Generating Highly-Routable Sparse Crossbars for PLDs

Guy Lemieux
Dept. of Elec. & Comp. Eng.
University of Toronto
Toronto, Canada
lemieux@eecg.toronto.edu

Paul Leventis*
Right Track CAD Corp.
313-720 Spadina Ave.
Toronto, Canada
paul@rtrack.com

David Lewis
Dept. of Elec. & Comp. Eng.
University of Toronto
Toronto, Canada
lewis@eecg.toronto.edu

ABSTRACT

A method for evaluating and constructing sparse crossbars which are both area efficient and highly routable is presented. The evaluation method uses a network flow algorithm to accurately compute the percentage of random test vectors that can be routed. The construction method attempts to maximize the spread of the switch locations, such that any given subset of input wires can connect to as many output wires as possible. Based on Hall's Theorem, we argue that this increases the likelihood of routing.

The hardest test vectors to route are those which attempt to use all of the crossbar outputs. Results in this paper show that area-efficient sparse crossbars can be constructed by providing more outputs than required and a sufficient number of switches. In a few specific case studies, it is shown that sparse crossbars with about 90% fewer switches than a full crossbar can be constructed, and these crossbars are capable of routing over 95% of randomly chosen routing vectors. In one case, a new switch matrix which can replace the one in the Altera FLEX8000 family is shown. This new switch matrix uses approximately 14% more transistors, yet can increase the routability of the most difficult test vectors from 1% to over 96%.

1. INTRODUCTION

Programmable logic devices commonly use full crossbars and sparse crossbars as building blocks in routing networks. Typically, a full crossbar is chosen when a highly-routable crossbar is desired, and a sparse crossbar containing significantly fewer crosspoints is selected when area use is most important. This naturally brings up the question, "Is it possible to get the best of both worlds?"

There are many instances where a highly routable crossbar would be preferred, but the area cost of a full crossbar is prohibitive. For example, the Plasma FPGA [5] in the Hewlett-Packard Teramac [4] reconfigurable logic system would have used full crossbars to guarantee routability. However, to save area, it was necessary to use only 1/4 of the the switches.

In Teramac and large-scale logic emulation systems, such as those by Quickturn [18], circuits are partitioned across a large number of FPGAs. Each of the generated subcircuits must successfully place-and-route in an FPGA, otherwise time-consuming re-partitioning and re-routing is required. To make routable subcircuits, one can intentionally underutilise the LUTs in the FPGA [9], or use an FPGA that is designed to be highly routable.

Highly routable components in a single FPGA can also benefit users by reducing compute time and memory use. The latest FPGAs by Altera and Xilinx have a large number of LUTs and wiring resources. To route these FPGAs, CAD tools usually store the following details in memory: a representation of the circuit, its mapping to FPGA resources, and a model of the entire FPGA. This leads to considerable memory use. For example, Altera recommends using 1GB of RAM to route designs for the APEX 20K1000E device [2]. It may be possible to make the CAD tools more efficient if they follow the Teramac and logic emulation system model: partition a circuit into smaller subcircuits, then place and route each piece independently. To do this effectively without rip-up and re-partitioning, there must be confidence that each subcircuit is very likely to route.

As another example, CPLDs are required to be highly routable because they are often close to 100% utilised. Full crossbars are not normally used in the global interconnect of CPLDs due to the area overhead involved, so an area-saving sparse pattern is required.

The above scenarios indicate that highly routable, sparsely populated crossbars would be useful, yet there is little published work in this area. In this paper, this issue is addressed by describing conditions for routability (Hall's Theorem), a method for evaluating routability without resorting to place-and-route experiments, and a construction algorithm that achieves good performance. Results for a few design cases are shown to exemplify the area requirements and routability obtainable from sparse crossbars.

2. CROSSBAR TYPES AND PROPERTIES

An $n \times m$ crossbar connects n different input wires to m output wires, typically with $n \geq m$. An example of a few crossbars are shown in Figure 1. At the locations where an input crosses an output wire, a programmable switch, or *crosspoint*, may be present. We use the term *capacity* of a crossbar to mean the number of signals being routed through it. The term *population* refers to the number of switches in the crossbar, p.

2.1 Full Crossbars

A *fully-populated crossbar* or *full crossbar* contains switches at every intersection point of the input and output wires, using a total of $p = n \cdot m$ switches. The term *crossbar* usually refers to a full crossbar. An example of a full crossbar is shown on the left in Figure 1.

Full crossbars are extremely flexible because they can connect *any* wire on the input side to connect to *any* wire on the output side, *i.e.* they support any permutation of the outputs. Additionally, full

*This work was performed at the University of Toronto.

FPGA 2000 Monterey, California USA

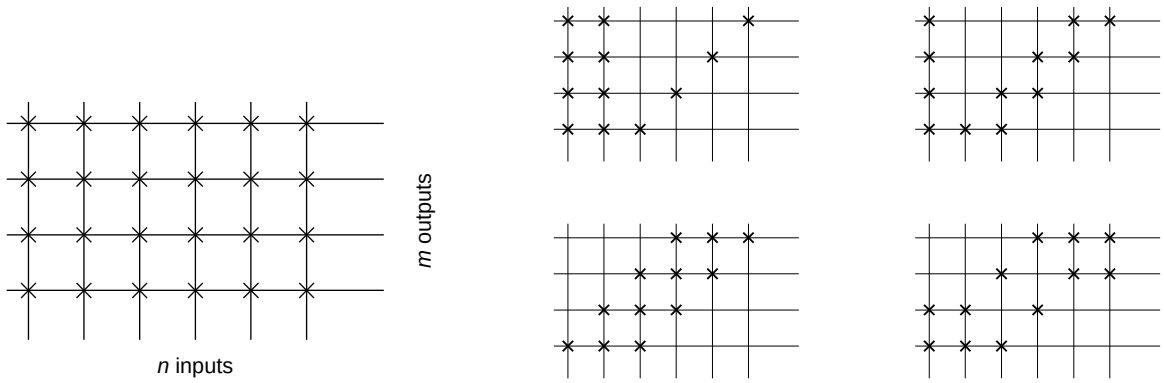

Figure 1: Examples of 6×4 crossbars: a full crossbar on the left, full-capacity minimal crossbars on the right.

crossbars can be used at *full capacity*: they can connect as many signals as the number of outputs in the crossbar.

2.2 Full-Capacity Minimal Crossbars

Full-capacity minimal crossbars are well-known constructions that use fewer switches than a full crossbar. They are slightly less flexible than full crossbars, but they retain the full-capacity property. For convenience, we shall refer to them simply as *minimal crossbars*.

Minimal crossbars are less flexible than full crossbars because they remove the freedom to assign a specific input wire to a specific output wire. Thus, any m input wires can be connected to all m output wires, but the ordering of the signals on the output wires may not be freely chosen.

A minimal crossbar always uses $p = (n-m+1) \cdot m$ switches. Nakamura [16] has shown that no switches can be removed from a minimal crossbar without also removing the full-capacity property. Minimal crossbars do not save many switches when $n \gg m$, but the number of switches is reduced from a quadratic expression to a roughly linear one when $n \simeq m$.

There are many different topologies for minimal crossbars, a few of which are shown in Figure 1. The simplest topology, called a *fat-and-slim crossbar*, uses a full crossbar between the first $n-m$ input wires and all m output wires. Each of the remaining m input wires have only one switch and are connected to a different output wire. This results in balanced *fan-in* for the output wires, but largely unbalanced *fan-out* on the inputs.

Some minimal crossbar topologies simultaneously balance the number of switches on the input and output wires. Fujiyoshi [11] defines a class of minimal crossbars called *bi-scattered* which have naturally balanced fan-in. They also provide a switch placement algorithm to generate bi-scattered crossbars with balanced fan-out. Guo [12] suggests a transformation that redistributes switches of a fat-and-slim crossbar, yet preserves full-capacity and the already-balanced fan-in arrangement. They prove this transformation can be used to obtain balanced or nearly-balanced (within ± 1) fan-outs.

2.3 Perfect Crossbars

Since both full and minimal crossbars support full-capacity, it is convenient to refer to them as *perfect* crossbars. Perfect crossbars are one way to implement an *(n, m)-concentrator*, a type of graph that can disjointly route any m-sized subset of the n inputs to m outputs.

2.4 Sparse Crossbars

A *sparse crossbar* refers to a crossbar which has few switches, *i.e.* is sparsely populated. The demarcation point of when a crossbar becomes "sparse" is debatable: for example, nearly square crossbars can be sparsely populated yet support full capacity. This paper assumes that a crossbar is sparse if it contains fewer than $p < (n-m+1) \cdot m$ switches. Hence, no matter how well it is designed, a sparse crossbar can never be made perfect.

Oruç [17] has proven that a sparse crossbar of guaranteed capacity c, where $c \leq m$, must contain $p \geq \lceil m \cdot (n-m+1)/(m-c+1) \rceil$ switches. This lower bound is not necessarily tight, but when $c = m$, the number of switches in a minimal crossbar is obtained. Sparse crossbars with guaranteed capacity c are also referred to as *(n, m, c)-concentrators*.

The transformation suggested by Guo [12] can also be applied to sparse crossbars, provided that there exists some input which covers all of the outputs reachable by another input. The transformation states that switches can be moved from the one input to the other, causing the guaranteed routing capacity of the crossbar, c to increase or stay the same — but it will never decrease. This is a beneficial transformation, but it is not often applicable in sparse crossbars because it is uncommon to have one input completely cover another.

2.5 Graph Representation

Crossbars are easily modeled as a graph when wires are represented by nodes and switches are represented by edges. A crossbar forms a bipartite graph G composed of two sets of nodes and a set of edges. The node sets are a set of input wires I and a set of output wires O. There are no edges within each set, but an edge can exist between any node in set I and any node in set O.

3. EVALUATING ROUTABILITY

The traditional approach to evaluate the routability of an FPGA, and hence evaluate the sparse crossbars contained therein, is to run place and route experiments with a suite of benchmark circuits. This is an effective method to design an FPGA and its CAD tools in concert, but it can be a lengthy process. As well, the routing performance of the crossbars in the FPGA relies upon the effectiveness of the CAD tools and the benchmarks to exercise the architecture.

Our goal was to find a quicker way to test the routability of sparse crossbars independently of the CAD tool or benchmark circuits used. As well, this new method should provide a more sensitive, yet still practical, measurement of routability. This approach also helps avoid the problem of "training" an FPGA architecture or CAD tool to a particular benchmark suite.

One routability metric considered was the maximum guaranteed capacity of a crossbar, c. With this metric, we wish to find the largest value c such that any subset $I' \subseteq I$ of size $|I'| \leq c$ is guaranteed to be routable. The main problem with this metric is that it is very difficult to compute: the algorithm has inherently exponential complexity because it must examine all subsets of I with cardinality c or smaller. We implemented a branch-and-bound algorithm to search for this

value, but it is impractical for large crossbars. A greedy heuristic search was also implemented, but the results were not robust.

Instead, the routability of a crossbar is measured using a Monte Carlo test. For this test, a number of random test vectors are generated, and each is routed on the crossbar using a network flow algorithm. The routability of the crossbar is estimated as the percentage of test vectors which can be successfully routed.

A test vector of size k is the number of signals to be routed through the crossbar. More specifically, it is a subset of the input wires, $I' \subset I$, where $|I'| = k$. In terms of real FPGA routing, this test vector represents the case where logic signals have already been assigned to specific wires due to previous routing restrictions.

A highly-routable crossbar must be able to route many of these pre-constrained vectors. We evaluate routability as a function of the number of signals in a test vector. This distinguishes the easily routed vectors, *i.e.*, when k is small, from the difficult ones.

In this paper, we arbitrarily define the *highly-routable* point as being able to route at least 95% of the hardest test vectors, *i.e.*, those containing the maximum number of signals intended to be carried by the crossbar.

A network flow algorithm [7] is used to route the test vectors because it is *guaranteed* to find a routing solution if one exists. When routing a test vector of size k, switches are assigned unit capacity, so a flow of size k must be found to produce a solution. If a lower flow value is found, it indicates the largest number of wires that were actually routable. This guarantee of finding a solution is important in that it represents an ideal routing tool, hence it isolates the effectiveness of the CAD tools from the performance of the crossbar.

4. ROUTABLE SWITCH PATTERNS

This section looks at the following basic problem: given p switches, how should they be placed in a sparse crossbar to make it as routable as possible. The foundations for the switch-placement algorithm presented in the next section are based on the following theorem and observations.

4.1 Hall's Theorem

Hall's Theorem [13] is a result that can be applied to bipartite graphs defining whether a *maximum matching* can be found. A matching is a subset of the edges in the graph such that no two edges share a node. Hence, every pair of edges in a matching involve 4 distinct nodes. Hall's Theorem gives the precise conditions under which a matching can exist.

Hall's Theorem. Given bipartite graph G composed of a set of edges, E, and two independent sets of nodes, X and Y, then G has a matching of X into Y if and only if

$$\forall S \subseteq X, |S| \leq |N(S)|$$

where $|S|$ denotes the cardinality of subset S, and $N(S)$ is the set of neighbours of S in Y.

4.2 Application of Hall's Theorem

In terms of sparse crossbars, a matching actually forms a routing solution of a sparse crossbar. The Y set represents the output wire set O, and X is a specific test vector of the input wires $X = I' \subseteq I$. A test vector is routable if and only if Hall's condition is satisfied, and the matching gives the solution. The edges in the matching are the switches which must be turned on to form the connections.

To design a routable sparse crossbar, switches should be placed so that Hall's condition is satisfied for as many test vectors as possible. For test vectors of size k, it is a necessary condition that at least k distinct output wires are reachable by switches.

The switch placement algorithm described in the next section assumes that the switches placed on any specific subset of input wires should be spread out to as many output wires as possible. This is equivalent to making the neighbour set $N(S)$ as large as possible so that Hall's condition is satisfied.

Switch placement is not trivial because the switch pattern chosen for one subset of input wires may consequently make the pattern for some other subset too close. We argue that this also implies that each input wire, having equal likelihood of being a part of any particular subset, should have an equal number of switches. If one input has fewer switches, it would not be able to "spread out" to as many different neighbours. As a result, subsets which included this input may be less routable. To get around this, it should be given more switches so the fan-outs of the input wires are roughly equal. A similar argument implies that the fan-ins of the output wires should also be balanced. For this reason, the switch matrices constructed in this paper all have balanced fan-in and fan-out.

4.3 Hamming Distance and Coding Theory

The switch placement problem requires that subsets of the input wires span as many output wires as possible. Doing this for every possible subset of input wires is a difficult task, so we chose to approximate this by spreading out the switches for every pair of input wires. In this form, the switch placement problem becomes identical to the problem designing communication codes so that code-design techniques such as those from [14] can be used.

The location where switches are placed on an input wire can be represented by a bitvector of length m, where a 1 in the bitvector indicates that a switch is present. There are n such bitvectors, one for each input, forming the codewords of a binary code.

The number of neighbours of an input wire subset is the number of ones in the bitwise-OR of their bitvectors. Given two bitvectors, bv_1 and bv_2, the increase in the number of neighbours (output wires) reached by the combination of the two is related to the Hamming distance[1] between them, $d(bv_1, bv_2)$. Spreading out the switches between a pair of input wires i and j is the same as maximizing $d(bv_i, bv_j)$. Code design techniques attempt to maximize the minimum d between all of the codewords.

5. SWITCH PLACEMENT ALGORITHM

In our construction algorithm, the switch pattern is determined in two stages: first an initial switch pattern is chosen, then that pattern is iteratively optimized. The minimum inputs required are the matrix size, $n \times m$, and the number of switches p.

5.1 Initial Switch Pattern

The goal of the initial switch pattern is to place switches so they will obey given fan-in and fan-out specifications. These specifications form a limit on the number of switches that will be placed on each wire. The user may provide any valid fan-in and/or fan-out distribution, or, if no specification is provided, a balanced one is automatically generated based on p, n and m.

A switch pattern which obeys the fan-in/out specifications is generated in one of two ways: either randomly, or by network flows. The random method generates random locations in the crossbar and places a switch there if it won't violate the fan-in/out specifications. If, after a certain number of tries, it cannot find a valid location for the next switch, it erases all switches and starts over. Usually an initial pattern is found the first time, unless there are a large number of switches to place. If it still fails after restarting a number times, the tool falls back to the network flow method.

The network flow method temporarily places a switch at every location in the crossbar, and assigns each a unit capacity. The maximum flow from the input to output wires is found, using the fan-out and fan-in specifications as flow capacities for the wires. If an initial

[1]The Hamming distance is the number of bit positions that differ between the two bitvectors.

Figure 2: The switch matrix on the left has identical Hamming costs before and after the swap indicated. After the swap, it cannot route any subsets which include wires {1,5,6}. Hence, the cost function is not always effective at distinguishing good switch swaps. The switch matrix on the right has lower Hamming cost after the swap indicated, and routes all subsets of size 3. Before the swap it could not route subset {1,2,3}. In this case, the cost function can identify a good swap.

switch pattern can be generated to obey the given constraints, it will be found as solution with a total flow of p. The switches which the flow solver used are kept, and the other switches are discarded. The network flow method is used as a backup method because solving the flow network is usually slower than the random method.

5.2 Switch Placement Optimizer

The routability of an initial switch pattern can be improved by moving a number of switches to produce a more "spread out" pattern using a number of "switch swaps". A simulated-annealing approach was initially used [15], following the approach used in [10]. Since the overall goal was similar, we chose to minimize the same cost function:

$$\sum_{\forall i,j} \frac{1}{d(bv_i, bv_j)^2}.$$

In the process of designing many switch matrices, it was noticed that any hill-climbing moves which raised the cost function were nearly always found again and undone. Instead, an algorithm that follows the simple approach of accepting any swap that lowers the cost function proved to be just as effective and considerably faster. An algorithm that systematically evaluated all possible swap candidates was also tried, but that algorithm ran considerably slower.

The resulting improvement algorithm works in a greedy fashion: it generates random swap candidates, but it only accepts the swaps if the routability improves. The algorithm stops when it is unable to find any improvement in cost after checking a large set of swap candidates (about 10 000, for example).

5.3 Cost Function Pitfalls

Other cost functions have been considered. As noted in [10], the alternative of maximizing the minimum Hamming distance of the code is difficult because not all switch swaps would lead to an observable change in the cost function.

Another cost function would be to maximize the total Hamming distance between all pairs, *i.e.*,

$$\sum_{\forall i,j} d(bv_i, bv_j).$$

Unfortunately, this does not sufficiently penalize close bitvectors. For example, consider the 3 bitvectors 111000, 011100, 000111 with Hamming distances of 2, 4, and 6. The alternative switch topology 111000, 001110, 010011 gives distances of 4, 4, and 4 and has better routability. However, no difference between these bitvector sets is found if only the total Hamming distance is examined.

It would also be possible to run a Monte Carlo simulation and accept a swap only if routability improved. However, this leads to two problems. First, a Monte Carlo simulation would be much slower to compute. Second, the results of a single swap may not be readily discernible by the simulation.

In comparison to the above alternatives, the Hamming distance cost used is relatively quick to compute and it can distinguish most (but not all) changes to the switch pattern.

5.4 Generating Swap Candidates

Swap candidates are determined by the four intersection points of two input wires and two output wires. To preserve the fan-in/out distribution profiles, a swap operation must consist of two switches and two empty locations positioned diagonally on the intersection points.

To generate a swap candidate, two input wires are chosen at random. Given the placement of switches on these two wires, two output wires are randomly selected chosen to form a swap candidate. If no valid candidate exists, a new pair of input wires is chosen.

The fan-in/out distribution profiles can also be preserved while moving a single switch, provided the following conditions are met. A switch can be moved to another output wire (along the same input wire) if the original output wire fan-in is one greater than the new output wire fan-in before the move. Similarly, a switch can be moved along an output wire provided the fan-outs of the old and new input wire locations differ by one. Improvements arising from these single swaps are done exhaustively after the greedy algorithm gives up on swapping switch pairs.

5.5 Limitations of the Algorithm

When $p \leq 2n$, the switch matrix is very sparse and we have examples of suboptimal performance by our algorithm. Under these conditions, small disconnected components may be present in the bipartite graph, yet they are indistinguishable by the cost function. For example, consider the leftmost matrix in Figure 2. Performing the switch swap indicated produces a matrix with identical cost, yet the new matrix is not as routable. The original matrix routes any test vector of size 3, but the new matrix cannot route the input subset {1,5,6}.

In contrast, consider the switch matrix on the right of Figure 2 which cannot route the subset {1,2,3}. Here, the algorithm will find the single switch move indicated to lower the cost, and the resulting switch pattern routes *all* groups of 3.

6. RESULTS

We have developed a tool in C++ to construct and test sparse crossbars using the switch placement algorithm and evaluation method described above. A number of routing experiments have been run on sparse crossbars with 168 inputs and 24 outputs. This default size was chosen because it is small enough to run experiments quickly, and it is the same size as the one used in Altera's FLEX8000 family [1]. Altera has confirmed that the FLEX8000 sparse crossbar contains 2 switches for every crossbar input [3], however we are not privy to the location of the switches.

6.1 Adding Extra Switches

The first set of experiments investigate how sensitive the routability of a sparse crossbar is to the addition of switches. It is uncer-

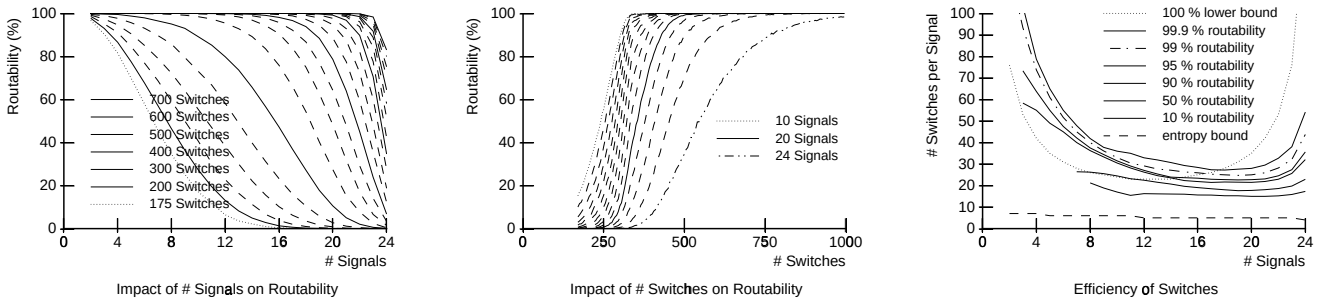

Figure 3: The effect of adding extra switches on routability of a 168×24 crossbar. The number of output wires is fixed at 24 in each graph. Three different ways of looking at the same routability data are presented. Monte Carlo simulations were done with 10,000 test vectors for each signal level.

tain if there is an obvious breakpoint in routability improvement, if routability increases in a smooth or discrete fashion, or how many switches are required to maintain a desired level. In Figure 3, a number of graphs and curves of the same data are shown to illustrate the effect of adding switches on routability.

The leftmost graph in Figure 3 shows a routability curve for each fixed switch count. For a given number of signals (*i.e.*, test vector size), 10,000 random vectors are routed. One curve shows the percentage of vectors routed as the test vector size increases. Clearly, large test vectors are more difficult to route, and sometimes the dropoff is very rapid. Each curve represents holding the number of switches constant: from 175 to 700 switches, in steps of 25. As switches are added, the entire routability curve shifts upward. Typically, the amount of the shift decreases as more switches are added, implying less utility is gained from each additional switch.

The middle graph in Figure 3 shows a similar routability curve for each test vector size, but the number of switches varies along the *x*-axis. For a given test vector size of 20 signals, for example, the greatest improvement in routability occurs as the number of switches is increased from 300 to 500. The largest test vector size of 24 signals shows the slowest improvement rate, and requires a large number of switches to become highly routable.

The rightmost graph in Figure 3 measures the utilisation of switches when routing the test vector signals. Each curve represents a fixed routability level, say 90%. The *x*-axis is the number of signals to be routed, and the *y*-axis is the smallest number of switches per signal required to achieve 90% routability. The curves show that most, but not all, of the crossbar outputs should be used to make efficient use of the switches. If nearly all of the outputs are used, *i.e.*, more than 20, significantly more switches per signal are needed to sustain the desired level of routability. Hence, the value obtained by adding each additional each switch in this region is small; many switches are needed to make a significant contribution to routability.

Two additional curves of interest are shown in this third graph: the entropy curve and the lower bound curve. The 100% lower bound curve, obtained from the formula $\lceil (n-k+1)m/(m-k+1) \rceil$, shows a lower bound for the minimum number of switches required to reach perfect routability. A large number of additional switches per signal is needed to go from 99.9% to 100% routability for large test vector sizes. This lower bound may also indicate inefficiency in our switch placements when the number of signals is small (≤ 16) — except the lower bound is not guaranteed to be tight.

The entropy curve shows the absolute minimum number of SRAM bits that would be needed to program the switch matrix. As shown by DeHon [8], the number of bits required is $\lceil \log_2 \binom{n}{k} \rceil$.

6.2 Adding Extra Output Wires

In the previous subsection, the switch matrix was designed with exactly 24 outputs to match the size of the Altera crossbar. Next, the number of crossbar output wires were gradually increased from 24 to 48, but the crossbar is used for only up to 24 signals. The results are shown in Figure 4 for a number of different switch counts. When the number of switches is low, the routability increase from having more output wires is not significant. However, once 340 switches are reached, dramatic improvements of up to 100% can be seen when additional output wires are used. Hence, a certain minimum number of switches must be present to take advantage of the extra output wires.

In the Altera FLEX8000 architecture, there is a cost associated with having more output wires. Each additional output must be considered as an additional input to the local interconnect in the Altera LABs[2]. If the local interconnect is to remain fully connected, additional switches must be placed inside the LAB. The total number of switches (sparse crossbar + local cluster) must be considered, and is shown in Figure 5. From this graph, it can be seen that the minimum number of switches at 99.95% routability is obtained with 30 output wires and approximately 1470 switches (510 switches in the sparse crossbar and 960 inside the LAB). This is significantly more than Altera's 1104 switches, but the level of routability is also much higher.

6.3 Adding Both Switches and Wires

To examine the combined effect of adding switches and widening the output stage of the sparse matrix, see Figure 6. Three key curves are shown: the baseline architecture, which is similar to the FLEX8000 with 336 switches and 24 outputs (dotted curve), the improvement from increasing to 30 output wires, and the improvement from increasing to 30 output wires and 510 switches (solid curves). In comparison, the 24-output crossbar is shown to be less routable with the same 510 switches (lower dashed curve).

However, adding output wires forced more switches to be added to the local interconnect. To make a fair comparison, these same switches should also be added to the 24-output crossbar, for a total of 702 switches (upper dashed curve). The result is still not as effective as the crossbar with more outputs.

6.4 Summary

The results from this section indicate that, to be area-efficient, a sparse crossbar should not be used at maximum utilisation. Rather, the number of outputs should be more than the number of signals

[2]A FLEX8000 LAB is a completely-connected cluster of eight 4-LUTs sharing 24 inputs.

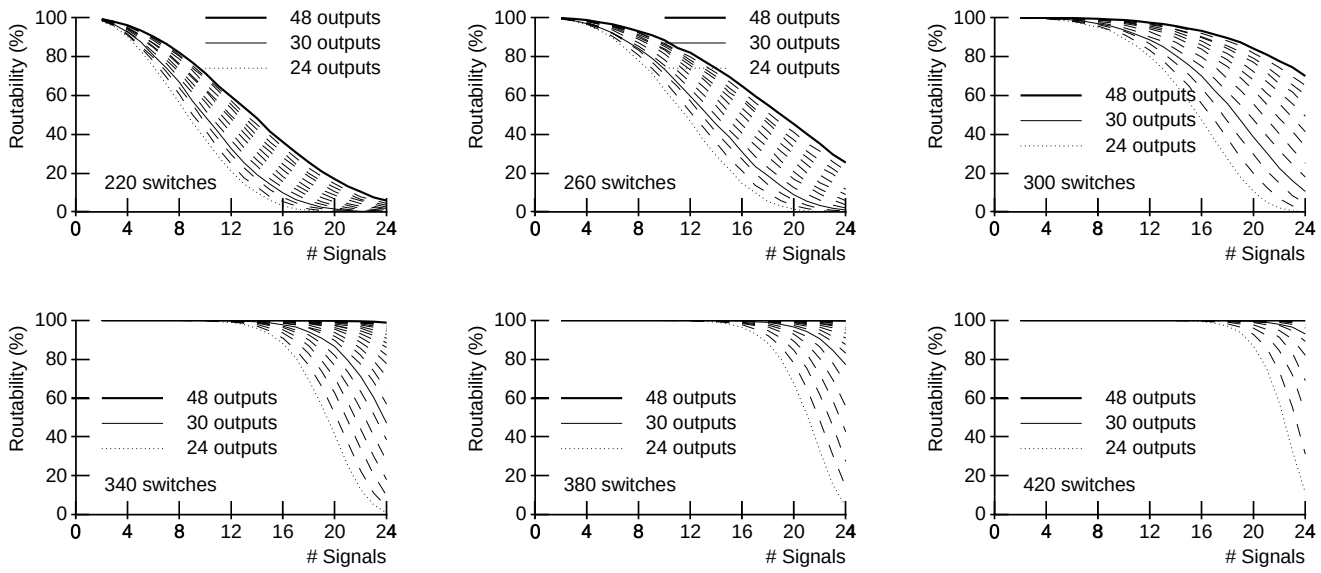

Figure 4: The effect of adding extra output wires on routability of a 168×24 crossbar. The number of switches is fixed in each graph. The curves in each graph show how routability improves as the number of output wires is increased from 24 to 48. There were 10,000 test vectors used for each signal level.

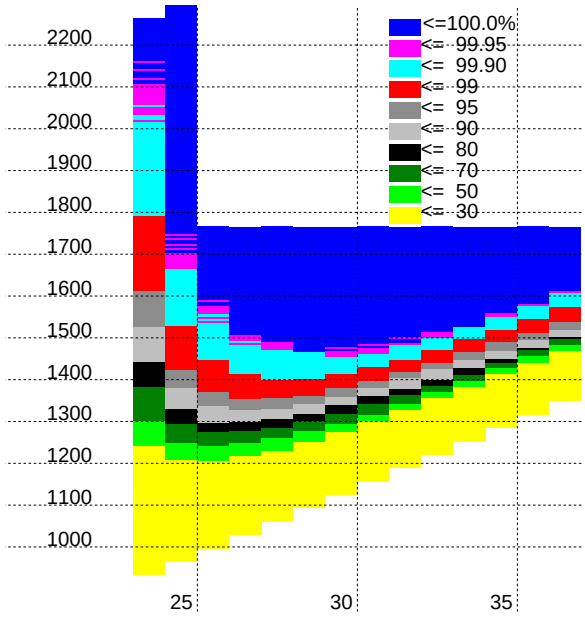

Figure 5: Effect of adding output wires on total switch counts. The number of outputs is varied along the *x*-axis, from 24 to 37. The total number of switches (including those in the sparse crossbar plus those in the full crossbar inside the LAB) varies along the *y*-axis. The shading of the graph represents the level of routability obtained for 20,000 test vectors, each containing 24 signals.

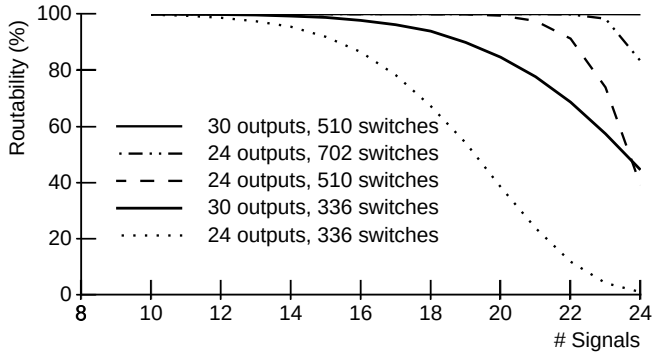

Figure 6: Combined effect of adding output wires and switches to achieve nearly 100% routability. A total of 20,000 test vectors were used for each signal level.

that are to be routed through it. As well, it is important to choose the number of switches and wires together, since a minimum number of switches are needed to benefit from the extra output wires.

7. DESIGN EXAMPLES

In the following design examples, based on architecture models in Figure 7, we searched for a number of sparse crossbar configurations which could achieve 95% or better routability and had the lowest area cost in terms of total transistors per LUT input. In counting transistors, we counted both the sparse crossbar switches and the switches of a fully-connected lower interconnect level.

Rather than use one SRAM cell and pass transistor per switch, we assumed a crossbar output is implemented using a single *n*-input multiplexer and encoded SRAM bits. We also assumed that each SRAM cell uses 6 transistors, and the *n*-input multiplexer uses a tree

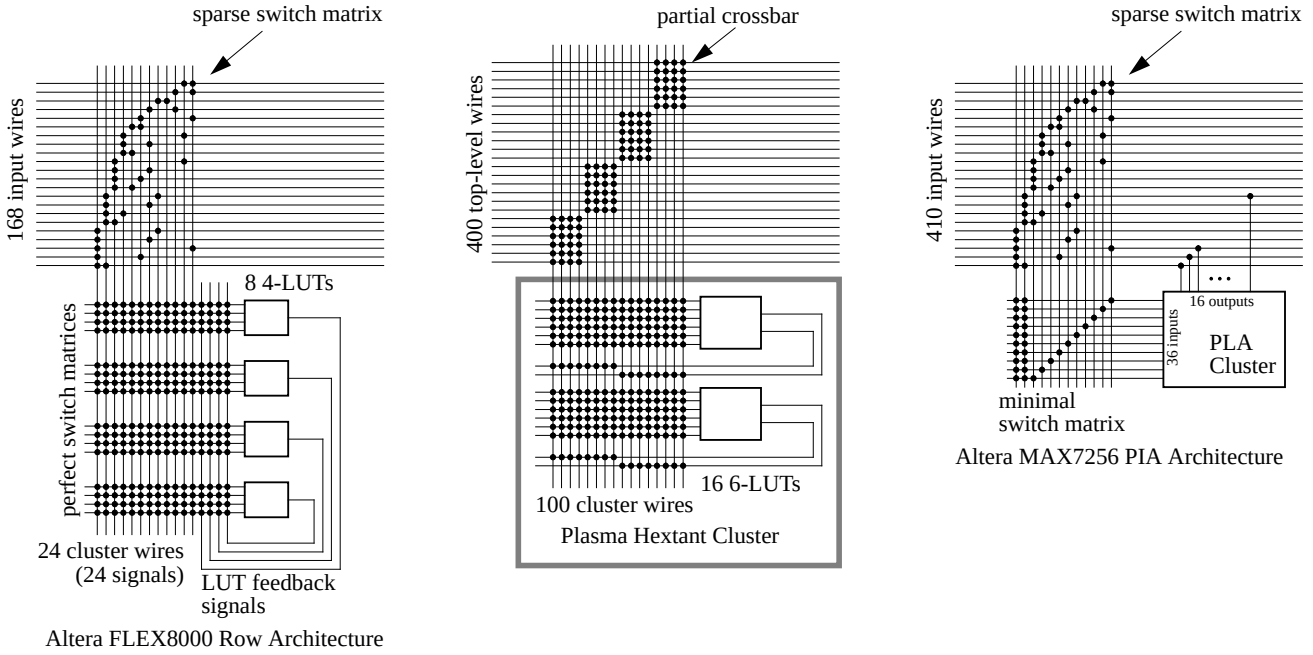

Figure 7: Models used for the Altera FLEX8000 FPGA, HP Plasma FPGA, and Altera MAX7256 CPLD architectures. The exact switch patterns shown were invented for illustration only.

of $2:1$ muxes requiring $2n-2$ pass transistors. To keep things simple, we did not account for any additional buffering of signals or wider transistor sizes that would accompany a real design.

To find the lowest-area configuration, we explored a variety of switch densities and wider-than-required output stages. Designs with 20% or more transistors than the baseline architecture were immediately rejected. The baseline architecture chosen assumed two switches per input wire and a fully connected local interconnect (similar to Altera FLEX8000). We performed a quick routability simulation for the remaining designs using only a few (1000) test vectors, and rejected those with less than 95% routability at the largest test vector size. We retested the some of the remaining designs with a larger number of test vectors to ensure their measured performance.

7.1 Altera FLEX8000

The Altera FLEX8000 device uses a 168×24 sparse crossbar to connect the FastTrack row wires into the LAB clusters. The sparse crossbar is 1/12 populated, such that each row wire has two "opportunities" to connect into a cluster. Within the cluster, the eight 4-LUT inputs select from 24 sparse crossbar outputs and 8 LUT feedback signals using a full crossbar. This design uses approximately 129 transistors per LUT input, including the cluster interconnect. With our switch placement, the routability is excellent when there are fewer than 10 signals entering a cluster. However, if 15 or more signals enter a cluster, the routability drops below 90%. At full capacity, the routability of 24 signals drops below 1%.

Our construction techniques and search found a 168×29 sparse crossbar containing 464 switching points, or 2.7 connections per input wire. This design uses approximately 157 transistors per LUT input, including the LUT feedback connections, representing an increase of 22%. Assuming the cluster interconnect used a minimal crossbar instead of a full crossbar, our search found a 168×26 crossbar with 546 switching points, using 147 transistors per LUT input, an increase of only 14%. For a modest increase in transistor count, the improvement in routability shown in Figure 8 is dramatic. Some other organizations found are listed in Table 1.

7.2 HP Teramac Plasma

Teramac[4], from HP Labs, is a large reconfigurable system made up of custom-designed Plasma[5] FPGAs. A full Teramac system is designed to have the capacity of about one million gates distributed over 1728 FPGA chips. An important goal in the Plasma design was to design a highly routable FPGA: to limit compile times to about an hour, placing and routing each FPGA must be done quickly (within 3 seconds). This approach meant each FPGA should be nearly 100% routable so that almost no time would be spent in ripup or repartitioning the mapped circuit.

The Plasma 2-level hierarchy comprises sixteen clusters, called *hextants*, of sixteen LUTs each. The six LUT inputs in each cluster are fully connected to 100 cluster-level wires, and the two LUT outputs are 1/2 populated. At the top level there are 400 signal wires, which must connect to the 100 cluster wires. This 400×100 partial crossbar is 1/4 populated using $10,000$ crosspoints, implying it is composed of four diagonally placed 100×25 full crossbars. Conservative measurements of the die photograph in [5] indicates that the partial crossbar switches alone consume 23% of total chip area, or 32% of the core area (excluding the I/O pads).

The Plasma chip is easy to route because the partial crossbars make it predictable to route: as long as fewer than 25 signals enter each full crossbar, it can be routed. The router need only consider which crossbar it routes to, and not the precise detailed route. Hence, there would be no need for ripup. Despite this advantage, there are few signal assignments that can satisfy the partial crossbar when more than 75 input signals are required, as shown in Figure 8.

Our sparse crossbar search found a 400×104 sparse crossbar with $1,456$ switching points, or 3.6 switches per top-level wire, for a switch density of roughly 1/28. This design uses approximately 292 transistors per LUT input, including the cluster-level interconnect (but not the LUT output switches).[3] Even though this sparse cross-

[3]Plasma used one 5T SRAM cell and one pass transistor per switch-

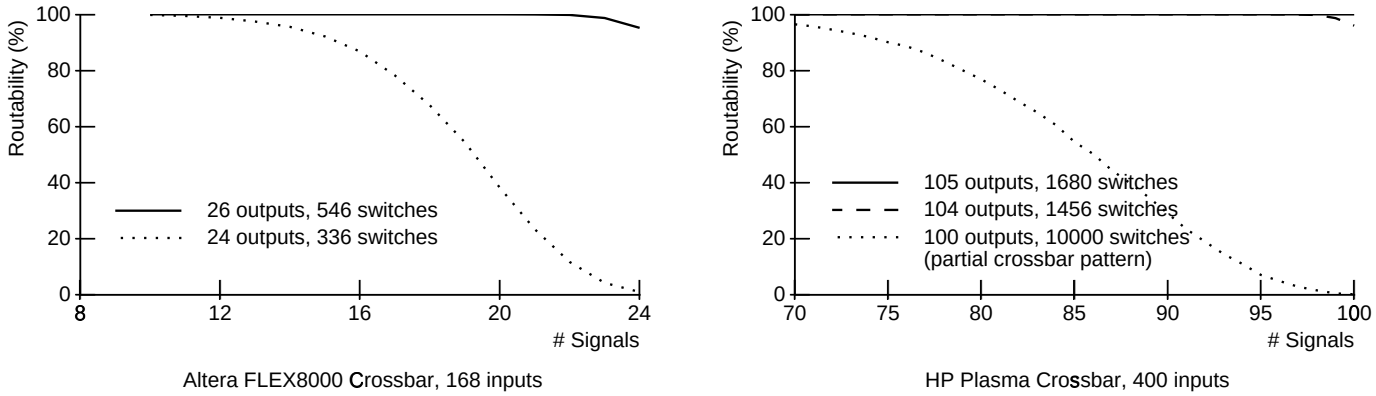

Figure 8: Routability Improvements made to the FLEX8000 and HP Plasma architectures.

Table 1: Highly routable, area-efficient sparse crossbars suitable for use in the Altera FLEX8000 family. The first group shows the number of switches required to obtain high routability with exactly 24 outputs. The second group adds a minimal crossbar between the sparse crossbar and the LAB full crossbar to reduce the sparse crossbar outputs to 24. The third group widens the full crossbar local interconnect inside the LAB to match the number of sparse crossbar outputs. All transistor counts include the local interconnect. Transistor counts in parentheses indicate that further reduction is possible if minimal crossbars are used within the LAB instead.

Crossbar Size	Switching Points	Transistors	Routability
Provide exactly 24 outputs from the sparse crossbar.			
168×24	336	4144	1.0%
168×24	1008	5776	98.6%
Reduce to exactly 24 outputs using an additional minimal crossbar.			
$168 \times 26 \times 24$	$546 + 72$	5468	96.1%
$168 \times 25 \times 24$	$700 + 48$	5492	98.2%
$168 \times 27 \times 24$	$567 + 96$	5650	99.2%
$168 \times 29 \times 24$	$464 + 144$	5694	98.6%
$168 \times 30 \times 24$	$450 + 168$	5800	98.2%
Provide more than 24 outputs, increase cluster interconnect.			
168×29	464	5022 (4830)	98.6%
168×30	450	5080 (4888)	98.2%
168×26	546	5084 (4700)	96.1%
168×31	434	5134 (4942)	98.6%
168×27	567	5218 (4834)	99.2%
168×25	700	5300 (4916)	98.2%

Table 2: Highly routable, area-efficient sparse crossbars suitable for use in HP Plasma FPGAs. Total transistor counts include the local cluster interconnect to choose LUT inputs, but not the LUT outputs.

Crossbar Size	Switching Points	Transistors	Routability
Partial crossbar switch pattern used by HP.			
400×100	10000	47040	0.4%
Sparse crossbar patterns found.			
400×104	1456	28048	95.9%
400×105	1365	28080	95.1%
400×103	1648	28218	98.8%
400×106	1378	28320	98.6%
400×107	1284	28346	96.9%
400×108	1296	28584	99.3%
400×109	1199	28604	97.1%
400×105	1680	28710	100.0%
400×102	1734	28788	96.6%
400×108	1404	28800	99.9%

bar contains nearly 1/7 the number of switching points of Plasma, it has significantly improved routability. It can route over 95% of vectors containing 100 input signals, whereas Plasma can route less than 1%. Given this new switch pattern, a router would have even higher assurances it could route each Plasma chip independently. Alternatively, a sparse crossbar of size 400×105 with $1,680$ switching points, or 299 transistors per LUT input, can be constructed which routes over 99.9% of the test vectors. A few other organizations found are listed in Table 2.

7.3 Altera MAX7000

The Altera MAX7256 CPLD has 2-levels of hierarchy, where the top level contains multiple sparse $n \times 36$ crossbars. The value of n used in the MAX7256 device is not known, but it is probably not more than 410 (one wire for each macrocell output and I/O pin). Here, we shall assume $n = 410$.

We have noted before that providing more than 36 crossbar outputs is necessary to keep area low while obtaining high routability. Rather than increase the number of inputs to the product-term AND planes by this amount, we have chosen to connect the crossbar outputs to a minimal crossbar, which can perfectly select any 36 of these signals for the AND plane. In this case, this minimal crossbar is area-efficient because it is close to being square, so it requires

ing point, but we are assuming a different implementation here.

Table 3: Highly routable, area-efficient sparse crossbars suitable for use in the Altera MAX7256 CPLD. Total transistor counts include the minimal crossbar selector, if appropriate, but not the product term array.

Crossbar Size	Switching Points	Transistors	Routability
1-level, provide exactly 36 outputs from the sparse crossbar.			
410×36	$2448 + 0$	6336	96.40%
410×36	$2952 + 0$	7344	99.70%
2-levels, reduce to exactly 36 outputs using a minimal crossbar.			
$410 \times 43 \times 36$	$1161 + 288$	4678	97.2%
$410 \times 42 \times 36$	$1218 + 252$	4692	97.6%
$410 \times 41 \times 36$	$1271 + 216$	4698	97.8%
$410 \times 39 \times 36$	$1443 + 144$	4860	96.7%
$410 \times 45 \times 36$	$1080 + 360$	4932	96.2%
$410 \times 38 \times 36$	$1558 + 108$	4984	96.2%
$410 \times 40 \times 36$	$1360 + 180$	5016	97.2%
$410 \times 43 \times 36$	$1333 + 288$	5022	100.0%

few switching points.

Assuming an SRAM and mux-based crossbar implementation, the best organization found used a 410×43 sparse crossbar containing 1161 switching points, or about 2.8 switches per input. This crossbar is over 97% routable when 36 signals are required. The minimal crossbar requires an additional 288 switching points. A total of 4678 transistors would be required to construct the two switching stages, or 129 transistors per output. Alternatively, a 410×43 sparse crossbar containing 1333 switching points was found to achieve over 99.9% routability. This system used 5022 transistors, or 139 transistors per output.

Both of these 2-level organizations use significantly fewer transistors (and switching points) than an architecture with only one sparse crossbar containing exactly 36 outputs. The best 2-level organization contains 26% fewer transistors and 40% fewer switching points than the best 1-level. A few other organizations found are shown in Table 3.

7.4 Varying FLEX8000 Cluster Size

In this section, we present the results of generating highly routable sparse crossbars for variations of the Altera FLEX8000 architecture shown in Figure 7. The goal is to understand the impact of cluster size and crossbar input size on the area efficiency of the sparse crossbar. To do this, we normalize the area based on transistors per LUT input. This accounts for the increased logic capacity of larger clusters, and allows us to directly compare the results.

We varied the cluster size, N, from between 2 and 12 LUTs, the number of top-level wires, n, were varied from 168 to 995. The maximum number of output signals required by the crossbar was set to be $3N$, which gives 24 inputs for the FLEX8000 case of $N = 8$. We also repeated these experiments with $2N + 2$ output signals, as recommended by Betz [6].

In general, it was usually possible to find multiple sparse crossbars which are both area-efficient and fit within the desired routability constraints. When multiple designs matching the criterion were found, the one with the lowest transistors per LUT input was selected. Sometimes a design could not be found, so the data point was left out of the results.

First, we examine the impact of cluster size on area as shown in the top graphs of Figure 9. For a sparse crossbar with only 168 inputs the effect of cluster size is not significant on area, but it can be seen that a cluster size between 4 and 7 gives the best efficiency. In contrast, small cluster sizes become very inefficient when the number of crossbar inputs is increased. Selecting a cluster size of at least 8 is necessary in these cases. Letting the aspect ratio of the sparse crossbar get too large hinders the efficiency.

Next, we examine the impact of increasing the number of crossbar inputs for specific cluster sizes, as shown in the lower graphs of Figure 9. This is an orthogonal view of the same data, except some cluster sizes have been left out for clarity. From this data, we can see that the area cost for a given cluster size is a roughly linear function of the size of the crossbar. The slope of the larger cluster sizes is smaller, making them more area-efficient at larger crossbar sizes.

8. CONCLUSIONS AND FUTURE WORK

We have shown a method for evaluating and constructing sparse crossbars. The construction technique is based on an understanding of Hall's Theorem to generate highly routable crossbars.

Routability of sparse crossbars can be improved by adding additional switches and by widening the output stage of the crossbar. The latter method was the most effective once there were enough switches to be used: approximately two per input in the case of a 168×24 crossbar. Careful evaluation using both methods is necessary to obtain optimum routability at minimum area.

We have demonstrated with a few design examples that it is beneficial to plan to underutilise the output stage of a sparse crossbar and design using the correct number of switches. In the Plasma example, only 4 additional output wires and a switch density of 1/28 was required to remain highly routable; this organization uses 75% fewer switches than Plasma, and obtains superior results. In the Altera FLEX8000 example, 5 additional output wires and a slight increase in switch density (from 1/12 to about 1/10) was needed to obtain over 95% routability. It was also found that cluster sizes between 4 and 7 give the most area-efficient interconnect in the FLEX8000.

Planning on high routability does not require an exorbitant amount of switching resources: in the examples given, the most dense switch pattern used was less than 1/10 populated.

For future work, we have tried to design and simulate cascaded sparse crossbars that depend on one another. To date, we have not been successful in generating consistently improved results with cascaded crossbars — independently optimized crossbars produce more consistent results. Also there is difficulty modeling congestion with a network flow solver, so plans are underway to integrate the construction algorithms into an actual router.

9. REFERENCES

[1] Altera, San Jose, CA. *1996 Data Book*, 1996.

[2] Altera, San Jose, CA. *News & Views Newsletter*, August 1999.

[3] Altera. *Private communication.*, 1999.

[4] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *IEEE Symposium on FPGA's for Custom Computing*, 1995.

[5] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, and G. Snider. Plasma: An FPGA for million gate systems. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 10–16, 1996.

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

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1993.

[8] A. DeHon. Entropy, counting, and programmable interconnect. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 73–79, 1996.

[9] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 69–78, 1999.

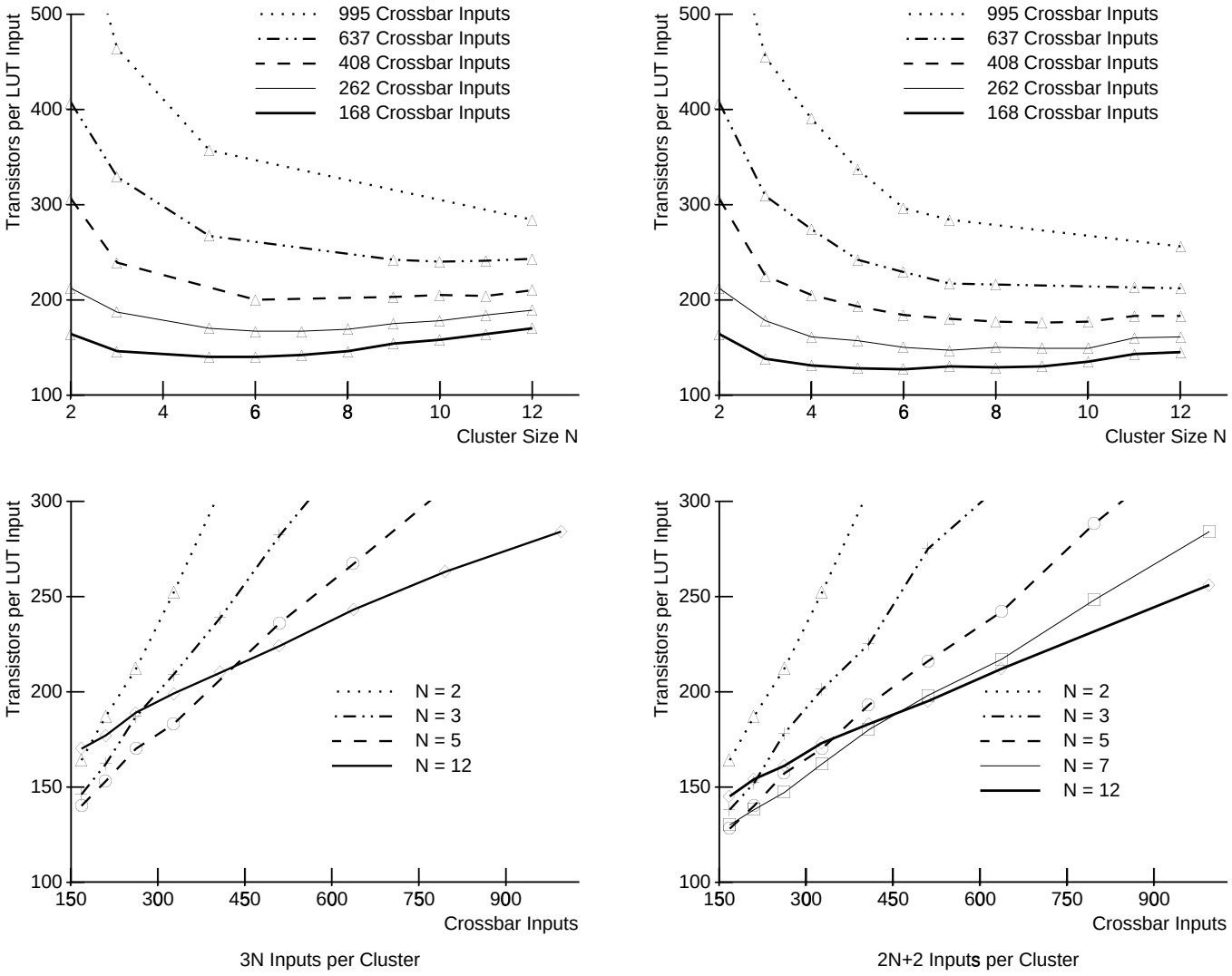

Figure 9: Effect of varying the cluster size N (upper graphs) and number of top-level inputs (lower graphs). There are $3N$ inputs per cluster on the left, $2N+2$ inputs on the right.

[10] A. El Gamal, L. A. Heinachandra, I. Shperling, and V. K. Wei. Using simulated annealing to design good codes. *IEEE Transactions on Information Theory*, 33(1):116–123, January 1987.

[11] K. Fujiyoshi, Y. Kajitani, and H. Niitsu. Design of optimum totally-perfect connection-blocks of FPGA. In *IEEE International Symposium on Circuits and Systems*, pages 221–224, May 1994.

[12] W. Guo and A. Y. Oruç. Regular sparse crossbar concentrators. *IEEE Transactions on Computers*, 47(3):363–368, March 1998.

[13] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[14] I. S. Honkala and P. R. J. Östergård. Applications in code design. In E. Aarts and J. Lenstra, editors, *Local Search in Combinatorial Optimization*, chapter 12. Wiley, 1997.

[15] P. Leventis. Placement algorithms and routing architecture for long-line based FPGAs. Bachelor thesis, University of Toronto, 1999.

[16] S. Nakamura and G. M. Masson. Lower bounds on crosspoints in concentrators. *IEEE Transactions on Computers*, C-31(12):1173–1179, December 1982.

[17] A. Y. Oruç and H. M. Huang. Crosspoint complexity of sparse crossbar concentrators. *IEEE Transactions on Information Theory*, 42(5):1466–1179, September 1996.

[18] J. Vargese, M. Butts, and J. Batcheller. An efficient logic emulation system. *IEEE Transactions on VLSI*, 1(2):171–174, June 1993.