Accelerating FPGA Routing Using Architecture-Adaptive A* Techniques

Akshay Sharma
Actel Corporation
Mountain View, CA – 94043, USA
Akshay.Sharma@actel.com

Scott Hauck
University of Washington
Seattle, WA – 98195, USA
hauck@ee.washington.edu

Abstract

The A algorithm is a well-known path-finding technique that is used to speed up FPGA routing. Previously published A*-based techniques are either targeted to a class of architecturally similar devices, or require prohibitive amounts of memory to preserve architecture adaptability. This work presents architecture-adaptive A* techniques that require significantly less memory than previously published work. Our techniques are able to produce routing runtimes that are within 7% (on an island-style architecture) and 9% better (on a hierarchical architecture) than targeted heuristic techniques. Memory improvements range between 30X (island-style) and 140X (hierarchical architecture).*

1. Introduction

Routing is an important step in the FPGA toolflow. FPGAs have a finite number of discrete routing resources, and the effectiveness of an FPGA router directly impacts the performance of an application netlist on a target device. Pathfinder [8] is the current, state-of-the-art FPGA routing algorithm. Pathfinder uses an iterative, negotiation-based approach to solve the FPGA routing problem. During the first routing iteration, nets are freely routed without paying attention to resource sharing. Individual nets are routed using a shortest path graph algorithm. At the end of the first iteration, resources are generally congested because multiple nets have shared them. During subsequent iterations, the cost of using a resource is increased based on the number of nets that share the resource, and the history of congestion on that resource. In effect, nets are made to negotiate for routing resources. If a resource is highly congested, nets that can use lower congestion alternatives are forced to do so. On the other hand, if the alternatives are more congested than the resource, then a net may still use that resource.

Pathfinder has proved to be one of the most powerful FPGA routing algorithms to date. Pathfinder's negotiation-based framework is a very effective technique for routing nets on FPGAs. More importantly, Pathfinder is a truly architecture-adaptive routing algorithm. The algorithm operates on a directed graph abstraction of an FPGA's interconnect structure, and can thus be used to route netlists on any FPGA that can be represented as a directed routing graph. We believe that Pathfinder's adaptability is one of the main reasons for its widespread acceptance.

When routing an individual net, Pathfinder uses a greedy search algorithm that is similar to Dijkstra's algorithm [5]. A net that has n sink terminals is routed using n searches. Further, each net may be ripped up and rerouted multiple times as the algorithm progresses through routing iterations. All in all, employing a search-based algorithm to do FPGA routing is a computationally expensive process.

A path-finding technique that is commonly used to speed up graph-based search is the A* algorithm [9]. The A* algorithm speeds up routing by pruning the search space of Dijkstra's algorithm. The search space is pruned by preferentially expanding the search wavefront in the direction of the target node. When the search is expanded around a given wire, the routing algorithm expands the search through the neighbor wire that is nearest the target node. This form of directed search is accomplished by augmenting the cost of a routing wire with a heuristically calculated estimate of the cost to the target node.

Equation 1: $f_n = g_n + h_n$

Consider Equation 1, in which g_n is the cost of a shortest path from the source to wire n, and h_n is a heuristically calculated estimate of the cost of a shortest path from n to the target node (hereafter, we refer to this estimate as a 'cost-to-target' estimate). The value f_n is the estimated cost of a shortest path from the source to the target that contains the wire n. The A* algorithm uses f_n to determine the cost of expanding the search through wire n. Note that Dijkstra's algorithm uses only g_n to calculate the cost of wire n.

To guarantee optimality, the cost-to-target estimate h_n at a given wire n must be less than or equal to the actual cost of the shortest path to the target. Overestimating the cost to the target node may provide even greater speedups, but then the search is not guaranteed to find an optimal path to the target. Currently, there is no architecture-adaptive, memory efficient technique for performing A* search on FPGAs. Our goal in this paper is the development of architecture-adaptive A* techniques that can be used to speed up the FPGA routing process.

2. Previous work

The work described in [11,12] discusses directed search techniques that can speed up the Pathfinder algorithm. These techniques are similar to the A* algorithm, and use a formulation like the one shown in Equation 1 to calculate the cost of expanding the search through a routing wire. During the routing process, cost-to-target estimates are heuristically calculated using geometric information. Estimate calculations often require potentially complex operations, and the cost of calculating estimates can slow down the router. Further, the techniques presented in [11,12] may need to be re-implemented whenever the interconnect architecture is changed, and are not suitable for non-Manhattan interconnect structures. Two examples of such interconnect structures are shown in Figure 1. The architecture on the left [2] provides different types of routing resources in the horizontal and vertical directions, and the architecture on the right [6] has a strictly hierarchical interconnect structure.

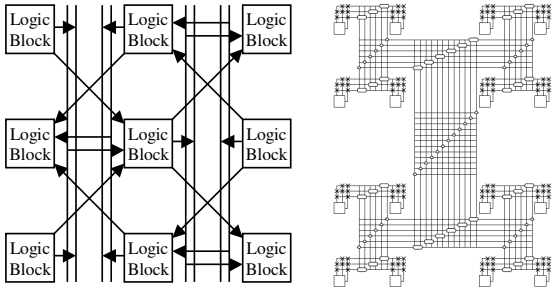

Figure 1. Non island-style interconnect structures [2,6].

Recall that Pathfinder's primary strength is its adaptability to different FPGA architectures. Existing A* techniques violate this assumption, since they hard-code interconnect assumptions into the cost-to-target estimators. In this paper, we present architecture-adaptive runtime enhancements to the Pathfinder algorithm. Our techniques are also based on using the A* algorithm to speed up the search process. However, our methodology is adaptive and does not rely on architecture-specific heuristic cost-to-target estimates. The techniques presented in this paper are routability-driven. Extending these techniques to include timing information may be accomplished in a manner similar to that described in [1].

3. Architecture-adaptive A* techniques

The developers of the Pathfinder algorithm briefly discussed the idea of using the A* algorithm to speed up routing [8]. They proposed the use of a pre-computed lookup table that would hold the cost of a shortest path from every routing wire to every sink terminal in the interconnect structure. Specifically, there would be a separate entry for every routing wire in this lookup table, and each entry would hold cost-to-target estimates for all sink terminals in the interconnect structure. During routing, the cost-to-target estimate at a routing wire could then be obtained using a simple table lookup.

Pre-computing and tabulating cost-to-target estimates in this fashion is indeed an adaptive scheme. Shortest paths can be calculated using Dijkstra's algorithm, and no architecture-specific information is required. The approach also guarantees an exact estimate of the shortest path in the absence of routing congestion. However, while the computational complexity of this approach is manageable, the space requirements for routing-rich structures may explode. Assuming an island-style, 10-track, 100x100 FPGA that has only single-length segments, the memory required to store the cost-to-target lookup table would be measured in GigaBytes. Memory requirements of this size are probably impractical.

Sharing a table entry among multiple routing wires that have similar cost-to-target estimates can reduce the memory requirement of the lookup table. For example, if one hundred wires share each table entry, the size of the table may be reduced by one hundred times. The cost-to-target estimate for a given sink terminal is the same for all wires that share the table entry, and can be calculated using a Dijkstra search that begins at the wire closest to the target. Specifically, the entire set of wires that share a table entry constitutes a "super" source node for the Dijkstra search. In this manner, we ensure that the cost-to-target estimate for a given sink terminal is the cost of a shortest path from the wire that is closest to the sink terminal. From this point on, we will refer to this method for calculating cost-to-target estimates as the *superDijkstra* method.

The important question now is how to identify wires that should share a table entry. Clearly, we would like to identify clusters of wires that have similar cost-to-target estimates, so that we can collect them together in a set that points to a single entry in the cost-to-target lookup table. Our first technique for clustering wires together is inspired by two observations:

- The number of logic units in an FPGA is generally much less than the number of interconnect wires.
- Logic units and interconnect wires are often interspersed in the FPGA fabric in a regular fashion.

Based on these observations, our first technique uses a proximity metric (described in the Section 4) to associate each wire with a logic unit. After each interconnect wire has been associated with a logic unit, all wires associated with the same logic unit are assigned to the same cluster. The cost-to-target estimates for each cluster are calculated using the *superDijkstra* method and stored in a lookup table. Since the number of table entries is equal to the number of logic units, the memory requirements of this technique are significantly less than a lookup table that has a separate entry for each wire in the interconnect structure.

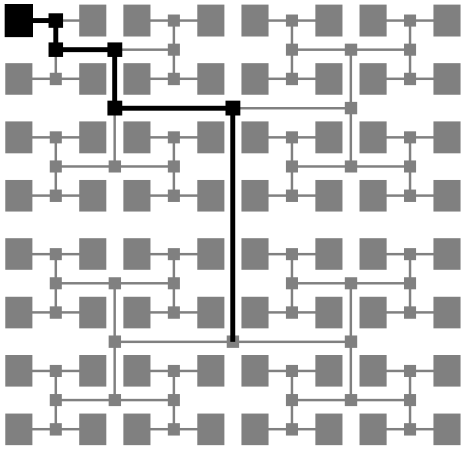

Figure 2: An example of a tree-based, hierarchical interconnect structure. Assume that the wires shown in black belong to the same cluster.

The associate-with-closest-logic-unit technique is probably well suited to island-style FPGAs. Since the logic and interconnect structures of an island-style FPGA are closely coupled, this approach may produce clusters of wires that have reasonably similar cost-to-target estimates. On hierarchical structures, the accuracy of an associate-with-closest-logic-unit approach may not be quite as good. For example, consider the tree-like interconnect structure in Figure 2. The routing wire that is topmost in the interconnect hierarchy is equally close to all logic units, while the wires in the next level are equally close to half the logic units, and so on. Associating wires with individual logic units in a strictly hierarchical interconnect structure may result in large cost-to-target underestimates.

In Figure 2, assume that the wires shown in black are associated with the black logic unit, and that the cost-to-target estimates for the cluster have been calculated using the *superDijkstra* method. The wire that directly connects to the black logic unit will have a cost-to-target estimate of five for the logic units in the northeast, southeast and southwest quadrants of the architecture. Note that the actual cost is nine wires for the northeast quadrant, and ten for the southeast and southwest quadrants. Estimates that are a factor of two below exact might slow down the router considerably. However, every wire in the cluster shown in Figure 2 does not suffer from the same problem. The cluster wire that is topmost in the interconnect hierarchy (black vertical line down the middle of Figure 2) will have exact cost-to-target estimates for all logic units in the northeast, southeast and southwest quadrants, and underestimates for logic units in the northwest quadrant.

To summarize, one would expect the associate-with-closest-logic-unit approach to work well for island-style structures. However, due to the approach's potential limitations on hierarchical structures, we feel that a more sophisticated technique might be necessary to produce reasonably accurate cost-to-target estimates across different interconnect styles.

4. K-means clustering

Our second technique for architecture adaptive clustering problem is to use the K-means algorithm, guided by each resource's cost-to-target estimates. K-means clustering is an iterative heuristic that is used to divide a dataset into K non-overlapping clusters based on a proximity metric. Pseudocode for the K-Means algorithm appears in Figure 3.

```
// D is the set of data-points in n-dimensional space that has to be divided into K clusters.
// The co-ordinates of a data-point d_i ∈ D are contained in the vector d_i.vec.
// d_i.vec is an n-dimensional vector.

K-Means {
   for i in 1…K {
      randomly select a data-point d_i from the set D.
      initialize the centroid of cluster clus_i to d_i.vec.
   }

   while (terminating condition not met) {
      for each d_i ∈ D {
        remove d_i's cluster assignment.
      }

      for each d_i ∈ D {
        for j in 1…K {
           diff_ij = vectorDifference (d_i.vec,clus_j.centroid)
        }
        assign d_i to the cluster clus_y such that diff_iy is
        minimum.
      }

      for j in 1…K {
        recalculate clus_j.centroid using the data-points
        currently assigned to clus_j.
      }
   }

}
```

Figure 3: Pseudocode for the K-Means clustering algorithm.

We now briefly describe our choices for the alparameters that characterize the K-Means algorithm.

Dataset (D): The dataset D simply consists of all the routing wires in the interconnect structure of the target device.

Number of Clusters (K): We experimentally determined that a value of K greater than or equal to the number of logic units in the target device is a reasonable choice. Section 5 describes the effect of K on the quality of clustering solutions.

Initial Seed Selection: The initial seeds consist of K/2 randomly selected logic-block output wires and K/2 randomly selected routing wires.

Terminating Condition: The K-Means algorithm is terminated when less than 1% of the dataset changed clusters during the previous clustering iteration.

Calculating Cost-to-Target Estimates: On completion of the clustering algorithm, the actual A* estimates for a cluster are calculated using the *superDijkstra* method.

Co-ordinate Space and Proximity Metric: The most important consideration in applying the K-Means algorithm to solve the interconnect clustering problem is the proximity metric. Specifically, we need to determine a co-ordinate space that is representative of the A* cost-to-target estimate at each wire in the dataset. In our implementation, the co-ordinates of a routing wire represent the cost of the shortest path to a randomly chosen subset S of the sink terminals in the interconnect structure. The co-ordinates of each routing wire are pre-calculated using Dijkstra's algorithm and stored in a table.

If the number of sink terminals in S is n, then the co-ordinates of a routing wire $d_i \in$ D are represented by an n-dimensional vector $d_i.vec$. Each entry c_{ij} ($j \in$ 1…n) in the vector $d_i.vec$ is the cost of a shortest path from the routing wire d_i to the sink terminal j. The co-ordinates for all $d_i \in$ D are calculated by launching individual Dijkstra searches from each sink terminal in the set S. Note that the edges in the underlying routing graph are reversed to enable Dijkstra searches that originate at sink terminals. At the end of a Dijkstra search that is launched at sink terminal j, the cost of a shortest path from every d_i to the terminal j is written into the corresponding c_{ij} entry of $di.vec$. The vector $d_i.vec$ is used by the K-Means algorithm to calculate the "distance" between the wire d_i and the centroid of each cluster. The distance between d_i and a cluster centroid is defined as the magnitude of the vector difference between $d_i.vec$ and the cluster centroid.

Note that the size of S directly influences the memory requirements of our clustering implementation. In the extreme case where S contains every sink terminal in the target device, the memory requirements would match the prohibitively large requirements of a table that stores the cost of a shortest path from each routing wire to every sink terminal. This would undermine the purpose of using a clustering algorithm to reduce the memory requirements of an A* estimate table. It is thus useful to sub-sample the number of sink terminals in the target device when setting up the set S.

Table 1: Comparison of memory requirements. Table sizes are in GB.

		Pathfinder	Clustering	
Size	ChanWidth	$\|S\| = N_T$	$\|S\| = 0.06*N_T$	Estimates
10x10	10	0.0012	0.0001	0.0001
20x20	10	0.0151	0.0009	0.0007
30x30	10	0.0707	0.0043	0.0035
40x40	10	0.2152	0.0130	0.0106
50x50	10	0.5132	0.0310	0.0253
60x60	10	1.0474	0.0631	0.0518
70x70	10	1.9185	0.1155	0.0949
80x80	10	3.2449	0.1951	0.1607
90x90	10	5.1629	0.3103	0.2559
100x100	10	7.8268	0.4703	0.3882
110x110	10	11.4087	0.6854	0.5662
120x120	10	16.0986	0.9669	0.7994
130x130	10	22.1044	1.3275	1.0980
140x140	10	29.6517	1.7805	1.4735
150x150	10	38.9842	2.3406	1.9380
160x160	10	50.3636	3.0236	2.5045
170x170	10	64.0690	3.8462	3.1869
180x180	10	80.3979	4.8262	4.0001
190x190	10	99.6654	5.9825	4.9599
200x200	10	122.2044	7.3351	6.0828

Table 1 compares the memory requirements of a clustering-based implementation that sub-samples the sink terminals with a table that stores the cost of a shortest path from each routing wire to every sink terminal in the target device. The target architecture is assumed to be a square island-style array that has only single-length wire segments. In our calculations, we assume that the sizes of a floating-point number, integer number, and a pointer are all four bytes. Column 1 lists the size of the target array, and column 2 lists the channel width of the target array. Let the total number of sink terminals in the target array be N_T. Column 3 lists the memory requirements of a table that stores the cost of a shortest path from each wire to every sink terminal in the target device (i.e. $|S| = N_T$). This corresponds to the exhaustive lookup table approach proposed by the creators of the Pathfinder algorithm in [8]. Column 4 lists the size of a table that stores costs to only 6% of the sink terminals ($|S| = 0.06*N_T$), and column 5 lists the size of a table that holds cost-to-target estimates for the clusters produced by a K-Means implementation where K = number of logic units in the target device. All memory requirements are reported in Gigabyte. It is clear from Table 1 that our K-Means clustering approach avoids the impractical memory requirements of a table that stores costs to every sink terminal in the target device.

Finally, note that the clustering process is a one-time preprocessing step that needs to be performed only on a per-architecture basis. The table of cost-to-target estimates produced by the clustering algorithm can be reused every time a new netlist is routed, and there is no additional runtime or memory cost incurred by our techniques on a per-netlist basis.

5. Results

We conduct three experiments to test the validity of using the K-Means algorithm to cluster the interconnect structure of an FPGA. The first experiment studies the effect of sub-sampling the sink terminals in the target device on the quality of clustering solutions. The second experiment studies the effect of the number of clusters (K) on quality, and the third experiment compares the quality of clustering-based A* estimates with heuristically calculated estimates. To evaluate the adaptability of our techniques, we conduct the experiments on an island-style interconnect architecture and HSRA [6]. Details of the architectural parameters used in our experiments can be found in [10].

Since the truest measure of the quality of an A* estimate is routing runtime, our quality metric is defined to be the CPU runtime per routing iteration when routing a placement on the target device. The placements for our experiments on island-style structures are obtained using VPR [1], and the placements for our experiments on HSRA are produced using Independence [10].

Finally, note that our clustering techniques are guaranteed to produce conservative cost-to-target estimates, and hence these techniques have no effect on routing quality.

5.1. Experiment 1 – Sub-sampling Sinks

Experiment 1 studies the effect of sub-sampling the number of sink terminals in the target device. The set of benchmark netlists used in this experiment is a subset of the netlists shown in Table 2 (island-style) and Table 3 (HSRA).

Figure 4 shows the variation in quality of clustering solutions. The x-axis represents the fraction of sink terminals that are used to represent the co-ordinates of each wire during clustering. The subset of sink terminals used in the experiment is randomly generated. The y-axis represents routing runtime measured in seconds per routing iteration. The curves show the variation in routing runtimes when using A* estimates produced by the K-Means clustering technique. The flat line shows the routing runtime when using architecture-specific heuristic A* estimates. The value of K in this experiment is equal to the number of logic units in the target device.

Figure 4 shows that using as little as 5% of the sink terminals during clustering may be sufficient to produce estimates that are comparable to heuristic estimates. This is not a surprising result. Due to the regularity of an FPGA's interconnect structure, a small subset of sink terminals may be sufficient in resolving the interconnect wires into reasonably formed clusters. Note that 5% of the sink terminals represents a variable number of sink terminals across the set of benchmark netlists. Depending on the size of the netlist, 5% of the sink terminals could be anywhere between two and fifty sink terminals.

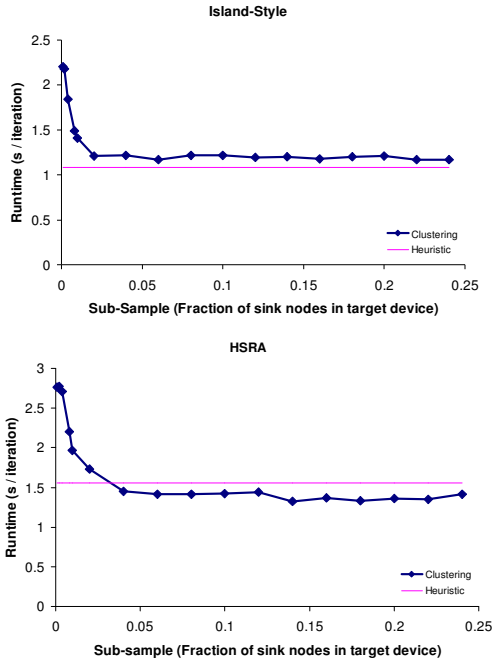

Figure 4: The effect of sub-sampling the number of sink terminals on routing runtime.

In Figure 5, we present the results of a second study that evaluates the quality of clustering solutions when using a small, fixed number of sink terminals. Figure 5 shows that using a small number (say 16) of randomly selected sink nodes may be enough to produce clustering solutions that are within approximately 15% of heuristic estimates.

5.2. Experiment 2 – Number of Clusters (K)

Experiment 2 studies the effect of the number of clusters (K) on the quality of clustering solutions. The set of benchmark netlists used in this experiment is identical to the set used in *Experiment 1*. We use a sub-sample of 6% for island-style architectures, and 14% for HSRA.

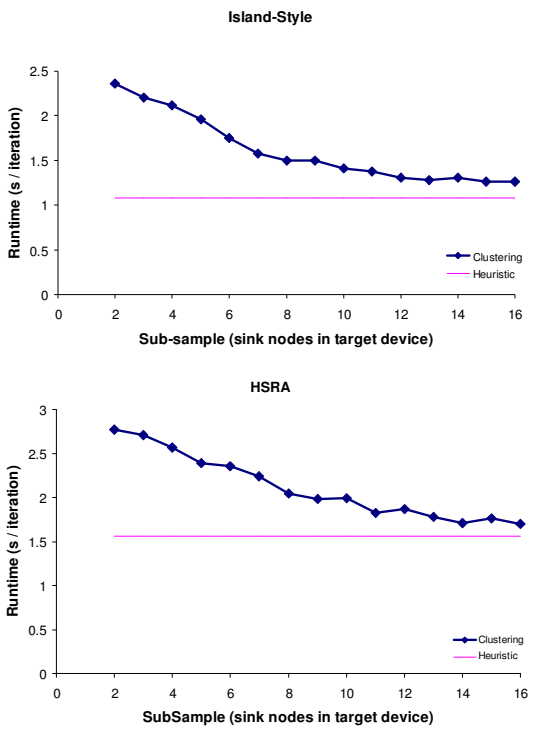

Figure 5: Using a small number of sink nodes may produce clustering solutions of acceptable quality.

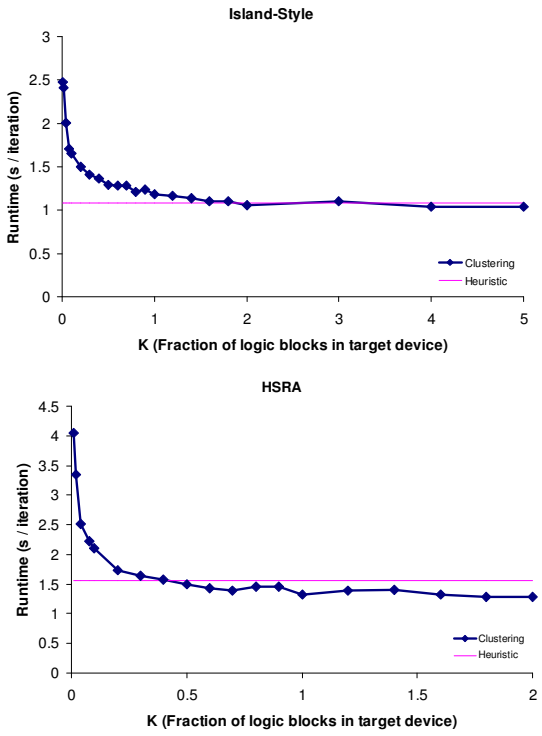

Figure 6: The effect of K on routing runtime.

Figure 6 shows the effect of K on routing runtime. The x-axis shows the value of K as a fraction of the number of logic units in the target device, and the y-axis shows routing runtime in seconds per routing iteration. The charts in Figure 6 show that a value of K equal to or greater than the number of logic units in the target device produces clustering solutions of qualities similar (within 10%) to heuristic estimates.

5.3. Quantitative Comparisons

Experiment 3 is a quantitative comparison of the quality of the A* estimates produced by our clustering techniques vs. heuristically calculated estimates. We use the following settings in this experiment:

- Associate-with-closest-logic-unit technique. This technique is implemented by running only the first iteration of K-Means clustering. K is chosen to be equal to the number of logic units in the target device ($K = N_L$), and initial seeds chosen to be logic unit outputs. The value of sink sub-sample is 6% ($|S| = 0.06*N_T$). These settings represent a relatively low-effort clustering step. This step might be undertaken when clustering runtime and memory requirements need to be very low.
- K-Means clustering, with a sink sub-sample value of 6% ($|S| = 0.06*N_T$) and K equal to the number of logic units in the target device ($K = N_L$). N_T is the total number of sink terminals in the target device, and N_L is the total number of logic units in the target device. These settings represent an empirically determined sweet-spot for our K-Means clustering technique.
- K-Means clustering, with a sink sub-sample value of 20% ($|S| = 0.2*N_T$) and K equal to twice the number of logic units in the target device ($K = 2*N_L$). These are aggressive settings that represent potentially high quality clustering solutions. Such settings may be used when absolutely the best quality clustering solutions are required, and clustering runtime and memory are of less concern.

Table 2 shows the results we obtained on the island-style architecture. Column 1 lists the netlist, column 2 lists the size of the smallest square array needed to just fit the netlist, and column 3 lists routing runtimes obtained on using heuristic estimates. Columns 4, 5, and 6 list routing runtimes and compression ratios (shown in brackets) produced by the low-effort associate-with-logic-unit technique, K-Means clustering at empirically determined settings ($|S| = 0.06*N_T$, $K = N_L$), and K-Means clustering at high-quality settings ($|S| = 0.20*N_T$, $K = 2*N_L$) respectively. Routing runtimes are normalized to runtimes produced by heuristic estimates. The compression ratio is defined as the ratio between the size of an exhaustive lookup table and a lookup table that holds cost-to-target estimates for the clusters

produced by each of the three techniques. The compression ratio is a measure of the memory gap between a version of Pathfinder that uses an exhaustive lookup table and a version that uses cost-to-target estimates produced by our clustering techniques. Column 7 shows routing runtimes produced by an undirected (no A*) search technique.

Across the set of benchmarks, the runtimes produced by our K-Means clustering techniques are approximately 7% (high-quality settings) and 11% (empirical settings) slower than the runtimes achieved by heuristically estimating A* costs. Both heuristic and clustering-based estimates are approximately 6X faster than an undirected search-based router. Finally, the routing runtimes produced by the associate-with-closest-logic-unit technique is within 5% of the runtimes produced by either of the K-Means clustering techniques. The near identical runtimes show that the associate-with-closest-logic-unit approach presented in Section 3 works as well as a more sophisticated clustering approach on an island-style architecture. The geometric mean of the compression ratios is 30:1 for the associate-with-closest-logic-unit approach and K-Means clustering at empirical settings. The ratio goes down to 18:1 for the higher-quality settings. This is to be expected, since we use double the number of starting clusters ($K = 2*N_L$) at the higher-quality settings.

Table 2: A comparison of routing runtimes on an island-style architecture.

			$\|S\| = 0.06*N_T$		$\|S\| = 0.20*N_T$	
Netlist	Size	Heur	Associate	K-Means ($K = N_L$)	K-Means ($K = 2*N_L$)	no A*
term1	6x6	1.00	0.89 (17:1)	1.44 (17:1)	1.22 (10:1)	4.22
s1423	6x6	1.00	1.57 (20:1)	1.57 (18:1)	1.14 (10:1)	3.86
i9	7x7	1.00	1.30 (17:1)	1.30 (17:1)	1.10 (10:1)	3.40
dalu	8x8	1.00	0.93 (24:1)	0.93 (22:1)	1.15 (13:1)	4.04
vda	9x9	1.00	1.20 (29:1)	1.08 (32:1)	1.08 (16:1)	4.78
x1	10x10	1.00	1.13 (20:1)	0.94 (19:1)	1.17 (11:1)	4.66
rot	8x8	1.00	0.95 (26:1)	1.11 (25:1)	0.89 (14:1)	3.32
pair	9x9	1.00	0.89 (30:1)	0.94 (36:1)	0.94 (18:1)	4.83
apex1	11x11	1.00	0.97 (40:1)	0.96 (37:1)	1.00 (23:1)	6.03
dsip	14x14	1.00	1.13 (22:1)	1.06 (23:1)	1.07 (13:1)	8.21
ex5p	12x12	1.00	1.03 (48:1)	1.12 (48:1)	1.05 (29:1)	7.30
s298	16x16	1.00	1.58 (25:1)	1.37 (23:1)	1.36 (14:1)	10.38
tseng	12x12	1.00	1.05 (27:1)	1.07 (29:1)	1.04 (17:1)	6.30
alu4	14x14	1.00	1.09 (30:1)	1.14 (30:1)	1.14 (19:1)	7.48
misex3	14x14	1.00	1.16 (40:1)	1.08 (41:1)	1.05 (23:1)	9.80
apex4	13x13	1.00	1.10 (46:1)	1.02 (45:1)	1.07 (27:1)	5.04
diffeq	14x14	1.00	1.19 (26:1)	1.13 (26:1)	1.08 (15:1)	5.29
bigkey	15x15	1.00	1.38 (26:1)	1.18 (26:1)	1.08 (16:1)	8.95
seq	15x15	1.00	1.19 (37:1)	1.10 (39:1)	1.05 (23:1)	7.22
des	15x15	1.00	1.20 (29:1)	1.17 (29:1)	1.05 (18:1)	4.35
apex2	16x16	1.00	1.08 (43:1)	1.09 (42:1)	1.04 (26:1)	8.19
frisc	22x22	1.00	1.08 (41:1)	1.02 (41:1)	1.06 (25:1)	8.56
elliptic	22x22	1.00	1.23 (41:1)	1.00 (40:1)	1.05 (24:1)	10.73
ex1010	25x25	1.00	0.92 (48:1)	1.15 (47:1)	1.07 (29:1)	9.66
s38584.1	29x29	1.00	1.07 (31:1)	1.20 (31:1)	1.07 (18:1)	17.07
clma	33x33	1.00	1.03 (48:1)	1.02 (48:1)	1.00 (29:1)	15.25
GEOMEAN		**1.00**	**1.12 (30:1)**	**1.11 (30:1)**	**1.07 (18:1)**	**6.59**

Table 3 shows the results that we obtained on HSRA. With the exception of column 2, the settings and columns are identical to Table 2. In this case, column 2 lists the number of logic units in the target device. Across the set of benchmarks, the runtimes produced by our clustering-based techniques are approximately 9% (higher-quality) and 7% (empirical settings) faster than the runtimes achieved by heuristically estimating A* costs. Both heuristic and clustering-based techniques are approximately ten times faster than an undirected search-based router. The runtimes produced by the associate-with-closest-logic-unit technique are approximately 16% slower than K-Means clustering at empirical settings, and 20% slower than higher-quality K-Means clustering. This is consistent with our intuition that associating interconnect wires with logic units in a hierarchical structure (Figure 2) will probably produce cost-to-target underestimates.

Table 3: A comparison of routing runtimes on HSRA.

			$\|S\| = 0.06*N_T$		$\|S\| = 0.20*N_T$	
Netlist	Size	Heur	Associate	K-Means ($K = N_L$)	K-Means ($K = 2*N_L$)	no A*
mm9b	256	1.00	1.48 (149:1)	1.16 (85:1)	1.29 (35:1)	3.87
Cse	256	1.00	1.22 (149:1)	1.03 (85:1)	1.06 (35:1)	4.39
s1423	256	1.00	1.00 (149:1)	0.92 (85:1)	0.85 (35:1)	5.23
9sym	512	1.00	1.20 (135:1)	0.81 (83:1)	0.69 (36:1)	15.42
ttt2	256	1.00	1.25 (149:1)	1.06 (85:1)	1.14 (35:1)	13.58
keyb	256	1.00	1.16 (149:1)	1.16 (85:1)	1.01 (35:1)	4.25
clip	512	1.00	1.14 (135:1)	1.02 (83:1)	1.01 (36:1)	21.38
term1	512	1.00	1.11 (150:1)	0.83 (95:1)	0.74 (39:1)	19.56
apex6	1024	1.00	1.26 (128:1)	1.24 (80:1)	1.34 (35:1)	6.53
vg2	512	1.00	1.16 (135:1)	0.96 (83:1)	0.95 (36:1)	16.81
frg1	1024	1.00	0.85 (142:1)	0.81 (88:1)	0.63 (39:1)	26.73
sbc	1024	1.00	1.13 (142:1)	0.87 (88:1)	0.87 (39:1)	12.41
styr	1024	1.00	1.06 (128:1)	0.83 (80:1)	0.74 (35:1)	13.60
i9	512	1.00	1.32 (150:1)	1.01 (95:1)	0.96 (39:1)	12.12
C3540	1024	1.00	0.79 (128:1)	0.79 (80:1)	0.72 (35:1)	5.89
sand	1024	1.00	0.88 (142:1)	0.80 (88:1)	0.81 (39:1)	10.67
x3	1024	1.00	0.88 (142:1)	0.80 (88:1)	0.85 (39:1)	3.60
planet	2048	1.00	1.14 (135:1)	0.81 (81:1)	0.89 (39:1)	13.67
rd84	2048	1.00	1.08 (135:1)	1.09 (81:1)	1.13 (39:1)	21.04
dalu	2048	1.00	0.84 (135:1)	0.82 (81:1)	0.89 (39:1)	16.62
GEOMEAN		**1.00**	**1.08 (140:1)**	**0.93 (85:1)**	**0.91 (37:1)**	**10.39**

There is a large gap in compression ratio between the associate-with-closest-logic-unit approach and K-Means clustering at empirical settings. In the associate-with-closest-logic-unit approach, each logic unit in the target device is an initial seed. At low sub-sampling values, a routing wire may be equidistant to several different logic units. Since a routing-wire must eventually be associated with a single logic unit, chances are that a number of logic-units at the end of the clustering process do not have any routing wires associated with them. These logic-unit seeds are eliminated and the number of final clusters is significantly less than the number of starting seeds. This effect is mitigated in K-Means clustering because the initial seeds are a mix of logic units and randomly selected routing wires. Thus, relatively few clusters are eliminated and the compression ratio is lower than the associate-with-logic-unit approach.

6. Conclusions

Our goal in this paper was the development of architecture-adaptive A* search techniques that can be used to speed up the Pathfinder algorithm. The

clustering-based techniques presented in this paper do not rely on architecture-specific heuristics to calculate cost-to-target estimates. This is in direct contrast to previously published techniques [11,12] that explicitly rely on Manhattan distance routability estimates, making them applicable only to island-style FPGAs. Our techniques should work on any FPGA that can be represented as a routing graph. The adaptability of our approach is demonstrated in *Experiment 3*; on an island-style architecture, the runtimes produced by the K-Means clustering approach are within 7% of heuristic estimates, and 11% better than heuristic estimates on HSRA. There are several potential benefits of using a clustering-based approach to calculate cost-to-target estimates:

Memory – During the routing process, the memory requirements of the lookup tables produced by our techniques are 18 – 30 times (island-style architecture) and 37 – 140 times (HSRA) less than the exhaustive lookup table proposed by the creators of the Pathfinder algorithm. Thus, our techniques offer adaptability, albeit at significantly smaller memory cost.

Cost of Calculation – The lookup tables produced by our techniques eliminate the task of calculating cost-to-target estimates on the fly during the routing process. Heuristically calculating estimates in the routing inner loop may be expensive when compared to the simple pointer dereferencing operations required to obtain estimates from a lookup table.

Usability Considerations - A production version of a truly architecture-adaptive Pathfinder implementation must be a stand-alone tool that requires minimal user intervention. An architecture-specific cost-to-target estimator may necessitate source code modifications and possible changes to the tool's interface on a per-architecture basis. We feel that users should not be expected to provide any architecture-specific enhancements to speed up Pathfinder. Our techniques do not require any per-architecture source-code changes, and interface with a routing tool through a cost-to-target lookup table.

Automatically Generated Architectures: During domain-specific reconfigurable architecture generation [3,4], the nature of the reconfigurable device's interconnect structure may be significantly different across application domains. If Pathfinder is used to route netlists on such architectures, then the cost-to-target estimator used by this flow must adapt to different interconnect structures. Expecting the user to modify the flow to produce cost-to-target estimates goes against the underlying philosophy of automatic architecture generation.

7. Acknowledgements

This work was supported by grants from the National Science Foundation, Altera Corporation, and the Alfred P Sloan Foundation.

8. References

[1] V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Boston MA, 1999.

[2] G. Boriello, C. Ebeling, S Hauck, and S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLS Systems Vol. 3 No. 4*, IEEE, New York NY, 1995, pp. 473 – 482.

[3] K. Compton, A. Sharma, S. Phillips, and S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field Programmable Logic and Applications*, Springer-Verlag, New York NY, 2002, pp. 59 – 68.

[4] K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", *IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, New York NY, 2001.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge MA, 1990.

[6] A. DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM Press, New York NY, 1999, pp. 69 – 78.

[7] A. Marquardt, V. Betz and J. Rose, "Speed and Area Tradeoffs in Cluster-Based FPGA Architectures", *IEEE Transactions on VLSI Systems Vol. 8 No. 1*, IEEE, New York NY, 2000, pp. 84 – 93.

[8] L. McMurchie, and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM Press, New York NY, 1995, pp 111-117.

[9] N. Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann, San Francisco CA, 1980.

[10] A Sharma, C. Ebeling, and S. Hauck, "Architecture-Adaptive Routability-Driven Placement for FPGAs", in *International Conference on Field Programmable Logic and Applications*, 2005.

[11] J. Swartz, V. Betz and J. Rose, "A Fast Routability-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ACM Press, New York NY, 1998, pp. 140 – 149.

[12] R. Tessier, "Negotiated A* Routing for FPGAs", *Fifth Canadian Workshop on Field Programmable Devices*, 1998.