Simultaneous Place and Route for Wire-Constrained FPGAs

Darren C. Cronquist

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Technical Report 95-03-01
March, 1995

Simultaneous Place and Route for Wire-Constrained FPGAs

Darren C. Cronquist

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract

Simulated annealing placement algorithms which use minimum wire length metrics based on rectilinear approximations fail to accurately account for an FPGA's routing resources since the number of logic block interconnections could be limited, causing certain placements to rely on resources which may not exist. In this paper we present a simulated annealing-based placement algorithm which performs a simple but effective route after each swap. We will show that, on wire-constrained FPGAs, our algorithm is a better evaluator for a given placement than the faster wire length metrics. In particular, on the Triptych 3-input RLB 4×16 array the algorithm achieves final delays ranging from 3.5% to 21.5% faster than delays yielded by a cost function tailored for the architecture. In addition, the algorithm demonstrates its adaptability by producing even better results on the most recent variant of the Triptych architecture. Finally, we will show that our method can be implemented without an unreasonable increase in execution time.

1 Introduction

A field programmable gate array (FPGA) consists of an array of configurable logic blocks linked via a predefined but programmable interconnect. Most of today's FPGA tools separate the functions of technology mapping, placement, and routing into self-contained programs, despite the fact that their success is highly dependent on one another. To take advantage of this interdependency, research has focused on merging these tools into one cohesive unit. In 1991, Beetem introduced a Penalty-Driven Iterative Improvement scheme for simultaneously placing and routing FPGAs containing cells used for both logic and routing [1]. In 1992, Schlag et al. presented a routability-driven algorithm for technology mapping of LUT-based FPGAs [9]. In 1993, Nag and Roy introduced an incremental placer for row-based FPGAs which analyzes post-layout timing and routability information to obtain better placements [8]. In 1994, Togawa et al. proposed a method for the simultaneous place and route of symmetrical FPGAs based on hierarchical bi-partitioning [13]. This paper introduces a simulated annealing based placement algorithm for wire-constrained FPGAs which uses fast but informative routes during placement to incorporate wirability information into the cost function.

Simulated annealing, introduced by Kirkpatrick et al. in 1983 [5], is an algorithmic technique used to solve optimization problems. Applications of this process to placement have been well-studied [11] and have been shown to produce good results at the cost of long run times. To implement a simulating annealing-based placement algorithm, a cost function that accurately evaluates the routability of a given placement must first be determined. Then, a random perturbation of the current state, such as the swapping of two logic blocks, produces a proposed solution which is either accepted or rejected according to an acceptance function. A commonly used function for acceptance is $e^{-\Delta C/T}$ where ΔC is the change in cost and T, the temperature, is a control variable. To obtain a low-cost solution, the temperature is initially set to a high value, allowing almost all placements, including those with a high cost increase, to be accepted. As the temperature slowly decreases, a lower acceptance rate forces some cells to become fixed in a good location on the array. At very low temperatures, almost all cells are stationary and only solutions which do not increase the cost are accepted. The simulated annealing algorithm produces close to optimum results since it explores the state space of a problem without becoming trapped in local minima. By the time the temperature reaches zero, a value close to the global minimum is reached. As a matter of fact, with a properly chosen cooling schedule and a sufficient amount of execution time, the simulated annealing process is guaranteed to yield a global minimum [7].

Success of a simulated annealing-based placement algorithm is dependent on the cost function's ability to accurately evaluate the routability of the current placement. The optimal method would be to perform a detailed route for every proposed placement to determine such factors as congestion, total wire length, and length of the critical path. Unfortunately, a typical anneal may involve hundreds of thousands of swaps, and hence any computation done on a per swap basis must be fast. As a result, most simulated annealing cost functions are based on quickly computed metrics such as the total wire length or semiperimeter (approximates the cost of wiring a source to its set of sinks by one half the perimeter of the enclosing rectangle). Not surprisingly, such simplistic cost functions fail to take into account the complexity of the target architecture in terms of the location and density of the routing resources. As a result, additional metrics often are incorporated into the cost function to allow for better conformity to a particular architecture.

Our method performs a route on a per swap basis by constructing an approximation to the minimum cost steiner tree linking a signal's required connections. The minimum cost steiner tree problem can be phrased as follows: given a graph $G = (V, E)$, a weight for each edge in E, and a subset T of V, find a subtree of G containing all vertices in T such that the sum of the edge costs in the subtree is a minimum [14]. The set of points in this subtree are *steiner points* and will be denoted by S. Karp proved this problem to be NP-hard for undirected graphs [4], and hence no polynomial time solution is known to exist. Since an undirected graph is a special case of a directed graph, the problem remains NP-hard for directed graphs.

2 Simultaneous Place and Route

Routing on a per swap basis yields information about which wires in the current placement have the highest demand for use. Since these congested wires directly effect routability, incorporating a corresponding penalty into the placer's cost function better leads to routability. We propose a simultaneous place and route algorithm which determines congestion information by approximating the minimum cost steiner tree. Our method is derived from Takahashi and Matsuyama's polynomial time approximation algorithm to the minimum steiner tree problem [12] as described in section 2.1. After discussing the directed graph model for representing the FPGA, we introduce our modifications to this algorithm and describe the resulting cost function metrics in section 2.3.

Finally, sections 2.4 and 2.5 examine methods for improving the running time and make comparisons to the semiperimeter calculation. In our analysis of the running time for approximating the minimum steiner tree, we let $n = |V|$, $y = |S|$, and $z = |T|$.

2.1 Minimum Cost Steiner Tree Approximation

In 1980, Takahashi and Matsuyama proposed an approximation algorithm for computing the minimum cost steiner tree which guarantees a result within a factor of $2 - 2/z$ of the optimum, where z is the number of nodes which must be connected. Hence, for all z, this algorithm produces a steiner tree whose cost is no worse than twice the minimum cost steiner tree. The complete algorithm is presented in figure 1 for an undirected graph $G = (V, E)$, a set of required connections T, and an initially empty set of steiner points S. By using Dijkstra's algorithm for computing the shortest

```
1. Move any node from from $T$ to $S$.
2. Determine an $(s, t)$ pair such that $s \in S$, $t \in T$, and there
      exists a path from $s$ to $t$ which is minimum over all pairs.
3. Add to $S$ all vertices along a minimum path from $s$ to $t$ and
      remove $t$ from $T$.
4. If $|T| > 0$ then goto 2.
```

Figure 1: *Approximating a Minimum Cost Steiner Tree*

paths, the algorithm can be completed in $O(zn^2)$ time.

2.2 The Architecture Description

An FPGA's architecture is specified via a *directed* weighted graph $G = (V, E)$ of the routing resources. Each node in V represents a wire in the array, and each edge in E represents a possible connection (typically via a programming bit). In addition, each node has a non-negative weight representing the cost of using the wire during the search for shortest paths (discussed in section 3.1). The architecture description also defines the height and width of the array and specifies which wires represent inputs and outputs (for logic blocks and pads).

2.3 The Routing Algorithm: MST-SPAR

The MST-SPAR (Minimum Steiner Tree - Simultaneous Place and Route) algorithm constructs an approximation to the minimum steiner tree for each signal's *network* – its source wire and set of sink wires. For a particular placement, every signal in the array has a unique source wire (a global input pad or logic block output) and a list of sink wires (global output pads, logic block inputs). This collection of wires is the set of required connections, T. Because these connections have to be valid for a given circuit, at least one path must exist from a signal's source wire to each of its sink wires. Then, to guarantee reachability between the S and T sets, the first step in figure 1 is modified to move the unique source wire from T to S instead of an arbitrary node. In addition, during the construction of the steiner set S, MST-SPAR maintains a count on the number of *conflicts* – different signals sharing the same wire – in the entire array. Now, when multiple minimum paths exist between the S and T sets, the tie-breaker becomes the path with the fewest conflicts in its first wire (finding the path with the minimum conflicts over all wires would increase the run time by a factor of y). The complete algorithm is presented in figure 2.

```
Delete current steiner set $S$ for $sig$, decrementing conflicts.
Let $S$ contain solely the source wire of $sig$.
Let $T$ equal the set of sink wires of $sig$.
Repeat
    Find wires $s \in S$ and $t \in T$ with a least conflicted minimum path over all pairs.
    Insert each wire along this path into $S$, incrementing conflicts.
    Remove $t$ from $T$.
Until $T$ is empty
```

Figure 2: *MST-SPAR Algorithm for Signal* sig

The process of creating steiner trees yields two valuable cost function metrics. The *Resources* is the total wire length of all steiner trees and is comparable to a semiperimeter calculation (which attempts to estimate the size of a network's wire length). The other metric, the *Sharings*, is the total number of wire conflicts and helps analyze the local routability of the array. Both of these measures are incorporated into the cost function to provide an accurate assessment of the overall routability (section 3.3 discusses a possible cost function). Since the final placement is handed off to a delay-optimizing router, the number of final conflicts does not have to be zero. The goal of MST-SPAR is not to produce a *routed* placement but instead a *routable* placement. However, if the number of wire conflicts happens to be zero, and if the timing constraints happen to be met, then the router need not be run.

In many ways, the shortest path constraint on routing a node in S to a node in T may seem counter-intuitive, primarily since the placer is unable to balance wire usage by routing around highly congested areas. Instead, it forges ahead along shortest paths creating wire conflicts. However, the key to MST-SPAR is that the *annealing process*, not the routing, relieves this congestion by finding placements which reduce the cost and hence have fewer conflicts.

As with Takahashi and Matsuyama's algorithm, the running time of MST-SPAR is $O(zn^2)$. Unfortunately, a typical FPGA description contains so many nodes that this computation is too expensive to be performed on a per swap basis. Hence, in the next section we show how the running time can be reduced by precomputing the shortest paths.

2.4 Improving the Running Time

Since the FPGA's routing structure is represented as a directed graph with nonnegative weights (section 2.2), the shortest paths from a node to a set of sink nodes can be computed via Dijkstra's algorithm. Moreover, because these shortest paths are fixed, a table of shortest paths from *every* node to *every* sink node can be precomputed prior to placement. As a result, the placer can build the steiner tree faster by indexing this precomputed table of shortest paths. Now, selecting the least conflicted minimum path can be done by looking at all pairs (s, t) such that $s \in S$ and $t \in T$ in $O(yz)$ time. Thus, the algorithm is reduced from $O(zn^2)$ to $O(yz^2)$. This improvement can be quite substantial since a typical signal has both $y \ll n$ and $z \ll n$. In section 3.2, the amount of resources required to precompute the minimum paths is parameterized in terms of the height and width of the array.

2.5 Comparison to the Semiperimeter Algorithm

The great advantage of the semiperimeter method is its simplicity. To determine the perimeter of a rectangle enclosing the source and sink wires, only one pass needs to made over the list of wires while the minimum and maximum coordinates are determined. The perimeter can then be computed directly. Hence, the semiperimeter method is $O(z)$, which is $O(yz)$ times faster than MST-SPAR.

3 Algorithm Implementation

Prior to applying MST-SPAR to a specific architecture, several steps of preparation must be completed. First of all, a file containing a directed weighted graph of the FPGA's routing resources (section 2.2) is created. Determining weights for the nodes in the graph is discussed in section 3.1. Next, this graph is used to build a table of shortest paths from every wire to every sink (section 3.2). Finally, the best algorithmic parameters are determined in order to produce a cost function closely associated with the routability of the specified architecture (section 3.3).

3.1 Choosing Node Weights

The architecture description specifies a weight for every wire in the FPGA in order to give the placer information about the cost of using a particular node in the graph. This weighting can be used in situations in which a preference needs to be given to certain wire types. It may try to predict the real delay of using a particular wire in an attempt to reduce the critical path. For example, if a long wire spanning 8 blocks has more transmission gate fanouts than a wire connecting two adjacent blocks, the real-time delay of using the long wire would typically be higher. Hence, an architecture description with a higher weight on long wires than on nearest neighbors would encourage the placer to prefer the low-delay connections. Another possibility is having the weightings deter certain routes. For example, an architecture may have some wires which should be used only when absolutely necessary, such as a bus-based FPGA in which at least one logic block input is restricted to nearest neighbor connections. If a required signal is not produced at a nearest neighbor, a potentially costly logic block route-through is permissible. Thus, heavily weighting the route-through wires prevents their use in a steiner tree unless necessary.

3.2 Storing the Shortest Paths

Although the computation of the shortest paths is potentially time consuming, it can be performed once for an architecture description and then stored in a binary file. The size of this file depends on the array's dimensions, wire count, symmetry, and functional unit input permutability. The following analysis of the resources required for the shortest paths assumes a generic $Width \times Height$ array with x-inputs per functional unit and ignores the pad structure. We define the following: $Storage$ is the total amount of bytes required on disk or in memory, $Wires$ is the total number of wires in the FPGA, and $Sinks$ is the number of sink wires in the FPGA. Since minimum paths from all nodes to all sinks must be stored,

$$Storage = C_1 Wires Sinks,$$

where the constant C_1 is the number of bytes of storage required for each source/sink pair. Assuming one functional unit per logic block, the number of sink wires is

$$Sinks = x(Width)(Height).$$

Finally, the total number of wires is simply

$$Wires = C_2(Width)(Height) + B_c Width + B_r Height,$$

where C_2 is the number of wires per logic block, and B_r and B_c are the number of bus wires per row/column respectively. If all variables are constant except the array dimensions, then

$$Storage = O(Width^2 Height^2)$$

with a constant of xC_1C_2 on the $Width^2 Height^2$ term. For our experiments on the Triptych architecture (described in section 4.1), $C_1 = 6$, $C_2 = 15$, and $x = 3$. Hence the amount of resources required is approximately $270 Width^2 Height^2$ bytes. Thus, a 4×16 array requires approximately a megabyte of disk and memory while an 8×64 array requires over 65 megabytes! Fortunately, by taking advantage of architecture specific attributes, the resources required can be significantly reduced. For example, for LUT based architectures, the inputs to a function can be interchanged by modifying the function computed by the LUT. Hence, using one sink per LUT reduces the resource requirements by a factor of x. In addition, many FPGAs have symmetries within the architecture making the computation of the shortest path from *every* wire to *every* sink redundant. For example, if an architecture is symmetric along any diagonal, horizontal, or vertical bisector, the storage requirements are cut by a factor of 2. For the Triptych architecture, the LUT optimization was performed, yielding a minimum paths file consuming approximately a third of a megabyte.

3.3 Selecting a Cost Function Based on *Resources* and *Sharings*

Our experiments with MST-SPAR have shown that a simple linear combination of the global metrics *Resources* and *Sharings* produces an efficient cost function. Thus, the total cost of a given placement is denoted by:

$$Cost = w_r Resources + w_s Sharings$$

where the constants w_r and w_s are the resource and sharing weightings respectively. In some sense, these values define a trade-off between reducing wire usage and reducing wire conflicts. For example, if $w_s/w_r = 5$, the placer can reach a state equivalent in cost to its current placement by performing a series of swaps which eliminates one conflict while consuming five more wires. Alternatively, it could reduce the wire usage by five at the cost of creating one wire conflict. Because this trade-off is dependent on the amount and location of wires, the optimal values for w_r and w_s are architecture dependent. The best w_s to w_r ratio for Triptych was found to be between 2 and 4. For the experiments in section 4, we use $w_s/w_r = 3$.

4 Experimental Results

The performance of MST-SPAR is examined on two variants of the Triptych architecture (described in section 4.1). A relative performance is achieved by comparing the results of MST-SPAR's placements and routes against results from two other algorithms based on quick wire-length metrics (section 4.2). Tests are run on a variety of both combinational and sequential circuits with a wide range on LUT utilizations (section 4.3). Section 4.4 discusses how each algorithm handles the global inputs and outputs. Finally, sections 4.5 and 4.6 show results of experiments varying the target architecture, the benchmarks, and the number of iterations per temperature drop (the run time) for the three different cost functions. Values for these parameters are chosen to reveal information about routability, critical path delay, and running time.

4.1 The FPGA Architectures

FPGAs with a limited set of routing resources best illustrate the benefits of the MST-SPAR algorithm. As a result, tests are run on two versions of a highly wire-constrained FPGA, Triptych [2]. Both arrays are 4×16, have 64 RLBs (routing and logic blocks), one 3-input LUT per RLB, one latch per RLB, 16 vertical segmented buses in each column, and nearest neighbor connections. The bottom four rows of a 4×16 is shown (without globalIO pads) in figure 3. A key aspect of the Triptych array is that every other logic block processes its inputs and outputs in a different direction. For example, the bottom left RLB in figure 3 takes inputs from the left and produces outputs on the right (in the direction of the arrows), while its northern and eastern neighbors do the opposite. Since there are no horizontal buses, this architecture relies on the RLBs to be used as routing resources by "routing-through" signals (i.e. any signal, even one not required by the LUT, can enter and exit the RLB). The most wire-constrained version of Triptych was the original design which provided only 3-inputs per RLB. Since the LUT mappings typically have an average number of inputs close to three, the RLB inputs are often entirely dedicated to LUT signals, limiting the ability to perform route-throughs. In addition to the original model, we run tests on the latest version of Triptych which increases the route-through capability by adding an extra bus input and

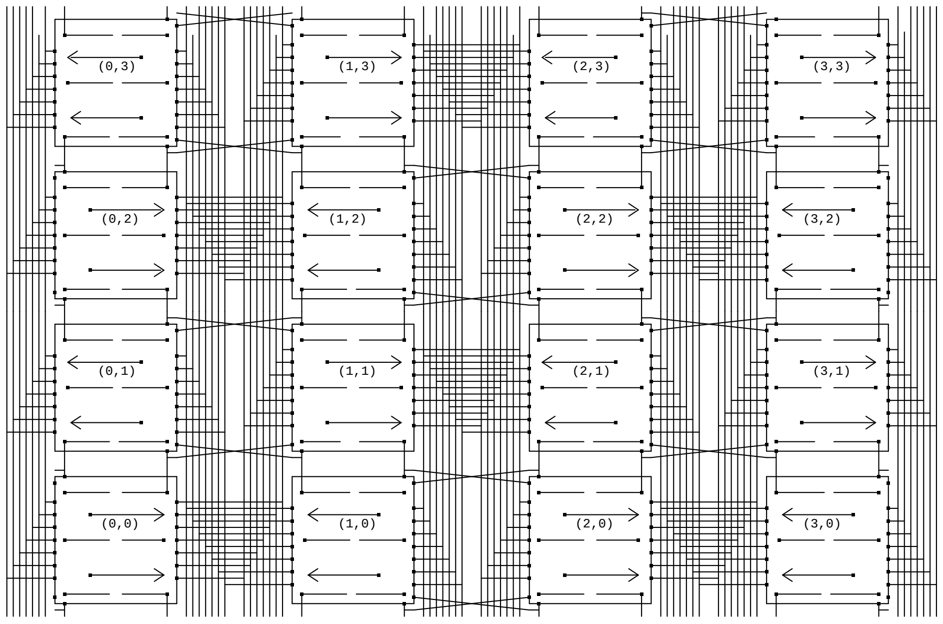

Figure 3: *Bottom 4 rows of Triptych* 4×16 *3-input RLB array*

output to each RLB. Throughout the remainder of this paper, the 3-input RLB Triptych array will be referred to as TriptychV3 and the 4-input RLB array as TriptychV4.

4.2 Three Algorithms: *Semi*, *Trip3*, and MST-SPAR

For both test architectures, MST-SPAR is compared against two cost functions based on fast wire-length metrics. The first, *Semi*, is a naive semiperimeter algorithm with no knowledge of the directional nature of the Triptych array. The second is *Trip3* – a cost function specifically tailored for the Triptych 3-input RLB array by incorporating an architecture specific semiperimeter calculation, density function, and local routability metric. In essence, this cost function preserves precious routing resources in highly congested areas by spreading empty RLBs throughout the array. All three annealers use a cooling schedule that performs a least squares approximation to predict the largest temperature drop that lowers the average cost by no more than a standard deviation [3]. A result of this method is that the number of iterations (swapings) per temperature drop is directly related to the run time of the algorithm. Hence, in order to vary the run times, the experiments parameterize the number of iterations. Once the three annealers have finished, all final placements are routed on a performance-driven negotiation based router [6].

Figure 4 shows three final placements for each of the three algorithms on the "squar5" ISCAS benchmark (described in section 4.3). The leftmost grid shows how *Semi*, without knowledge of the routing resources, simply tries to place all of the signals as close together as possible. As a matter of fact, the entire right column of the array remains empty! In sharp contrast, the center grid demonstrates how *Trip3* spreads empty RLBs throughout the array to better provide for routability. Finally, the rightmost grid shows how MST-SPAR achieves a combination of these two extremes

					out1	[8]				[15]	
					[20]		[15]	out4	[47]	[13]	
	[31]	[32]		out2		[16]		[37]			
out3	[34]	[36]		[10]	[22]		out5		out5	[20]	
[12]	[8]	out7			[47]		[13]	[16]	nn10	[22]	[29]
[29]	[16]	[45]		[17]		[34]			[18]		
nn10	out1	[44]			[40]	[37]			out0	[17]	out3
[18]	[17]	[42]		[25]			[41]	[41]		[25]	
out0	[15]	nn70			nn7		out6	[39]	[34]	nn70	
[22]	out5	[25]		out3		[39]			[42]	[31]	
out2	[13]	out6		[29]			out4		[32]	[45]	
[20]	[10]	[41]			[12]	[42]		[36]			
[47]	[37]	[40]			[31]	[36]		out6	out7	[44]	out2
	out4	[39]		out0			[44]		[10]	[12]	
				[18]		[32]	[45]		[40]	[8]	
					nn10	out7			out1		

Figure 4: Three Final Placements: Left is *Semi*, Center is *Trip3*, and Right is MST-SPAR

by having several dense areas interspersed with empty RLBs. For the three anneals in figure 4, *Semi*'s placement didn't come close to routing (the router's lowest number of wire conflicts was nine), *Trip3*'s placement routed with a delay of 58.5ns, and MST-SPAR's placement routed with a delay of 49.5ns. The placements shown in figure 4 were the best (minimum delay) from four separate runs each with a different random seed.

4.3 Benchmarks

Since both of the target architectures have 64 3-input LUTs, benchmarks were chosen with LUT-utilizations ranging from 21 used LUTs to 58 (table 1). These circuits were taken from ISCAS93 and mapped to 3-input LUTs with the synthesis tool SIS [10]. To ensure a quality mapping, four different SIS scripts were run on the benchmarks, each using combinations of script.algebraic, script.rugged, and full_simplify. The best mappings (fewest cells) were chosen for all circuits accept s349d, s349f, s386a, s386f, and s386g. These five benchmarks were artificially created (using a process based on the Fiduccia-Matteyses algorithm for partitioning) to produce benchmarks with high LUT-utilizations.

4.4 Handling Global Inputs and Outputs

Our initial tests used a fixed assignment of global input and output signals to Triptych's pads. Unfortunately, the *Trip3* algorithm wasn't designed to take advantage of a fixed pad assignment. Thus, in order to prevent the global inputs and outputs from skewing the results against *Trip3*, no global signals are fixed prior to placement. Instead, pad assignments for all global signals are determined by the router to best satisfy the routing constraints. Both MST-SPAR and *Trip3* take advantage of this by assuming that the pads can take on any global signal. In *Trip3* this is accomplished by swapping both the logic block and pad signals during the annealing process. MST-SPAR handles the unassigned pads by adding two artificial nodes to the graph: node *GlobalInputs*

Table 1: Benchmarks

Benchmark	Used LUTs	Used Latches	Global Inputs	Global Outputs	Avg. Inputs per LUT
pm1	21	0	16	13	2.52
x2	22	0	10	7	2.77
beecount	23	3	3	4	2.57
cu	25	0	14	11	2.64
mux	28	0	21	1	2.82
cmb	28	0	16	4	2.46
misex1	28	0	8	7	2.75
dk15	29	2	3	5	2.79
sqrt8	31	0	8	4	2.81
squar5	34	0	5	8	2.79
bbara	36	4	4	2	2.64
ex4	38	4	6	9	2.58
s386a	42	3	12	7	2.86
opus	45	4	6	6	2.71
misex2	48	0	25	18	2.75
s349d	50	11	6	10	2.72
s349f	55	13	8	9	2.71
s386f	56	4	12	7	2.80
f51m	56	0	8	8	2.70
s349g	58	13	10	10	2.64

fans-out to all of the global input pads while node *GlobalOutputs* is a fanout of all of the global output pads. Prior to running MST-SPAR, all global input signals are given *GlobalInputs* as a source node, and all global output signals are given *GlobalOutputs* as a sink node (these assignments induce no conflicts). As a result, MST-SPAR gives a preference for pad assignments by finding a placement which yields a low number of pad conflicts. However, as with *Trip3*, the ultimate choice for pads lies with the router. Unlike MST-SPAR and *Trip3*, *Semi* does not take advantage of the uncommitted global IOs. Instead, when computing its bounding box, it simply ignores all nodes associated with a global IO.

4.5 Results: Triptych 3-input RLB Array

We begin our tests with the highly wire-constrained TriptychV3 architecture (section 4.1). As an initial experiment, we fix the number of iterations at 8192 and run tests on the first 13 benchmarks in table 1. Each algorithm is executed four times with four random seeds on all benchmarks. For the tests which routed, table 2 lists the average number of wires used per test, the average final delay per test, and the minimum final delay over all tests (both in *ns* and normalized to MST-SPAR). For the tests that failed to route, the average minimum number of conflicts obtained by the router is reported (a circuit routes if and only if the number of wire conflicts is zero).

Semi failed to route 11 of the 13 circuits, MST-SPAR failed on three, and *Trip3* failed on only two. The semiperimeter's poor performance is clearly due fact that it doesn't consider the limitations of the routing resources. Placing all of the signals in one dense area (as in figure 4) doesn't account for Triptych's RLBs which can be used for *both* logic and routing. For the 11 benchmarks in which

Table 2: Results TriptychV3 with 8192 iterations per temperature drop

		Routed Circuits					*No Route*	
				Avg	Minimum			
			Avg	Final	Final			Avg
			Used	Delay	Delay			Min
Benchmark	Algorithm	#	Wires	(*ns*)	*ns*	Norm	#	Confl.
pm1	MST-SPAR	4	309	32.6	29.5	1.00	0	–
	Semi	1	368	48.0	48.0	1.63	3	1.0
	Trip3	4	344	36.9	34.0	1.15	0	–
x2	MST-SPAR	3	324	45.3	39.5	1.00	1	2.0
	Semi	0	–	–	–	–	4	2.7
	Trip3	4	367	51.6	48.0	1.22	0	–
beecount	MST-SPAR	4	282	31.4	29.5	1.00	0	–
	Semi	1	333	39.5	39.5	1.34	3	1.3
	Trip3	4	334	34.7	33.0	1.12	0	–
cu	MST-SPAR	4	356	40.0	35.5	1.00	0	–
	Semi	0	–	–	–	–	4	3.5
	Trip3	4	392	42.7	39.5	1.11	0	–
mux	MST-SPAR	3	409	52.1	49.5	1.00	1	1.0
	Semi	0	–	–	–	–	4	9.0
	Trip3	4	435	62.1	56.5	1.14	0	–
cmb	MST-SPAR	4	358	44.0	40.5	1.00	0	–
	Semi	0	–	–	–	–	4	4.7
	Trip3	4	416	52.7	45.5	1.12	0	–
misex1	MST-SPAR	4	420	64.0	56.5	1.00	0	–
	Semi	0	–	–	–	–	4	6.7
	Trip3	4	460	67.5	58.5	1.04	0	–
dk15	MST-SPAR	3	401	41.0	36.5	1.00	1	2.0
	Semi	0	–	–	–	–	4	5.2
	Trip3	4	452	43.9	38.0	1.04	0	–
sqrt8	MST-SPAR	0	–	–	–	–	4	2.2
	Semi	0	–	–	–	–	4	14.0
	Trip3	2	502	99.5	99.5	N/A	0	1.5
squar5	MST-SPAR	2	468	52.5	49.5	1.00	2	1.0
	Semi	0	–	–	–	–	4	8.7
	Trip3	3	524	63.6	58.5	1.18	1	1.0
bbara	MST-SPAR	1	449	52.0	52.0	1.00	3	1.7
	Semi	0	–	–	–	–	4	16.5
	Trip3	0	–	–	–	–	4	3.5
ex4	MST-SPAR	0	–	–	–	–	4	1.5
	Semi	0	–	–	–	–	4	13.0
	Trip3	1	561	97.0	97.0	N/A	3	1.0
s386a	MST-SPAR	0	–	–	–	–	4	8.2
	Semi	0	–	–	–	–	4	40.5
	Trip3	0	–	–	–	–	4	22.0

both *Trip3* and MST-SPAR succeeded in routing, MST-SPAR out-performed *Trip3* in every case. The minimum final delay for MST-SPAR was anywhere from 3.5% to 21.5% faster than that of *Trip3*. Similarly the average delay (over the trials which routed) was faster for all 11 benchmarks. This reduction in delay could be related to MST-SPAR's low wire usage. For these 11 benchmarks, MST-SPAR used anywhere from 9.5% to 18.4% fewer wires on average than did *Trip3*. These results are not surprising after looking once again at figure 4. In order to obtain better routability, *Trip3*'s cost function spreads empty RLBs throughout the array to be used as routing resources. However, placing these empty RLBs is highly dependent on the circuit parameters such as the number of used LUTs, number of IOs, and the total number of source/sink connections which must be made. On the other hand, MST-SPAR explores not only the routing structure during the annealing process, but also the attributes of the benchmark itself. Since it is performing a route after each swap, it can better predict where the congested areas will appear at route time. Moreover, because the placer's routes approximate minimum cost steiner trees, the router requires fewer wires and in turn produces routed circuits with a faster delay than that of *Trip3*.

Table 2 showed the performance that can be gained by using MST-SPAR. However, for a fair evaluation, the run times of the algorithms must first be examined by varying the number of iterations per temperature drop. Table 3 shows how the performance of MST-SPAR and *Trip3*

Table 3: Results on TriptychV3 and benchmark "cmb"

			Routed Circuits				*No Route*	
Algorithm	Iter per Temp Drop	Anneal User Time (*min*)	#	Avg Used Wires	Avg Final Delay (*ns*)	Min Final Delay (*ns*)	#	Avg Min Conflicts
MST-SPAR	512	1.4	2	380	44.2	41.0	2	2.0
	1024	1.8	2	361	44.0	40.5	2	1.0
	2048	3.3	4	371	43.2	38.5	0	–
	4096	6.2	2	371	49.0	46.5	2	1.5
	5120	7.5	3	363	46.3	43.0	1	1.0
	6144	9.1	4	364	45.1	43.0	0	–
	8192	11.7	4	358	44.0	40.5	0	–
Trip3	1024	0.4	4	432	54.7	51.5	0	–
	2048	0.8	4	435	56.0	52.5	0	–
	4096	1.0	4	417	53.2	50.5	0	–
	6144	1.8	4	417	53.1	50.5	0	–
	8192	2.2	4	416	52.7	45.5	0	–
	12288	3.7	4	420	53.1	43.5	0	–
	16384	4.3	4	424	48.7	43.5	0	–

varies with running time (*Semi* was excluded from these tests because of its failure to produce routable placements). Even for small run times, MST-SPAR out performs the *Trip3* algorithm in terms of delay. MST-SPAR's lowest delay (38.5ns) was 13% faster than *Trip3*'s lowest and was actually discovered in less time (3.3 minutes vs. 3.7 minutes). As a matter of fact, only one running time, 6.2 minutes, didn't outperform *Trip3*'s best minimum of 43.5ns.

An issue raised by table 3 is routability. It seems that by equalizing the running time, the MST-SPAR algorithm produces better results than *Trip3* at the cost of less routability. To analyze this issue further, a more challenging benchmark, "squar5", is analyzed over different run times in table 4. As expected, both algorithms have some difficulty routing the benchmark, although *Trip3* does

Table 4: Results on TriptychV3 and benchmark "squar5"

			Routed Circuits				*No Route*	
Algorithm	Iter	Anneal User Time (*min*)	#	Avg Used Wires	Avg Final Delay (*ns*)	Min Final Delay (*ns*)	#	Avg Min Conflicts
MST-SPAR	512	2.5	1	543	64.0	64.0	3	3.3
	1024	3.5	0	–	–	–	4	1.7
	2048	7.0	1	530	52.5	52.5	3	2.0
	4096	14.1	0	–	–	–	4	2.2
	5120	16.7	1	491	47.5	47.5	3	1.3
	6144	20.7	0	–	–	–	4	2.0
	8192	27.5	2	468	52.5	49.5	2	1.0
Trip3	1024	0.4	2	548	69.5	62.5	2	1.0
	2048	0.9	2	560	66.5	65.0	2	1.0
	4096	1.8	2	525	60.0	59.5	2	1.0
	6144	2.4	4	521	61.7	54.0	0	–
	8192	3.5	4	524	63.5	58.5	0	–
	12288	4.7	3	524	63.6	58.5	1	1.0
	16384	6.7	4	524	58.4	54.5	0	–

much better than MST-SPAR. As a matter of fact, *Trip3* places and routes 21 of the 28 trials while MST-SPAR only places and routes 5. As with the "cmb" benchmark from table 3, MST-SPAR yields a lower minimum delay than *Trip3* on almost all of the placements which route. Moreover, the results in table 4 seem to show that routability of MST-SPAR's placements is not related to the running time. These experiments indicate a trade-off between the MST-SPAR and *Trip3* algorithms. On a circuit with a low LUT utilization, MST-SPAR should be used since it yields lower final delays even at comparable running times. However, on a circuit which pushes an architecture's resources to the limit, either algorithm could be used depending on whether routability or delay is the most important criterion. If routability is the main concern, one could repeatedly run the *Trip3* algorithm hoping to obtain a reasonable delay. On the other hand, if a low final delay is required, MST-SPAR could be run repeatedly hoping to find a placement which routes. Because of MST-SPAR's longer run time, more trials of the *Trip3* algorithm can be executed. However, the experiments seem to indicate than even many successful place and routes on *Trip3* do not outperform MST-SPAR in terms of delay. A related issue is the fact that the cost function could be changed on different trials. For example, MST-SPAR could vary the w_s and w_r parameters prior to the execution of each trial in hope of finding a ratio which better leads to routability. Similarly, *trip3* could vary its density and local routability weightings in an attempt to achieve a better delay. This could potentially improve the performance of both algorithms in terms of routability and delay.

4.6 Results: Triptych 4-input RLB Array

To examine the adaptability of the MST-SPAR algorithm, experiments are run on the 4-input RLB variant to the Triptych architecture, TriptychV4. Because higher utilizations can be achieved with this architecture, the last eight benchmarks from table 1 are tested, and the results are listed in table 5 for 8192 iterations per temperature drop and 4 trials per test. MST-SPAR performed the

Table 5: Results on TriptychV4 with 8192 iterations

		Routed Circuits					*No Route*	
				Avg	Minimum			
			Avg	Final	Final			Avg
			Used	Delay	Delay			Min
Benchmark	Algorithm	#	Wires	(*ns*)	*ns*	Norm	#	Confl.
s386a	MST-SPAR	4	731	47.0	43.5	1.00	0	–
	Semi	4	816	63.7	57.0	1.31	0	–
	Trip3	4	811	57.9	54.5	1.25	0	–
opus	MST-SPAR	4	748	73.0	67.0	1.00	0	–
	Semi	4	799	90.0	84.5	1.26	0	–
	Trip3	4	827	90.3	83.5	1.25	0	–
misex2	MST-SPAR	0	–	–	–	–	4	6.2
	Semi	0	–	–	–	–	4	9.0
	Trip3	0	–	–	–	–	4	11.2
s349d	MST-SPAR	4	818	46.6	42.5	1.00	0	–
	Semi	4	871	58.9	43.0	1.01	0	–
	Trip3	4	890	57.0	50.5	1.19	0	–
s349f	MST-SPAR	4	900	49.7	44.0	1.00	0	–
	Semi	4	936	64.5	58.5	1.33	0	–
	Trip3	0	–	–	–	–	4	4.0
s386f	MST-SPAR	1	912	52.0	52.0	1.00	3	2.0
	Semi	0	–	–	–	–	4	4.5
	Trip3	0	–	–	–	–	4	22.2
f51m	MST-SPAR	2	931	192.5	181.0	1.00	2	1.5
	Semi	0	–	–	–	–	4	2.2
	Trip3	0	–	–	–	–	4	7.2
s349g	MST-SPAR	3	917	52.7	48.5	1.00	1	2.0
	Semi	3	959	62.1	54.0	1.11	1	2.0
	Trip3	0	–	–	–	–	4	10.0

best by placing and routing 7 of the 8 benchmarks, while *Semi* placed and routed 5 of 8 and *Trip3* only 3 of 8. In addition, MST-SPAR had minimum final delays faster than the other two algorithms on all benchmarks. As a matter of fact, MST-SPAR even had the fewest wire conflicts for the trials which didn't route. The fact that the *Semi* algorithm outperforms *Trip3* was unexpected since the semiperimeter calculation has no knowledge of either the directional nature of the Triptych array or the global IOs. This result implies that changes to the Triptych architecture have made the *Trip3* algorithm ineffective. A possible explanation is that *Trip3*'s cost function metrics were designed to find a good placement for empty logic blocks in order to provide a certain level of routability. Since the TriptychV3 architecture had a maximum LUT utilization of around 60% (table 2 shows that "ex4" was the highest utilized circuit to place and route), the *Trip3* algorithm could be relying on the existence of these empty RLBs. With TriptychV4's additional RLB input and output leading to routable LUT utilizations of over 90%, the placement problem has dramatically changed. Now that empty blocks are no longer required in congested areas, even the semiperimeter algorithm is able to place and route over half of the benchmarks.

As with the TriptychV3 architecture, the running times of the algorithms on TriptychV4 must be analyzed prior to making final conclusions about performance. First, an easily routable circuit, "opus", is examined with results shown in table 6. Clearly, MST-SPAR outperforms the other two

Table 6: Results on TriptychV4 and benchmark "opus"

			Routed Circuits			
Algorithm	Iter	Anneal User Time (*min*)	#	Avg Used Wires	Avg Final Delay (*ns*)	Min Final Delay (*ns*)
MST-SPAR	512	3	4	773	82.5	77.0
	1024	5	4	765	81.0	73.5
	2048	10	4	764	81.5	72.5
	4096	17	4	754	74.5	73.5
	5120	22	4	739	75.7	72.0
	6144	27	4	750	80.2	77.0
	8192	35	4	748	73.0	67.0
Semi	1024	1	4	823	89.0	80.0
	2048	1	4	826	95.2	81.0
	4096	2	4	786	86.1	76.5
	6144	3	4	793	81.6	76.0
	8192	4	4	799	90.0	84.5
	12288	5	4	773	82.5	77.0
	16384	6	3	799	85.0	81.0
Trip3	1024	1	4	842	98.5	92.0
	2048	1	4	843	93.2	80.0
	4096	2	4	829	88.4	82.5
	6144	3	4	801	83.7	69.0
	8192	4	4	827	90.2	83.5
	12288	6	4	821	86.6	81.0
	16384	7	4	819	89.0	80.5

algorithms even at lower run times. As a matter of fact, only 2 of the 7 MST-SPAR run times produce a minimum final delay slower than *Semi*'s minimum delay over all run times (76.0*ns*). Furthermore, a long run time (35 minutes) for MST-SPAR yielded a final delay 13% faster than *Semi*'s best while a short run time (5 minutes) yielded a final delay 3% faster. These results for the minimum final delay do not extend to *Trip3* since one of the 28 trials "got lucky" and produced a delay of 69*ns* – a full 21*ns* below the average delay over all *Trip3* trials and 9.5*ns* below the second lowest trial. However, the *average* final delays show that MST-SPAR still outperforms *Trip3* (in the context of run time). As a matter of fact, all average final delays from MST-SPAR were faster than the lowest average final delay for *Trip3* (83.7*ns*).

As a last experiment, we address the issue of routability and running time on the TriptychV4 architecture by looking at the benchmark with the highest LUT-utilization, "s349g", in table 7. Both MST-SPAR and *Semi* successfully placed and routed 15 of the 28 trials. Not only did *Trip3* fail to place and route 26 of the 28 trials, but it also had many wire conflicts in these unroutable placements. Once again, MST-SPAR outperformed both *Semi* and *Trip3* in terms of delay. For all run times, MST-SPAR's average final delay was faster than both *Semi* (59.8ns) and *Trip3* (61.5ns). In addition, 4 of the 7 run-times had better minimum delays than the lowest minimum delay for *Semi* (52.0ns). Furthermore, an examination of all run times shows that MST-SPAR produced a minimum delay (42.5ns) that was 24% faster than *Semi* and 45% faster than *Trip3*.

Table 7: Results on TriptychV4 and benchmark "s349g"

			Routed Circuits				*No Route*	
Algorithm	Iter	Anneal User Time (*min*)	#	Avg Used Wires	Avg Final Delay (*ns*)	Min Final Delay (*ns*)	#	Avg Min Confl.
MST-SPAR	512	4	1	942	57.5	57.5	3	8.0
	1024	6	2	938	46.0	42.5	2	4.0
	2048	12	2	941	53.7	52.0	2	2.5
	4096	20	1	918	58.0	58.0	3	1.6
	5120	29	3	934	51.0	48.5	1	1.0
	6144	31	3	933	59.0	55.0	1	2.0
	8192	45	3	917	52.7	48.5	1	2.0
Semi	1024	1	1	970	71.0	71.0	3	2.3
	2048	2	0	–	–	–	4	1.5
	4096	2	3	954	63.0	52.0	1	1.0
	6144	3	3	957	59.8	53.0	1	1.0
	8192	5	3	959	62.1	54.0	1	2.0
	12288	5	2	960	70.5	53.0	2	2.0
	16384	7	3	952	67.0	61.0	1	1.0
Trip3	1024	1	0	–	–	–	4	12.7
	2048	1	0	–	–	–	4	12.7
	4096	2	1	991	65.5	65.5	3	11.0
	6144	3	0	–	–	–	4	9.5
	8192	4	0	–	–	–	4	10.0
	12288	6	0	–	–	–	4	4.5
	16384	9	1	976	61.5	61.5	3	7.3

5 Future Work

Since the experimental results for MST-SPAR are very promising, there are many possible avenues for further exploration. The most obvious is to examine how the algorithm extends to larger FPGA arrays such as an 8×32 or a 16×64. In particular, how do the extra resource requirements (section 3.2) affect the performance and run time? Can these time and resource requirements be reduced by partitioning the large arrays into smaller ones, while maintaining performance?

In addition to focusing on the array size, it would be interesting to examine MST-SPAR's performance on an FPGA with primarily bus-based routing resources. A key question is whether or not the steiner tree algorithm selects the best "steiner buses" when multiple sinks are fanouts of the same bus.

The experiments in section 4 showed that the MST-SPAR algorithm produced better timing results than both *Semi* and *Trip3* despite the fact that the algorithm does no critical path analysis. Since MST-SPAR actually performs routes during placement, it could also do a quick analysis of the critical path after each route and incorporate the results into the cost function. An interesting experiment would be to determine a quick and productive critical path metric and analyze its affects on MST-SPAR's run time and performance.

6 Conclusion

MST-SPAR has been shown to be an effective algorithm for placement on wire-constrained FPGAs such as Triptych. This simulated annealing-based placement algorithm reroutes signals on a per swap basis in order to provide the cost function with vital routing statistics such as wire usage and wire conflicts. The experimental results on two versions the Triptych FPGA show that MST-SPAR yields lower critical path delays than two other algorithms based on simple wire length metrics. In particular, for a set of benchmarks mapped to the Triptych 3-input RLB 4×16 array, MST-SPAR produces placements with final delays ranging from 3.5% to 21.5% faster than delays yielded by a cost function tailored for the architecture. Furthermore, the experimental results show that even when its running time is reduced, MST-SPAR still produces placements with faster delays. In addition, the adaptability of MST-SPAR was demonstrated by showing similar performance results on a variant of the Triptych FPGA without making any algorithmic modifications. Hence, not only is MST-SPAR an attractive placement algorithm because of its performance, but also because its adaptability allows for an architecture to evolve without requiring major modifications to the placement tools.

7 Acknowledgments

I would like to thank Larry McMurchie for his numerous valuable discussions and both Carl Ebeling and Gaetano Borriello for their suggestions and support.

References

[1] J. F. Beetem, “Simultaneous Placement and Routing of the LABYRINTH Reconfigurable Logic Array,” *The International Workshop on Field Programmable Logic and Applications*, 1991, pp. 232-243.

[2] S. Hauck, G. Borriello, and C. Ebeling, “TRIPTYCH: An FPGA Architecture with Integrated Logic and Routing,” *Proceedings of the 1992 Conference on Advanced Research in VLSI and Parallel Systems*, March 1992, pp. 26-43.

[3] M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, “An Efficient General Cooling Schedule for Simulated Annealing,” *IEEE International Conference on Computer-Aided Design*, 1986, pp. 381-384.

[4] R. M. Karp, “Reducibility Among Combinatorial Problems,” in R. E. Miller and J. W. Thatcher (eds.) *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-103.

[5] S. Kirkpatrick, C. Gelatt, and M. Vecchi, “Optimization by Simulated Annealing,” *Science*, 220/4598, 1983, pp. 671-680.

[6] L. McMurchie, C. Ebeling, and G. Borriello, “An Architecture-Adaptive, Performance-Driven Router for FPGAs,” TR #94-05-01, University of Washington, May 1994.

[7] D. Mitra, F. Romeo, and A. L. Sangiovanni-Vincentelli, “Convergence and Finite-time Behavior of Simluated Annealing,” *Proceedings of the 24th Conference on Decision and Control*, 1985, pp. 761-767.

[8] S. Nag and K. Roy, “Iterative Wirability and Performance Improvement for FPGAs,” *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993, pp. 321-325.

[9] M. Schlag, J. Kong, and P. Chan, “Routability-Driven Technology Mapping for LookUp Table-Based FPGAs,” TR #UCSC-CRL-92-06, University of California at Santa Cruz, February 1992.

[10] E. Sentovich et al., “SIS: A System for Sequential Circuit Synthesis,” Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA, May 1992.

[11] K. Shahookar and P. Mazumder, “VLSI Cell Placement Techniques,” *ACM Computing Surveys*, Vol. 23, No. 2, June 1991, pp. 143-220.

[12] H. Takahashi and A. Matsuyama, “An Approximate Solution for the Steiner Problem in Graphs,” *Math. Japonica*, Vol. 24, 1980, pp. 573-577.

[13] N. Togawa, M. Sato, and T. Ohtsuki, “A Simultaneous Placement and Global Routing Algorithm for Symmetric FPGAs,” *The 2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, Session 8, 1994.

[14] P. Winter, “Steiner Problem in Networks: A Survey,” *NETWORKS*, Vol. 17, 1987, pp. 129-167.