channel representation. There are two staggered segmented routing tracks in each of three lengths: 8, 16, and 32 RLBs. A seventh routing track is unsegmented and dedicated for top/bottom chip input pins [Borello95].

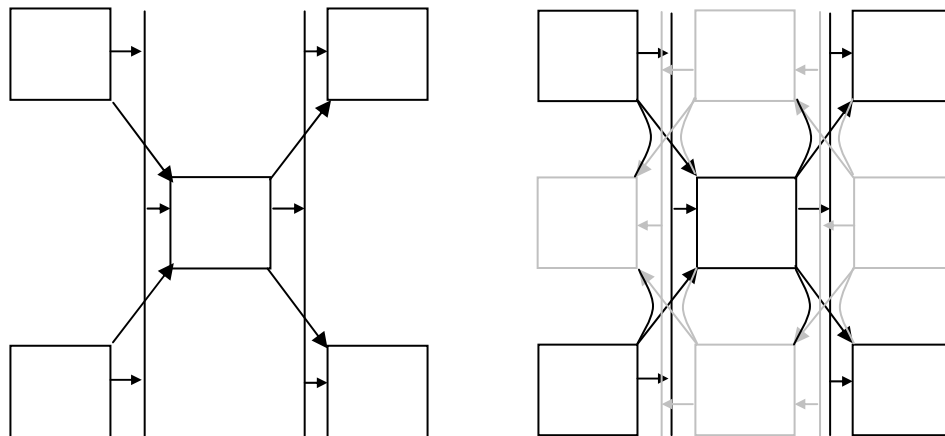


Figure 13: Vertical channel connectivity to RLBs. Right-going vertical channels connect right-going cells in adjacent rows (left). Like the RLBs themselves, the vertical channel structure is reversed and overlain to integrate with the checkerboard of RLBs, as shown on the right.

At the borders of the array, left-going and right-going cells are connected to each other via external vertical channels to allow data to loop back into RLBs moving data in the opposite direction. This is shown in Figure 14.

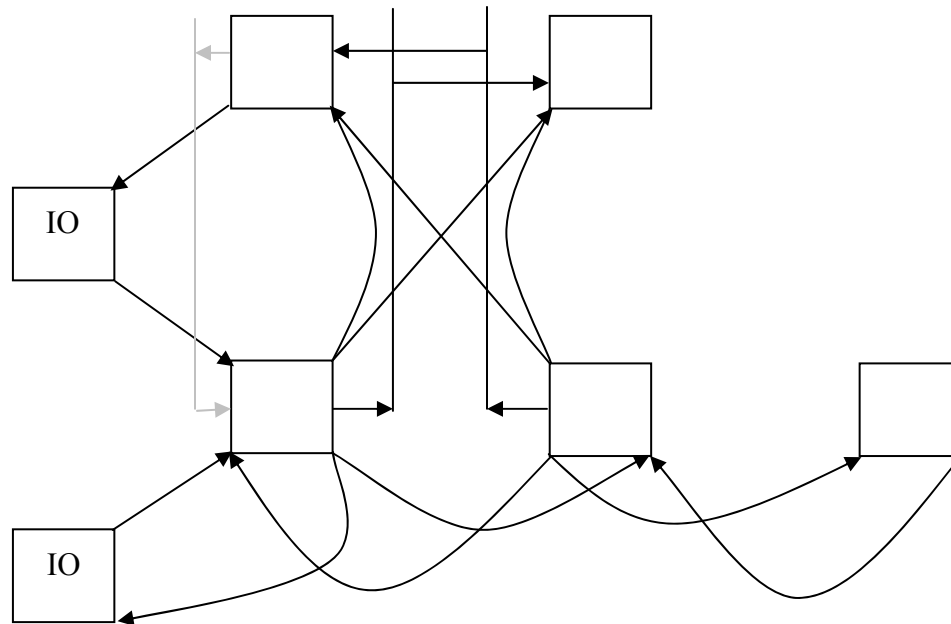


Figure 14: Side IO and boundary connections for Triptych RLB diagonals. Side IOs are bidirectional, able to act as output pins by receiving RLB outputs and as input pins driving RLB inputs. RLBs on the boundary have their output diagonals move laterally rather than diagonally, enabling all RLBs in the array to receive data from one of two neighbors. A single vertical channel is provided between IOs and the outer RLB columns to loop data back into the array, as shown in gray.

Triptych's I/O structure follows naturally from the RLB and interconnect structure. At the top and bottom of the array, input-only pins connect to the vertical channels. On the left side of the array, there are right-going cells with no RLBs to receive inputs from, and left-going cells with no RLBs to connect their outputs to. Instead, these connections are made to I/O pins. Each pin receives one output from a left-going cell and drives an input of one right-going cell except at the corners, where both connections are made to the same RLB. This structure is shown in Figure 14. A completely analogous structure is also formed on the right side of the array.

4.3 Alternate Triptych Architecture Styles

Most FPGA architectures feature four-input LUTs as the smallest logical unit in their array of logic resources. Triptych chooses three-input LUTs instead because the area penalty incurred for mapping a circuit to a larger number of LUTs is made up for by the

improved RLB density that can be achieved by using 3-input LUTs instead of 4-input LUTs [Borellio95].

In an attempt to balance RLB density with the improved routeability that can be achieved with 4-input RLBs, a 4-input RLB with a 3-input LUT was proposed. Figure 15 shows the structure of the 4-input RLB. The input diagonals' connectivity is unchanged from the 3-input RLB architecture except that multiplexing is now required to select whether they are used in computing the logic function. The extra input is obtained from the vertical channels. In order to keep the multiplexing of the outputs at 4:1 and to provide full flexibility for the output diagonals, each of the middle inputs is restricted in choice to one diagonal and one of the middle outputs. Additionally, the feedback path from the D latch to the middle input is eliminated to accommodate selection between four vertical channel tracks for both middle inputs.

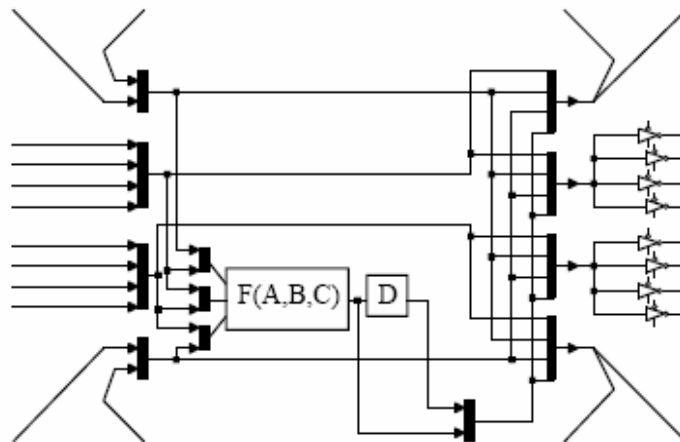


Figure 15: 4-input RLB structure. The extra input is accepted from the vertical channels. Multiplexing is provided to select which three of the four inputs will be used in the logic function. Routing paths are provided for the diagonals to all four outputs and for the two middle inputs to one diagonal and one middle output each. [Hauck 95a].

[Borellio95] does not explicitly discuss the connectivity of the 4-input RLBs to the vertical channels, the number of routing tracks in each channel, or the connectivity (if any) of top and bottom pins to the channels. A reasonable hypothesis based upon the

structure of the RLB itself, and the one used in this work, is that the top and bottom pins are eliminated and that the vertical channel is expanded from seven to eight routing tracks. The logical extension of the six non-pin tracks' segment size suggests giving the additional tracks a segment size of 64 RLBs in length. This was implemented for testing 4-input RLB architectures.

4.4 Triptych Placement Software

The original placement tool for Triptych [Ebeling95] was not available for use in this experiment. For this reason a new placer using the original cost function was written. Writing a new placer had the additional advantage that the implementation of the simulated annealing algorithm used could be kept similar to that of Independence so that the cost function itself was the primary difference between the two tools. Thus, like Independence, the Triptych placer uses a starting temperature calculation, cooling schedule, and distance limit calculation similar to that of VPR (see section 2.2.1).

The cost function developed in [Ebeling95] addresses several issues in wirelength estimation that are peculiar to the Triptych architecture. The asymmetry of Triptych's routing resources in an XY plane means that conventional distance estimations such as Manhattan distance or VPR's bounding box model are not useful for Triptych [Ebeling95]. Placement software for Triptych thus requires a wirelength calculation that accounts for the composition and direction of its interconnect. Additionally, because a portion of the interconnect is embedded within the RLBs, placement quality will be sensitive to how the RLBs were utilized. Congestion thus plays a more crucial role in placement to Triptych than to architectures that do not have integrated logic and routing resources. The overall cost function is

$$TotalCost = A * wireCost + B * pegCost + C * localRouteCost$$

Where A , B , and C are weighting terms, one of which may always be chosen to be one.

WireCost, *pegCost*, and *localRouteCost* are explained each in turn below. [Ebeling95] also provides a term for delay. Because Independence lacks a delay parameter in its cost function, this term was omitted. This allowed for a more direct comparison between the wirelength and congestion factors in each placer's cost function.

wireCost is the sum of the wirelength estimates for each net in the netlist. For single destination nets, the wirelength can be taken as the distance from source to sink. This distance is determined using a lookup table that provides a shortest-path cost to travel from one RLB to another, taking the cost to travel to an adjacent RLB via a diagonal to be one, and taking the cost to travel a distance of 2 RLBs along a vertical channel to be one. This scaling of vertical channel distance was chosen experimentally in [Ebeling95]. It reflects the fact that for nearby connections, it is preferable to use diagonal interconnect, while for distant connections it is somewhat preferable to use the vertical channels.

For multi-destination nets, some metric is needed to combine the distances from the source to each sink into a single cost term. For Triptych, this is

$$\text{wirelength} = 0.9 * (\text{semiperimeter}) + 0.1 * (\text{avg_distance})$$

semiperimeter is a measure of the smallest bounding box needed to enclose the source and all sinks, and thus for Triptych is the sum of the longest distance in each of the four diagonal directions. *avg_distance* indicates the average distance from the source to each of the sinks. Giving this a small weighting in the overall wirelength accounts for the fact that a wirelength estimate with a larger semiperimeter but a lower average distance might yield a better quality placement [Ebeling95].

pegCost is a measure of the density of RLB inputs used for logic functions. It is useful in evaluating the quality of a placement since as more inputs per RLB are used for logic, fewer are available for local routing. Thus, the number of inputs used for logic (pegs) is

counted within every unique 3x3 window of RLBs, including those at chip boundaries for which not all RLBs in the window actually exist. If the number of pegs per RLB in a given window exceeds a certain threshold, dubbed the peg threshold, then the contribution to *pegCost* for that window is the amount above the threshold squared; otherwise the contribution is set to zero. By imposing such a stiff penalty for exceeding a threshold, placement moves that distribute the logic somewhat sparsely are rewarded. Because there are routing resources internal to the RLBs, doing this also tends to improve congestion [Ebeling95].

localRouteCost is a penalty given to nets that can be identified as unrouteable purely from local context. A pair of adjacent RLBs with a left-going cell on the left and a right-going cell on the right share diagonal connections to their inputs coming from the two RLBs directly above them and the two RLBs below them, as shown in Figure 16. By assuming that vertical channels can provide routes if necessary, a check of whether all of the inputs to the pair of RLBs can be simultaneously provided either by the four diagonals or by vertical channels is equivalent to checking whether the placement to the six cells being examined has caused a net to be unrouteable, i.e. the placement has blocked a required route. Similar checks can also be made at the array boundaries by correctly isolating groups of RLB pins and I/Os that share common neighbors [Ebeling95].

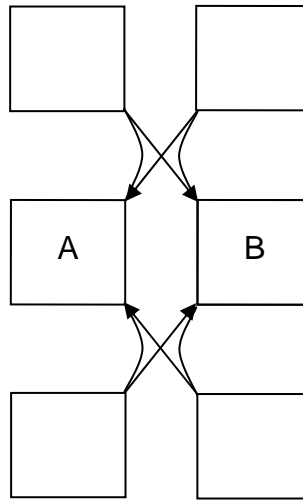


Figure 16: Cells A (left-going) and B (right-going) are the only two cells driven by the output diagonals of the cells directly above and below them. These wires can thus be easily checked for conditions that result in the inputs of A and B being unroutable.

5 Triptych Placer Results

Since this placement algorithm has been implemented previously, but none of the code was available for reuse, the Triptych placer was recoded from scratch. It was then necessary to determine values for the weighting parameters A, B, and C presented in the previous section. It was also necessary to experimentally determine the pegs per RLB threshold for contribution to the peg cost, discussed previously. After determining these parameters, the new Triptych placer's performance was then compared against that reported [Ebeling95] to verify that it was a "good" implementation.

Independence also has a tunable weighting parameter, λ , discussed in section 3.1. The best value for λ was experimentally determined as well. Both tools were then compared based upon the results after all optimizations were made.

5.1 Determining the Triptych Placer's Parameters

To evaluate performance for a given parameter combination, the array size (in RLBs) was parameterized with a fixed row-to-column ratio of 8:1. To create an analogous metric to that of [Sharma05] (to be reused when comparing against Independence), the minimum number of columns required to produce a routable netlist was used for comparison. In order to account for the random nature of simulated annealing, each placement trial was conducted ten times using different seeds for the random move generator. An array size that produced any routable placements in the ten runs was considered successful. For runtime considerations the maximum number of columns allowed before declaring the netlist "unrouteable" was 16. All tests were conducted on the ILSW93 netlists used for benchmarking in [Ebeling95]. The best value for a given parameter was chosen according to which value produced the minimum sum of the number of columns required to route all netlists in the benchmark suite, with netlists that failed to route assigned a score of 17.

Figure 17 shows the results of varying the peg cost threshold. The threshold value was varied from 0.1 to 2.5 pegs per RLB. Because all parameter testing was done in parallel, nominal values were used for the weighting parameters: $A = 1$, $B = 0.1$, and $C = 72$ corresponding to roughly equal weighting of each term in the cost function following initial placement of the smaller benchmark netlists. Peg cost threshold values of 0.8 and 0.9 produced the same minimum sum, although the minimum column size was not the same for every netlist (see Table 8 in Appendix).

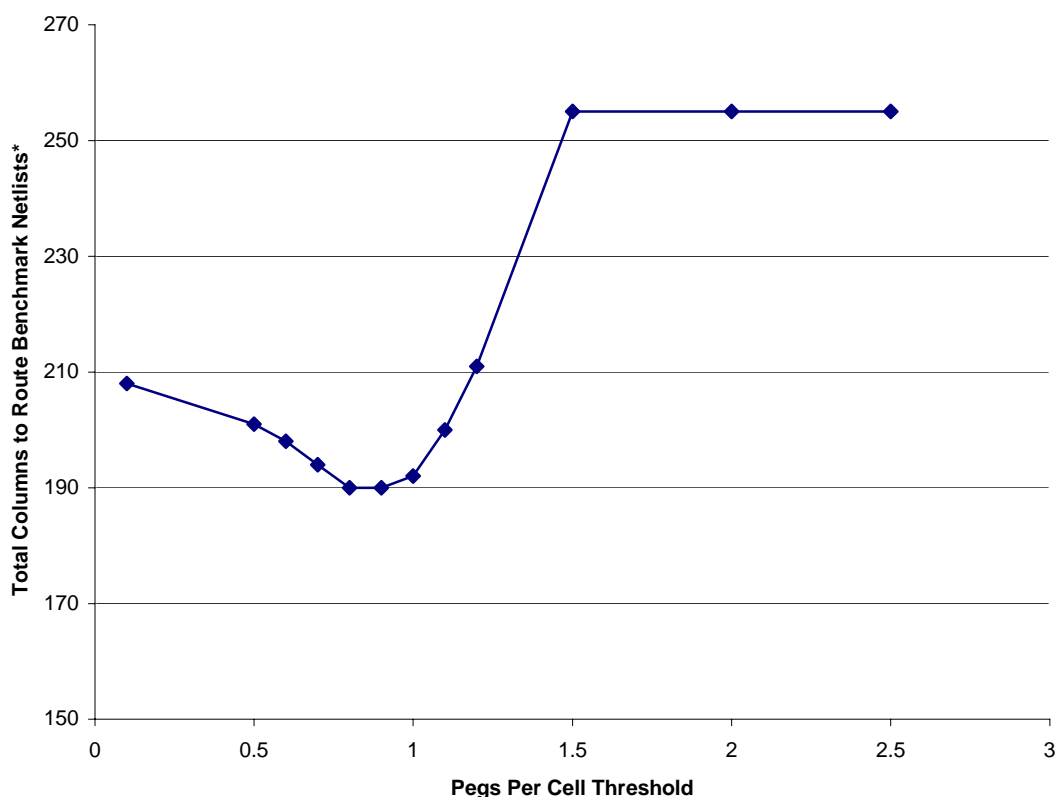


Figure 17: Determination of best pegs per cell threshold for the peg cost term in the Triptych placer cost function. The y axis shows the sum of the number of RLB columns needed to route the netlist, while the x axis shows the pegs per cell threshold. Values of 0.8 and 0.9 produced the best results.

For selection of the weighting parameters, the parameter A weighting wirelength cost was fixed at one. B and C were varied independently. B was tested in parallel with A, consequently nominal values for peg threshold of 1.0 and $C=72$ were used. Parameter

testing for C was done after other parameter testing; consequently best-choice values for peg threshold and B were used in determining C.

Figure 18 shows the results for tuning the peg cost weighting parameter B. The nominal value of $B = 0.1$ proved slightly superior to other choices in a similar range, while making B either very large or very small proved ineffective.

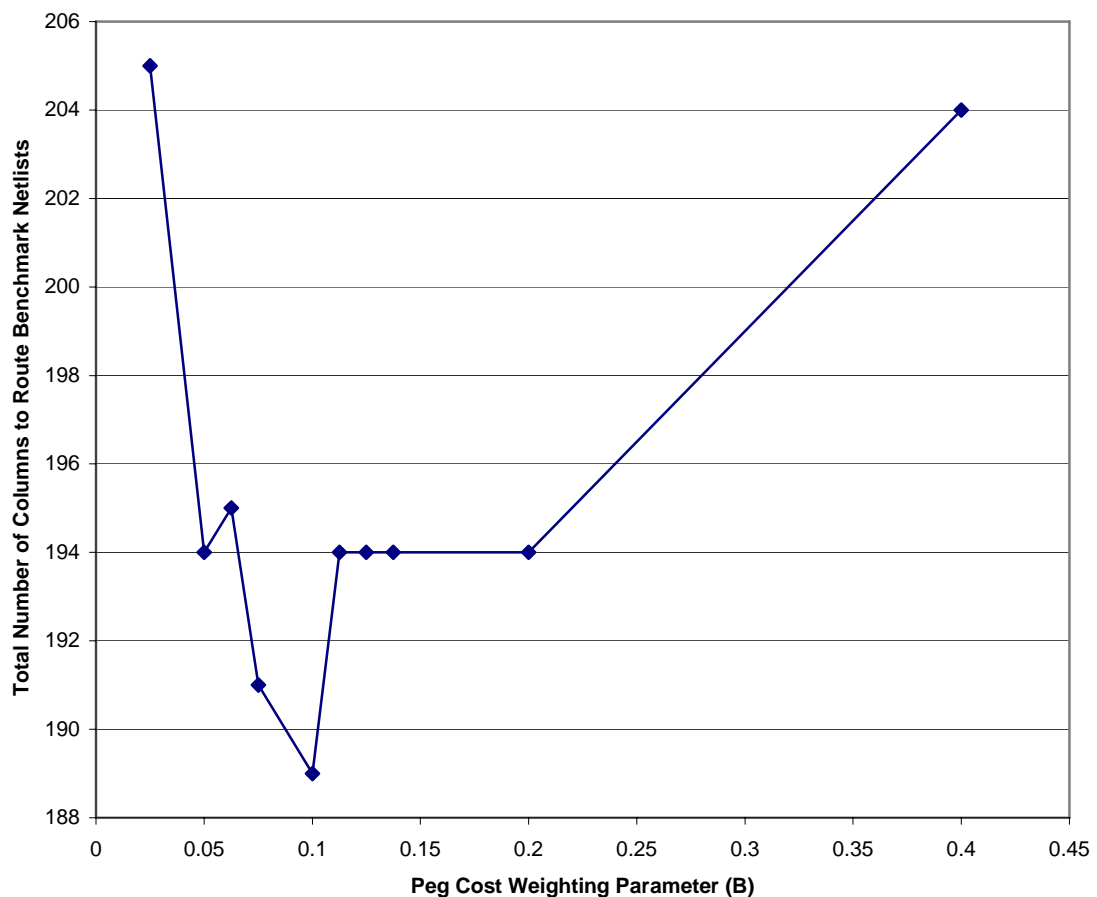


Figure 18: Determination of weighting parameter B. Within the range 0.05-0.2 the total number of columns remains very similar; the best choice for B found proved to be the nominal value of 0.1, corresponding to roughly equal weighting with wirelength.

Figure 19 shows the results for the local route cost weighting parameter C. The performance is highly similar over a wide range of nonzero values (see also Table 10 in Appendix), although a value of $C=1.125$ was shown to be slightly superior to both larger

and smaller values. When $C=0$, performance degrades severely (number of columns required = 228), indicating that while this cost term need not be weighted heavily it cannot be ignored.

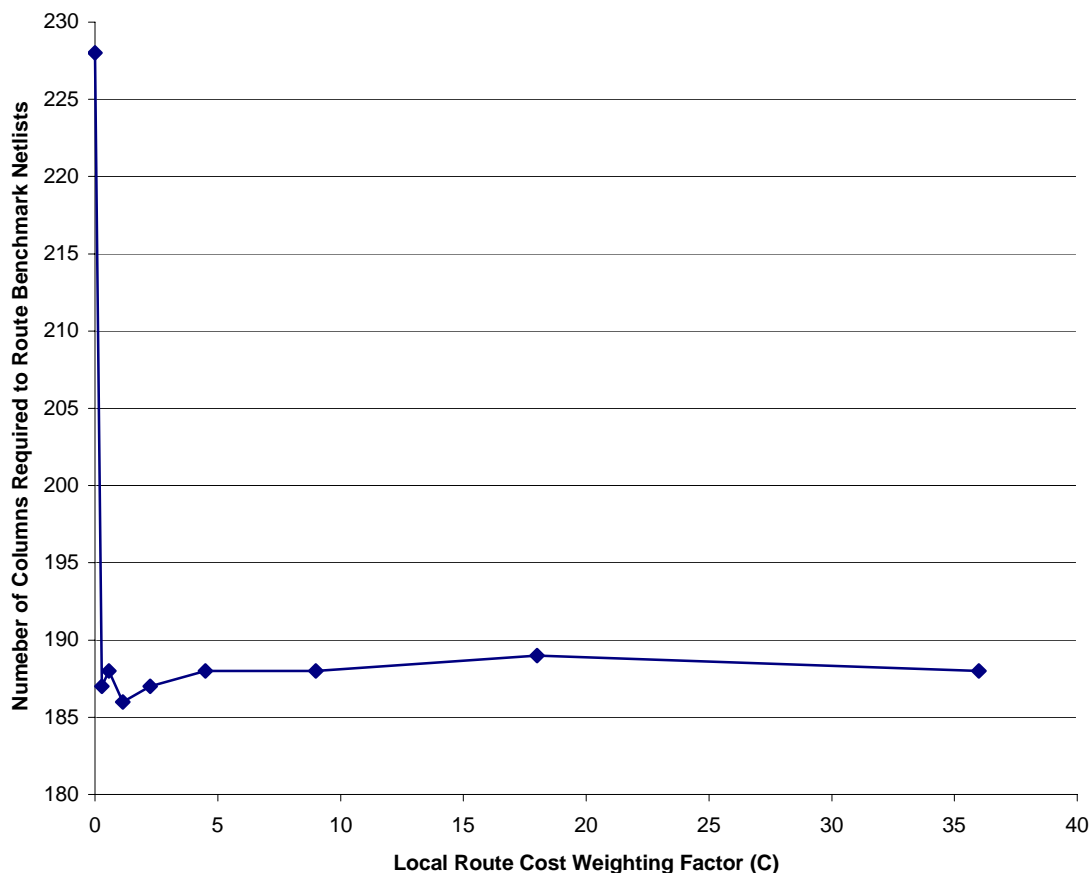


Figure 19: Determination of weighting parameter C. Note that for $C=0$ that many of the netlists are not routeable. The value of $C = 1.125$ proved to be the best among those tried.

5.2 Quality Versus Original Triptych Placer

The placement algorithm described above was originally implemented in [Ebeling95] and verified on a Triptych architecture using four-input RLBs. The array size was fixed at 64 rows by 8 columns. Verification was performed by selecting all ILSW93 netlists of size 150-300 RLBs with 128 or fewer I/Os, i.e. few enough to fit on the 64x8 array. The largest netlist successfully placed at this array size was 235 RLBs.

The same ILSW netlists were used to verify this implementation of the Triptych placer. However, the synthesis method used could not be reproduced; consequently the circuits tested were noticeably larger (185-457 logic blocks) than their published size in [Ebeling95]. Additionally, there is some ambiguity in [Borello95] as to the exact connectivity of the vertical channels in the 4-input RLB scheme as discussed in section 4.3.

Table 2 shows the benchmark netlists placed and routed in [Ebeling95] to a 64x8 array. The first column shows the netlist name. The second shows its size in RLBs as synthesized for [Ebeling95], while the third column shows the size in RLBs as synthesized for this work. Whether [Ebeling95] successfully routed the netlist to a 64x8 array is shown in the fourth column. As will be shown in section 5.4, only keyb was successfully routed to a 64x8 array in this thesis.

Table 2: Comparison of RLB count between [Ebeling95] synthesis and synthesis performed for this work.

Netlist	RLBs, [Ebeling95]	RLBs Current Synthesis	[Ebeling95] Routed?
ex1	150	220	YES
keyb	150	185	YES
C880	152	356	YES
clip	155	260	YES
C1908	159	258	YES
mm9b	163	447	YES
bw	169	232	YES
s832	173	243	YES
s820	176	212	YES
x1	192	242	YES
s953	220	372	YES
s1423	235	428	YES
styr	295	410	NO
planet	296	457	NO
planet1	297	457	NO

As Table 2 suggests, comparing the same benchmark netlists with such disparate RLB counts serves little purpose. Thus, to get additional insight as to how the quality of the new implementation compared to the original, benchmark netlists were reselected based upon logic block count under the synthesis scheme used for this work. Once again netlists with IO counts that exceeded the number available on a 64x8 array were rejected. Table 3 shows the results for each netlist meeting these criteria, the logic block count of each, and the number of columns required to route each netlist. As can be seen, the largest netlist that could be placed to a 64x8 array contained 187 logic blocks. It is unknown whether this performance gap is due to problems with the placement tool, issues arising from the difference in synthesis, or due to differences in the 4-RLB architecture implementation compared to the original implementation. It is also possible that the custom router developed for [Ebeling95] is superior to Independence's router for routing Triptych. Given that Independence implements Pathfinder for its router, however, this seems unlikely. Of all possible sources for the performance gap, architecture implementation differences seem most likely.

Table 3: Performance of Triptych Placer with 4-input RLBs on netlists with 150-300 logic blocks and less than 128 I/Os.

Netlist	Number of RLBs	Minimum Number of Columns to Route
s641	150	7
Tbk	152	7
Pma	155	7
Sao2	158	7
Cse	160	7
Term1	162	7
apex7	173	8
s510	174	8
9symml	176	7
mult32a	184	8
Keyb	185	8
dk16	187	7
example2	207	10
s820	212	9
ex1	220	9

Table 3 (continued)

Bw	232	9
C432	232	9
x1	242	10
s832	243	9
9sym	254	9
C1908	258	11
Clip	260	10
s838	270	10
rd84	273	11

5.3 Tuning Independence

The parameter λ weighting the congestion cost relative to the wire cost in Independence was varied between 1 and 32 and tested in the same manner as the Triptych placer parameter tests on the same netlists. Figure 20 shows a graph of the sum of minimum columns vs. λ for 3-input RLB and 4-input RLB architectures; the data for 3-input RLBs is provided in tabular form in the Appendix, Table 11, and the data table for 4-input RLBs is provided as Table 12.

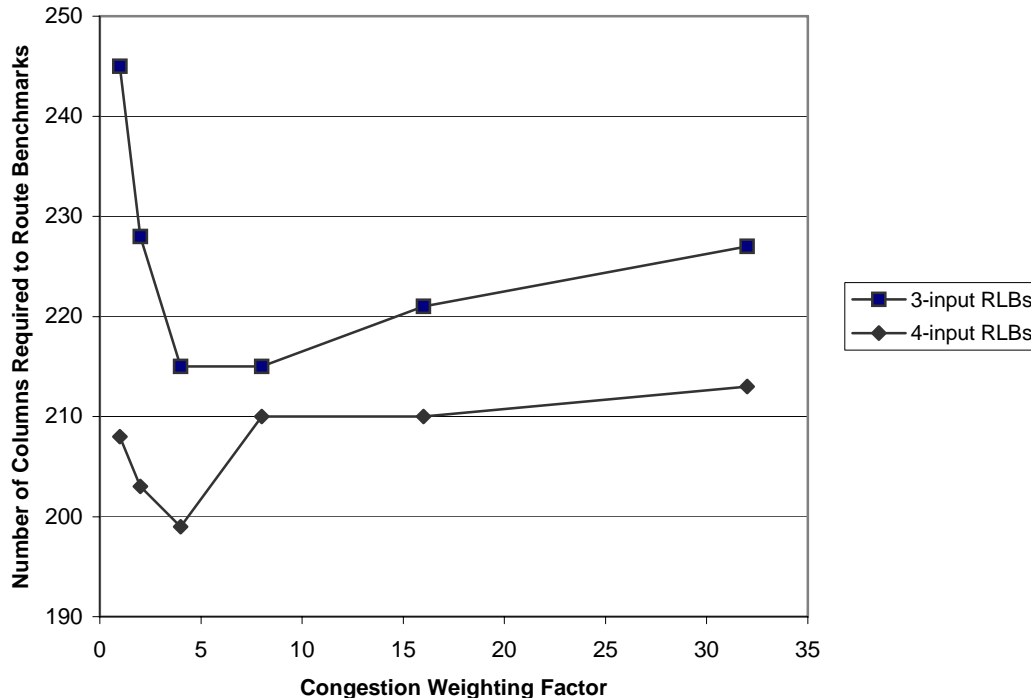


Figure 20: Graph of minimum number of columns required to route benchmarks vs. congestion weighting parameter λ . The upper line indicates results for 3-input RLB architectures, while the lower line indicates results for 4-input RLB architectures.

From these results, we can see that $\lambda = 4$ is about right for placing to Triptych. The overall shape of the curves for both architecture variations is similar to results for the architectures tested in [Sharma05], although it appears that Triptych benefits from slightly higher weighting of congestion. This is consistent with the intuition that because Triptych is a routing-poor architecture it will be more sensitive to routing congestion than a routing-rich architecture. Additionally, it was found that $\lambda = 4$ roughly corresponded to giving equal weight to wirelength and congestion in initial placement costs for smaller benchmarks, which is consistent with the equal weighting that wirelength and peg cost receive in the Triptych custom placer.

5.4 Comparison to Independence

All tests comparing Independence to the Triptych placer were tested on both 3-input and 4-input architectures using the test methodology described in section 5.1. For both placement tools the best values for the tuning parameters found in the tests of previous sections were used— $B = 0.8$ and $C = 4.5$ for the Triptych placer, and $\lambda = 4$ for Independence

Table 4 shows the results for each of the ILSW93 netlists tested on the 3-input RLB architecture. The second and third columns indicate the number of inputs and outputs in the netlist, respectively. The fourth column shows the number of logic blocks. The minimum number of columns needed for the Triptych placer to produce a routable netlist is shown in the fifth column, while the sixth column shows the same for Independence.

Table 4: Benchmark results for Triptych placer vs. Independence, 3-input RLBs. Netlists that failed to route were assigned a value of 17 for computing the sum.

Netlist	Inputs	Outputs	Logic Blocks	Triptych	Independence
Keyb	8	2	185	9	11
s820	19	19	212	10	11
ex1	9	19	220	10	12
Bw	5	28	232	10	11
x1	51	35	242	11	13
s832	19	19	243	10	12
C1908	33	25	258	12	14
Clip	9	5	260	11	13
C880	60	26	356	14	FAIL
s953	17	26	372	14	16
Styr	10	10	410	14	FAIL
s1423	18	5	428	15	FAIL
mm9b	13	9	447	16	FAIL
Planet1	8	19	457	15	FAIL
Planet	8	19	457	16	FAIL
SUM				186	215

Table 5 shows the same data for the 4-input RLBs.

Table 5: Benchmark results for Triptych placer vs. Independence, 4-input RLBs

Netlist	Inputs	Outputs	Logic Blocks	Triptych	Independence
Keyb	8	2	185	8	9
s820	19	19	212	9	10
ex1	9	19	220	9	10
Bw	5	28	232	9	10
x1	51	35	242	10	11
s832	19	19	243	9	11
C1908	33	25	258	11	13
Clip	9	5	260	10	11
C880	60	26	356	13	15
s953	17	26	372	13	15
Styr	10	10	410	13	16
s1423	18	5	428	13	FAIL
mm9b	13	9	447	14	FAIL
Planet1	8	19	457	14	FAIL
Planet	8	19	457	14	FAIL
SUM				169	199

In each architecture variation the Triptych placer requires fewer columns than Independence for every netlist in the benchmark. Additionally some netlists could not be routed to an array with 16 or fewer columns using Independence's placements. Because the sums in each table for Independence assign netlists that fail to route a value of 17, the total columns reported for Independence represent a lower bound on the actual number required. Thus, Independence requires *at least* 15.6% more columns than the Triptych placer for 3-input RLBs, and *at least* 17.8% more columns than the Triptych placer for 4-input RLBs.

Because previous results for Independence compared to previous placers had yielded similar or superior performance, this result merited further investigation. In all tests run on all benchmarks, there were five cases in which an Independence placement came

within one routing violation of placing to a smaller array than the smallest one it was able to route to successfully. For these five test cases, the placement cost of the final placement was computed using both tools' cost functions. The costs of the placements produced by the Triptych placer were then computed for the same five test cases. Note that in all five cases the Triptych placer's placement routed successfully.

Table 6 shows the placement costs calculated by Independence. The first column indicates the netlist. The second column indicates the array size. The third and fourth columns show the wire cost and congestion cost (unweighted) calculated for the Independence placements. The fifth and sixth columns show the wire cost and congestion cost calculated for the Triptych placer placements. The results obtained indicate that sometimes the calculated cost for the Triptych placer placements is in fact higher than that of the Independence placements despite the fact that they route successfully and the Independence placements do not. Overall, however, the average cost of each placer's placements as calculated by Independence are about the same. It is important to note that because these calculations were performed treating the placement as an initial placement, the history term of the congestion cost is zero. The congestion costs shown below thus are purely based upon Independence's initial attempt at routing the placement.

Table 6: Independence placement cost calculations for selected benchmark test cases

Netlist	Array Size	Independence		Triptych Placer	
		Wirelength	Congestion	Wirelength	Congestion
ex1	88x11	2915	1039	2848	990
Clip	96x12	3199	1207	3383	1212
Styr	128x16	5991	2260	5698	1996
C880 Case 1	128x16	5453	1849	5639	1896
C880 Case 2	128x16	5383	1869	5741	1896

Table 7 shows the placement costs calculated by the Triptych placer. The third through fifth columns break down the cost of each Independence placement, while the sixth

through eighth columns break down the cost of each Triptych placer placement. In each case the wire cost of the Independence placement is at least twice that of the corresponding Triptych placement. It can also be seen that while the Triptych placer ultimately reduces the peg cost and local route cost of each placement to zero, Independence does not necessarily achieve this. This is particularly important for the local route cost since in general a nonzero local route cost implies that the placement is not routeable.

Table 7 Triptych placement cost calculations for selected benchmark test cases

Netlist	Array Size	Independence			Triptych Placer		
		Wire Cost	Peg Cost	Local Route	Wire Cost	Peg Cost	Local Route
ex1	88x11	14.124	2.201	0	5.611	0	0
Clip	96x12	12.120	1.681	0	6.058	0	0
Styr	128x16	16.303	1.658	0	6.223	0	0
C880 Case 1	128x16	16.425	1.558	0.010	6.427	0	0
C880 Case 2	128x16	16.915	0.861	0.20	6.579	0	0

Taken together, the results shown in Table 6 and Table 7 demonstrate that the Triptych placer's cost function is better able to discern placement quality than that of Independence. In particular, the importance of the local routing violation check is again clear as it was in testing the Triptych placer by itself. Figure 21 displays the area surrounding the location of the single local routing violation for Independence's placement of the C880 netlist denoted Case 1 in the above tables. In this case, the routing violation occurred because cell 423gat_155_ with three required inputs was placed adjacent to an IO pin for which it produced the required output. Because there is only one route to the output pin, it is impossible to route the netlist given this placement. A local routing check by the Triptych placer detects this type of violation.

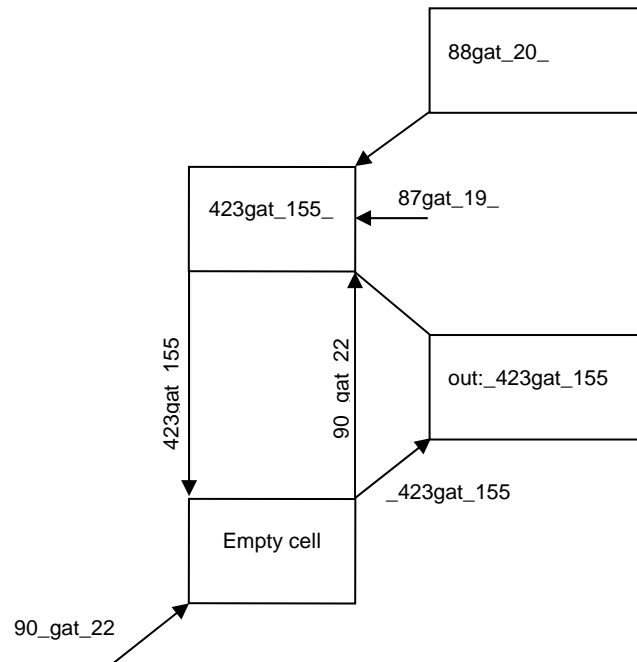


Figure 21: An example of a local routing violation produced by an Independence placement. In this case, net `_90_gat_22_` and net `_423gat_155_` are both trying to use the empty cell's upper diagonal. Because cell `423gat_155_` requires inputs to all three pins, and none of these inputs is the net being routed to output `out:_423gat_155`, it can be determined purely from this context that the netlist is not routeable.

A more generic type of local routing violation occurred in Independence's placement of C880, denoted Case 2 in the tables, and is shown in Figure 22. In this case, logic blocks have been placed to both RLBs in an RLB pair, shown in the middle. Each block has three inputs, and so all four diagonal inputs must be used to route this placement. As can be seen in Figure 22, the cell [254]'s diagonal cannot be used in routing any of the input nets to the middle RLBs because none of its input or output nets match those required. This also would be detected by the Triptych placer's local routing violation check.

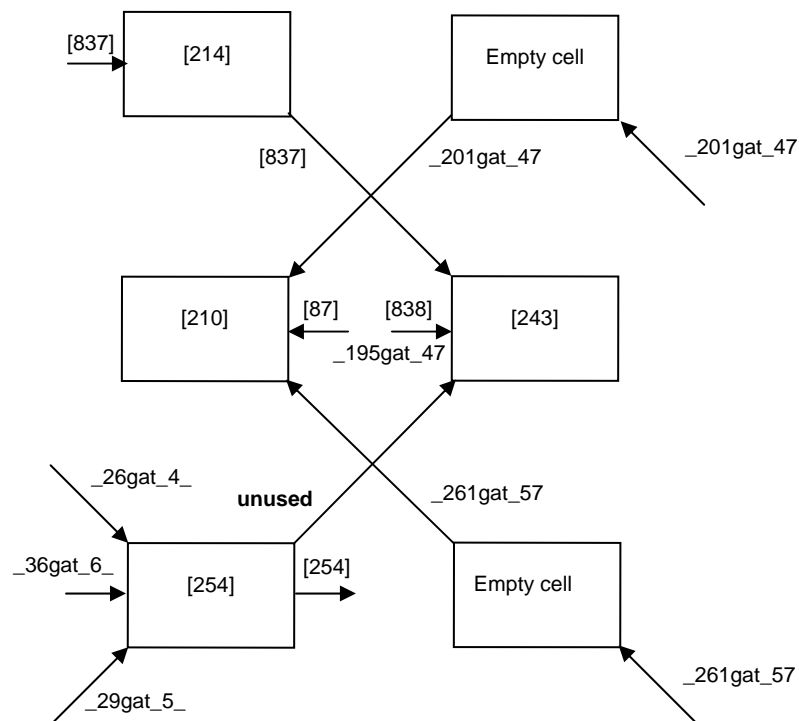


Figure 22: A second example of a local routing violation produced by Independence. In this case, six different nets need to be routed to the RLB pair [210] and [243]. This means that all four adjacent RLBs who have diagonals connecting to this pair must be used to route the six required nets. Because cell [254] has three input nets and an output net that are not required by [210] or [243], its upper diagonal cannot be used in routing and the netlist is therefore guaranteed to be unroutable.

The need for this check underscores the importance of localized routing and congestion to Triptych placements. With this already in mind, it is then also clear why the peg cost calculation, which essentially measures local congestion across the entire array, might have advantages over the congestion calculation of Independence, which in general reflects congestion in a more global sense.

6 Conclusion

The new Triptych placement tool demonstrated superior performance in terms of area to Independence. Independence required 15.6% more columns to route the benchmark netlists to 3-input RLB architectures and 17.8% more columns to route the benchmark netlists to 4-input RLB architectures. Investigation of selected test cases demonstrates the Triptych placer cost function's higher sensitivity to both wirelength and congestion compared to Independence, as shown in Table 6 and Table 7. Wirelength estimates calculated by Independence were similar for test cases in which Independence failed to produce a routable placement but the Triptych placer succeeded. The peg cost heuristic for measuring routing density appears to have advantages over Independence's congestion estimate for evaluating placement quality on the Triptych architecture. Additionally, two out of five test cases studied were found to have failed to route specifically due to local routing violations of the type the Triptych custom placer identifies. By contrast, the Independence cost function calculations for the five test cases examined produced similar costs for placements produced by Independence that failed to route and placements produced by Triptych that routed successfully.

7 Future Work

Although this thesis demonstrates that the reason for Independence's poor performance compared to the Triptych custom placer stems from a lack of sensitivity in its cost function, the reasons for this deficiency have yet to be explored. Future work in this area can be logically divided between examining wirelength and examining congestion. An explanation is needed as to why Independence placements in the test cases considered had wirelength as measured by the Triptych wirelength heuristic of more than twice the value of Triptych placer placements to the same array size. Similarly an explanation is needed as to why for the same test cases Independence produces similar wirelength estimates for each tool's placements. Further work could also look attempt to address two issues regarding Independence's congestion. The first is Independence's seemingly limited ability to eliminate local routing violations. The second is Independence's inability to distinguish congestion of routable placements and unroutable placements to the Triptych architecture. Future work could investigate these issues by comparing cost calculations for test cases such as those presented here on a net-by-net or node-by-node basis to more precisely ascertain what place-and-route scenarios prevent Independence from performing similarly to the Triptych custom placer.

References

[Betz97] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Seventh International Workshop on Field-Programmable Logic and Applications*, pp 213-222, 1997

[Chang97] Y.-W. Chang; D.F. Wong; C.K. Wong, "Universal Switch-Module Design for Symmetric-Array-Based FPGAs", *Proceedings of the 1996 ACM Fourth International Symposium on Field-Programmable Gate Arrays*, 1996.

[Cronquist99] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and Carl Ebeling. "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.

[DeHon99] A. DeHon. "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or why you don't really want 100% LUT utilization)", *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, 1999.

[Ebeling95] C. Ebeling, L. McMurchie, S. Hauck, S. Burns, "Placement and Routing Tools for the Triptych FPGA", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 473-482, December, 1995.

[Borellio95] G. Borriello, C. Ebeling, S. Hauck, and S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 491-501, December, 1995.

[Lam88] J. Lam and J. Delosme, "Performance of a New Annealing Schedule", *DAC*, 1988, pp. 306-311.

[McMurchie95] L. McMurchie and C. Ebeling. "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays Aided Design*, pp. 111-117, February 1995.

[Swartz90] W. Swartz and C. Sechen, "New Algorithms for the Placement and Routing of Macro Cells", *ICCAD*, pp. 336-339, 1990.

[Sharma01] A. Sharma and S. Hauck, "Development of a Place and Route Tool for the RaPiD Architecture", *Master's Thesis, University of Washington*, 2001.

[Sharma05] A. Sharma and S. Hauck, "Place and Route Technologies for FPGA Architecture Advancement", *PhD Thesis, University of Washington*, 2005.

[Xilinx04] Xilinx, Inc. “Spartan II-2.5V FPGA Family: Complete Data Sheet”,
<http://direct.xilinx.com/bvdocs/publications/ds001.pdf>, August, 2004.

[Xilinx06] Xilinx, Inc. “Virtex-4 Capabilities”,
http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/index.htm, 2006.

Appendix: Data Tables for Triptych, Independence Tuning Parameters

Table 8: Minimum number of columns as a function of peg threshold. Netlists that failed to route were assigned a value of 17 for computing the sum.

Netlist	0.1	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.5	2.0	2.5
ex1	11	11	10	10	10	10	10	10	10	FAIL	FAIL	FAIL
keyb	10	9	9	9	9	9	9	9	11	FAIL	FAIL	FAIL
C880	16	15	15	15	14	14	14	15	FAIL	FAIL	FAIL	FAIL
clip	12	12	11	11	11	11	11	11	11	FAIL	FAIL	FAIL
C1908	13	13	13	13	12	12	15	FAIL	FAIL	FAIL	FAIL	FAIL
mm9b	FAIL	FAIL	FAIL	16	16	16	16	FAIL	FAIL	FAIL	FAIL	FAIL
bw	11	10	10	10	10	10	10	10	10	FAIL	FAIL	FAIL
s832	11	11	11	11	11	11	11	11	11	FAIL	FAIL	FAIL
s820	11	10	10	10	10	10	10	10	10	FAIL	FAIL	FAIL
x1	12	12	12	12	11	11	11	12	15	FAIL	FAIL	FAIL
s953	16	15	15	15	14	15	15	FAIL	FAIL	FAIL	FAIL	FAIL
s1423	FAIL	16	16	15	15	15	14	14	14	FAIL	FAIL	FAIL
styr	FAIL	16	16	15	15	15	15	15	FAIL	FAIL	FAIL	FAIL
planet	FAIL	FAIL	FAIL	16	16	16	16	16	FAIL	FAIL	FAIL	FAIL
planet1	FAIL	FAIL	16	16	16	15	15	16	FAIL	FAIL	FAIL	FAIL
SUM	208	201	198	194	190	190	192	200	211	255	255	255

Table 9: Minimum number of columns as a function of peg cost weighting parameter B. Netlists that failed to route were assigned a value of 17 for computing the sum.

Test	0.025	0.05	0.0625	0.075	0.1	0.1125	0.125	0.1375	0.2	0.4
ex1	10	10	10	10	10	10	11	10	10	11
keyb	11	9	9	9	9	9	9	9	9	9
C880	FAIL	15	15	14	14	14	15	14	15	16
clip	12	11	11	11	11	11	11	11	11	12
C1908	16	14	15	15	14	15	13	13	13	14
mm9b	FAIL	16	16	16	16	16	16	16	FAIL	FAIL
bw	10	10	10	10	10	10	10	10	10	11
s832	11	11	11	10	10	11	11	11	11	11
s820	10	10	10	10	10	10	10	10	10	11
x1	12	12	12	11	11	12	11	11	11	12
s953	FAIL	FAIL	16	15	15	14	16	FAIL	15	15
s1423	14	14	15	14	14	15	14	15	15	15
styr	FAIL	15	15	15	15	15	15	15	15	16
planet	16	15	15	16	15	16	16	16	16	FAIL
planet1	15	15	15	15	15	16	16	16	16	FAIL
	205	194	195	191	189	194	194	194	194	204

Table 10: Minimum number of columns as a function of routeability cost weighting parameter C. Netlists that failed to route were assigned a value of 17 for computing the sum.

Netlist	0	0.28125	0.5625	1.125	2.25	4.5	9	18	36
ex1	FAIL	10	10	10	10	10	10	10	10
keyb	9	9	9	9	9	9	9	9	9
C880	FAIL	14	14	14	14	14	14	14	14
clip	12	11	11	11	11	11	11	11	11
C1908	FAIL	12	12	12	12	12	12	13	12
mm9b	16	16	16	16	16	16	16	16	16
bw	13	10	10	10	10	10	10	10	10
s832	FAIL	10	11	10	11	11	11	11	11
s820	13	10	10	10	10	10	10	10	10
x1	FAIL	11	11	11	11	11	11	11	11
s953	FAIL	14	14	14	14	14	14	14	14
s1423	14	14	14	15	14	15	15	14	14
styr	15	15	15	14	14	15	15	15	15
planet	FAIL	16	15	15	15	15	15	16	15
planet1	FAIL	15	16	15	16	15	15	15	16
SUM	228	187	188	186	187	188	188	189	188

Table 11: Minimum number of columns as a function of congestion weighting factor, 3-input RLBs. Netlists that failed to route were assigned a value of 17 for computing the sum.

Netlist	1	2	4	8	16	32
ex1	16	12	12	11	12	13
keyb	14	12	11	10	11	11
C880	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
clip	FAIL	16	13	13	13	14
C1908	FAIL	FAIL	14	14	15	16
mm9b	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
bw	14	12	11	11	12	12
s832	FAIL	14	12	12	13	14
s820	15	12	11	12	12	12
x1	16	14	13	13	14	16
s953	FAIL	FAIL	16	FAIL	FAIL	FAIL
s1423	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
styr	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
planet	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
planet1	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
SUM	245	228	215	215	221	227

Table 12: Minimum number of columns as a function of congestion weighting factor, 4-input RLBs. Netlists that failed to route were assigned a value of 17 for computing the sum.

Netlist	1	2	4	8	16	32
ex1	11	10	10	10	11	11
keyb	9	9	9	9	10	10
C880	FAIL	16	15	16	16	FAIL
clip	12	12	11	12	12	12
C1908	14	13	13	13	14	14
mm9b	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
bw	10	10	10	12	11	11
s832	12	11	11	13	11	12
s820	10	10	10	11	11	11
x1	13	12	11	13	12	13
s953	16	16	15	FAIL	FAIL	FAIL
s1423	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
styr	16	16	16	16	FAIL	FAIL
planet	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
planet1	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
SUM	208	203	199	210	210	213