Chapter 11. Logic Partition Orderings

Introduction

Chapter 10 discussed methods of efficiently splitting a logic circuit into two parts. In order to harness these techniques for multi-FPGA systems, some method of breaking a circuit into more than two parts is necessary. There has been a significant amount of work done on multi-way partitioning . However, these works have primarily focused on problems where there are no restrictions on how the partitions are interconnected (that is, there is no reason to prefer or avoid connections between any pair of partitions). Unfortunately, in many multi-FPGA systems only a subset of the FPGAs are connected, and routing between FPGAs not directly connected will use many more external resources than routing between connected FPGAs. Works that have handled the limited connectivity problem take a significant amount of time [Roy93], possibly even exponential in the number of partitions [Vijayan90].

Our approach to the multi-FPGA partitioning problem is to harness the work on standard bipartitioning, such as that discussed in Chapter 10, as well as multi-way partitioning algorithms for some restricted situations. We do this by recursively applying the bipartitioning algorithms to the circuit until it is cut into the required number of pieces. While this approach is greedy, and thus if applied unwisely can produce poor results (Chapter 9), we can use the greediness to our advantage. If the first "greedier" cuts of the logic correspond to the most critical bottlenecks in the multi-FPGA system, then optimizing these cuts more than subsequent cuts is the correct thing to do. In fact, this method of iterative bipartitioning has already been applied to partitioning of logic within a single ASIC to simplify placement [Suaris87]. In this approach, the chip is divided recursively in half, alternating between horizontal and vertical cuts.

One issue the multi-FPGA system partitioning problem raises that ASIC partitioning does not is how to find the critical routing bottlenecks in the system. In an ASIC, all routing occurs on a flat surface, and a cut through the center of the chip represents the routing bottleneck. A multi-FPGA system can have an arbitrary topology, and it may not be obvious where the critical bottleneck is. What is necessary is to find a partition ordering which specifies the critical bottlenecks in the multi-FPGA system and determines the order of cuts to make in any logic being mapped to that topology. In many cases we can handle this by requiring that the designer of the multi-FPGA system explicitly specify the proper cuts to make. However, there are several issues in multi-FPGA systems that argue for an automatic method for finding these bottlenecks.

The primary argument for automatic methods to find critical bottlenecks is the current trend towards more flexible multi-FPGA system architectures. Instead of fixed systems with a finite mix of resources, many current systems are offering much greater flexibility (see Chapter 5). For some, it is primarily the ability to connect together small systems to handle much larger problems, possibly with the additional capability of adding other non-FPGA chips. Other systems have a fixed connection pattern between daughter card or chip locations, but allow a large variety of types and capacities of resources to be inserted. There is also the possibility of much greater flexibility, with systems that allow very complex and customized topologies to be built (such as our Springbok system, described in Chapter 6). While rules for static partition orderings might be generated in some cases, the flexibility of current and future systems forces us to adopt an automatic approach to the partition ordering problem. A software system capable of automatically adapting to an arbitrary multi-FPGA topology also becomes a valuable tool for people building their own multi-FPGA system. Since the tool automatically adapts to the topology, it can be used by designers of systems who have little or no knowledge of partitioning techniques.

Another trend is towards faster automatic mapping turnaround times. Just as was found for software compilation, a faster turnaround time from specification to implementation allows a more interactive use of a multi-FPGA system, making these systems more attractive. One way to speed up the mapping process is to retain most of a previous mapping after a small change, and simply update incrementally. Even when starting from scratch, if the mapping process fails because it uses too much of some resource, it is better to remap only that part of the system where the failure occurred (possibly with neighboring parts that have resources to spare) than restart the entire mapping process. In either case, the area(s) to be remapped are likely to be very irregular. Even if it is easy for the designers to determine a partition ordering for the entire system, precomputing a partition ordering for all possible subsets is very difficult. Again, the solution to this problem is to develop an automatic algorithm to determine partition orderings.

In the rest of this chapter we discuss our solution to a problem we believe has not yet been investigated: determining how to iteratively apply 2-way and N-way partitioning steps to best map a circuit onto chips in a fixed topology.

Partition Ordering Challenges

Before we discuss solutions, we need to define some terms. In this chapter, the *topology* is the FPGAs and connections built into the multi-FPGA system. A *partition* is a subset of the topology where all FPGAs are on the same side of all cuts made in the system. A partition may be broken into *groups* by a candidate cut, with each group being a set of FPGAs from the partition which are still connected after the cut is applied.

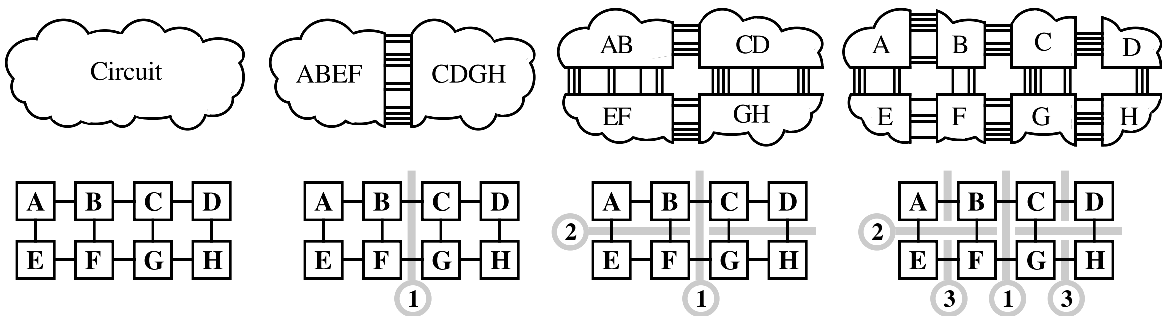

Figure 85. Example of iterative bipartitioning. The circuit (top left) is partitioned onto the multi-FPGA topology (bottom left) in a series of steps. Each partitioning corresponds to the most critical bottleneck remaining in the multi-FPGA system, and after each partitioning the placement of the logic is restricted to a subset of the topology (labeling on circuit partitions).

We can approach the problem of determining a partitioning order as a partitioning problem on the multi-FPGA topology itself. That is, we recursively split up the topology graph, attempting to optimize some cost metric on the routing resources cut. Then, we split the logic graph in the same manner, restricting the logic on each side of the split to the available capacity on either side of the corresponding cut of the topology, while attempting to keep the number of signals cut in the logic graph to be less than the amount of routing resources crossing the cut in the topology. An example of this is in Figure 85.

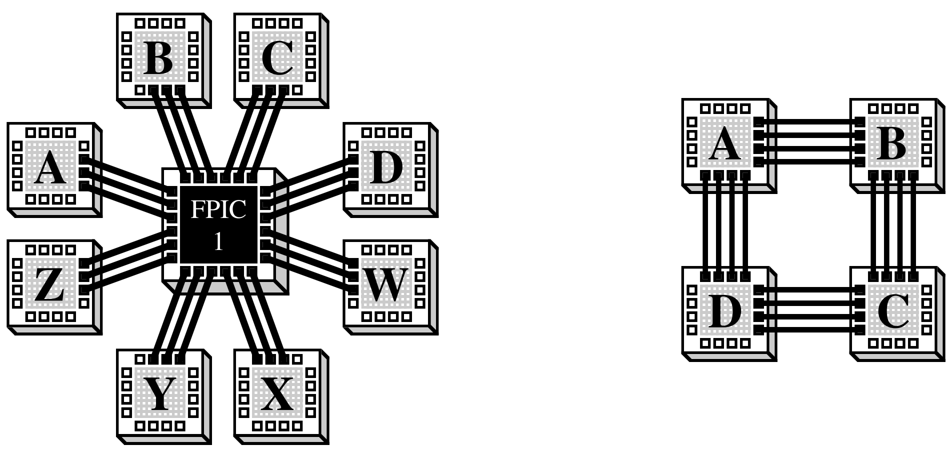

Figure 86. Example topology for the discussion of connectedness of partitions (left), and for the discussion of multiple partition creation (right).

There are two important issues that must be considered in any partition ordering computation: partitions must be connected, and a cost metric for evaluating cuts must be determined. The issue of connected groupings is demonstrated in Figure 86 left. The topology shown has a central chip **R** that acts as a routing hub, and eight other FPGAs connected only to **R**. If we apply a standard partitioning algorithm to this topology, it is likely that the partition that does not include **R** will include two or more FPGAs. For example, one likely cut has **A**-**D** on one side, and **W**-**Z** plus **R** on the other. The problem with this cut occurs when we try to partition the logic according to this cut scheme. If we assume all the connections between **R** and the other FPGAs contain 10 wires, then when we do the cut described above we will allow 40 nets to be cut. However, this ignores the fact that once the **A**-**D** partition is broken up into the individual FPGAs, there may need to be connections between these FPGAs. However, there are no wires within the **A**-**D** partition to handle these connections, and the wires needed to make these connections may have already been allocated for signals to the **W**-**Z** partition during the first split. What is necessary is to require that any partitioning of the circuit produce connected partitions. Thus, there will be at least one possible path within a partition to carry signals between its FPGAs (ensuring that there are enough wires is the responsibility of the cost metric for evaluating partitionings, discussed below). Thus, in a bipartitioning of the topology in Figure 86 left, all but one of the FPGAs, including **R**, would have to be in one partition, and the other FPGA in the other partition (better methods for this topology will be discussed later).

In the introduction, we discussed finding the critical bottlenecks in the system, and partitioning the logic accordingly. However, we didn't define what a critical bottleneck is. For our purposes, we define the critical bottleneck as that cut through the system that most restricts the routing. That is, assume that we take a very small circuit and map it randomly to our topology. If it fits, we repeat this with a slightly larger circuit, until we can no longer route the circuit. This will yield at least one set of edges that are completely saturated with signals, and which splits the circuit into two or more pieces. Since these edges are the first that become cut, they are probably the edges that will most restrict the mapping of a circuit to this topology, and represent where the most care should be taken during partitioning. The critical bottleneck is within this set of edges. We can evaluate a cut in a topology (which splits it into two parts) and determine its criticality. If L_1 is the total logic capacity of all FPGAs on one side of a cut, and L_2 is the capacity on the other side, then out of $(L_1+L_2)^2$ signals, $2*L_1*L_2$ signals will cross this cut in a randomly mapped circuit (The factor of 2 is because we treat the source and destination separately, so a route from A to B is different that a route from B to A). Thus, $(2*L_1*L_2)/(L_1+L_2)^2$ is the percentage of all nets crossing this cut. Since (L_1+L_2) is a constant for a given topology, we can ignore it. Let $wire_{12}$ be the number of edges in the topology crossing the cut. Thus, the cut that will saturate the earliest is the cut with the least edges to handle the most routing. To define the criticality of a cut, we use the ratio of $wire_{12}$ to the percentage of nets crossing this edge (without the constant terms), which is $wire_{12}/(L_1*L_2)$. Note that this is the standard ratiocut metric [Wei89]. The lower the ratiocut value, the more critical the cut in the topology.

Basic Partition Ordering Algorithm

As we discussed earlier, what we want is an algorithm that will determine how to partition logic onto a given topology. It needs to find a cut in the topology that reflects the most constrained routing (assuming random placement of logic), while ensuring that the partitions formed are connected. There is an existing algorithm that provides much of this functionality. As proposed by Yeh, Cheng, and Lin [Yeh92], we can partition a graph by iteratively choosing random start and end points of a route. We then find the shortest path between these two points, and use up a small fraction of the resources on this path to handle this route. Distances are *exp(10*flow/cap)*, where *cap* is the capacity of the edge, and *flow* is the amount of that capacity already used by previous routes. A net whose capacity has been used up is removed from the system, and no longer connects between any of its terminals. The iteration of choosing random points and then routing between them is repeated until there are no longer any resources available to route between the next random pair of points. At this stage, we have broken the circuit up into at least two groups, where a group is a set of all nodes still connected together. Note that there may in fact be more than two partitions created. For example, if we are partitioning three FPGAs connected by only a three-terminal wire, the system will be broken into three partitions. If we are partitioning four FPGAs in a mesh (Figure 86 right), it may be broken into four partitions, since the resources on all four edges may be used up.

There are several modifications that we have made to this basic algorithm. First, using the edge length in the original algorithm, namely *exp(10*flow/cap)*, yields real-valued path lengths. During the shortest-path calculation, we have a queue of currently found shortest paths. We remove the shortest of these paths, and add the neighbors of this path back into the queue, as long as that neighbor has not yet been reached by a shorter path. Since the edge lengths are real values, we need to use a tree data structure for the queue, resulting in *O(log n)* time for each insertion and deletion. To fix this, we have changed the algorithm to use an edge length of $round(exp_2(10*flow/cap))$. Since we round the edge lengths, all path lengths are integers. This, plus the fact that the maximum length of an edge is 2^{10} (or 1024), means we can very efficiently implement the shortest path calculation. Instead of a tree used as a queue of current shortest paths, we have an array of lists. Since the path we are extending at each step is the smallest still in the queue, and since the maximum path we can add to the list is at most 1024 longer than the current path, we only need keep 1025 separate active queues (see Figure 87). Thus, we can implement the queue as an array of 1025 elements (with element j representing paths of length j+i*1025, for i≥0). In this way, insertions *O(1)*. Deletions are also *O(1)* since at most we must look at each queue location for the next element, and there are a constant number of locations.

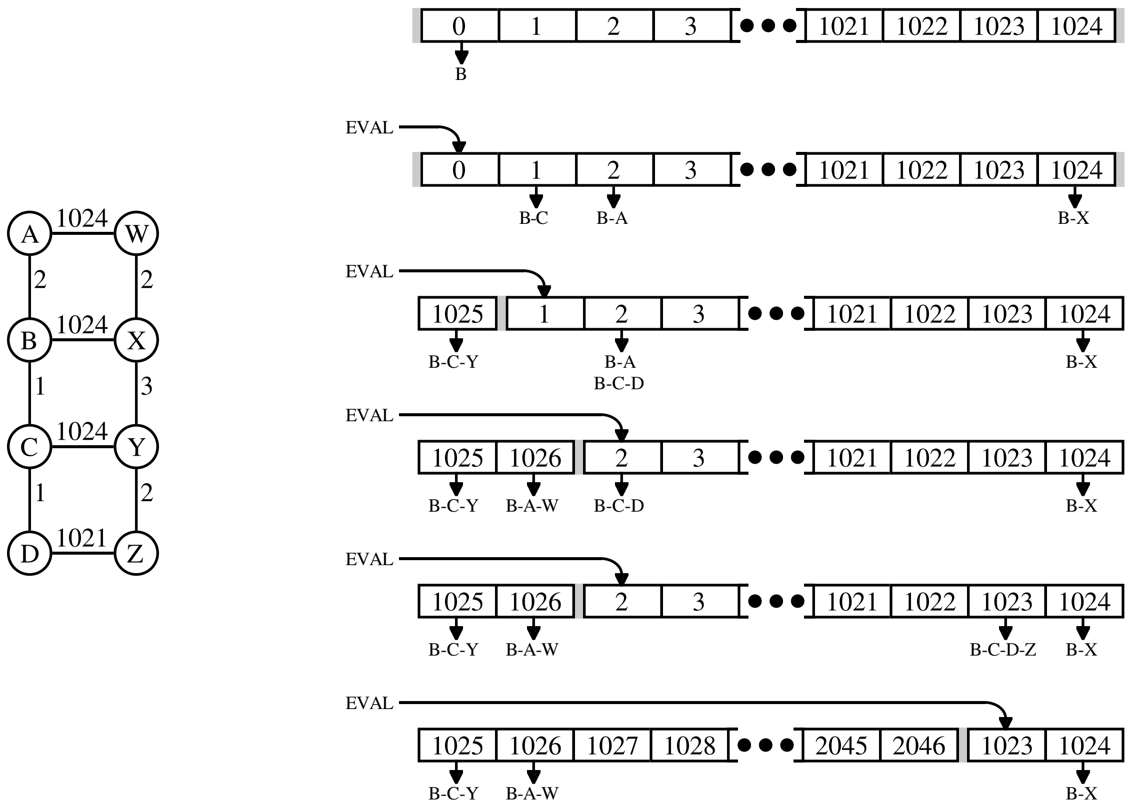

Figure 87. Queue formulation of the shortest-path algorithm. In attempting to find the shortest path from **B** to **Z** in figure at left (number on edges are edge lengths), the queue states are shown in order at right, from top (earliest) to bottom (latest). In the first step, the source is inserted in the 0 bucket. Then iteratively the shortest path is removed, and the neighbors are added to the appropriate bucket. Notice that the 0 bucket is reused as the 1025 bucket, 1 as 1026, and so on. In the final step, the shortest path from **B** to **Z** is found in bucket 1023, and the algorithm finishes.

Another issue is the distribution of sources and sinks of random routes in the algorithm. Not all chips in a multi-FPGA system are the same, and the choice of source-destination pairs needs to reflect this. Most obviously, if a routing-only FPGA or FPIC is included in the topology, there should be no sources or sinks assigned to that chip. The capacity of logic-bearing chips should also be considered. If one FPGA in the system has twice the capacity of another, then twice as much logic will likely be assigned to it, and twice as many random routes should start and end there. Thus, the distribution of random routes should be directly related to the logic capacities of the chips in the system. Note that if a route starts and ends in the same FPGA it can be ignored. One final consideration is the handling of external inputs and outputs. In many multi-FPGA systems there will be specific ports that are used for communication with the environment, or with non-FPGA chips within the multi-FPGA system. Note that for our purposes FPICs are treated as FPGAs with zero logic capacity, so "non-FPGA chips" refers to memories, microprocessors, and other devices that cannot be assigned random logic nor can be used to route inter-FPGA signals. To handle the routing demands caused by these external connections, the FPGAs to which they are connected have their capacities increased, which increases the portions of random routes beginning or ending in that chip.

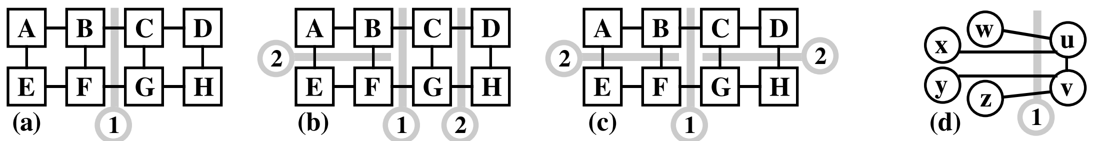

Figure 88. Examples of multiple cuts in a topology (cuts are in gray, with lower numbers indicating earlier cuts). If the initial cut (a) is used to split the entire system, uncoordinated subcuts (b) may occur. Keeping the system together would generate more coordinated cuts (c). An example circuit is also shown (d).

Now that we know how to perform a single cut of the topology, the question arises of how to recursively apply this algorithm to generate multiple cuts in the system. The obvious approach is to use the cut generated in each step to break the topology into two separate topologies, and handle these halves independently. However, consider the system shown in Figure 88a. If we use the first cutline generated (gray line labeled "1") to split the topology, then the two halves will independently choose either a horizontal or vertical cutline (Figure 88b). There are two problems with this. First, there is the issue of terminal propagation [Dunlop85]. When we perform the split between **AB** and **EF**, there often needs to be information transferred between both sides of the previous cut. For example, say that we are mapping the circuit in Figure 88 d, and have already partitioned **u** and **v** onto **CDGH**, and **w**-**z** onto **ABEF**. It should be obvious that **w** and **x** should be partitioned into the same group, so that **u** can be placed in the adjacent FPGA on the other side of the first cut (for example **w** and **x** can be put into **B**, and **u** into **C**). If this is not done, one or more extra vertical connections will be necessary. However, with the partition ordering specified in Figure 88b, there is no way for the partitioner to figure this out, since **u** and **v** have not yet been partitioned into the top or bottom half of the system. However, if we use the partition ordering shown in Figure 88 c, **u** and **v** will be assigned to either the top or bottom half of the topology, and terminal propagation algorithms [Dunlop85] can be applied to properly group together **w** and **x**. A second issue is multi-section algorithms [Suaris87, Bapat91]. These algorithms can often take two or more intersecting cuts, such as those in Figure 88c, and perform them simultaneously (both cuts labeled 2 in the figure would be considered as one single cut, so all three cuts shown could be done simultaneously). These algorithms would be able to recognize a situation such as the mapping in Figure 88d, and handle it correctly.

The answer to the previous issues is to perform similar cuts on either side of a previous cut simultaneously. There is a simple way to accomplish this in our current algorithm. Instead of breaking the topology up into two independent halves after a cut is found, we instead keep the topology together, and restrict the random routes we consider. That is, we only consider random routes that start and end in the same side of all previous cuts. So in Figure 88a, after the cut labeled 1 has been performed, the topology is kept intact. Routes between **A**, **B**, **E**, and **F** would be considered, but no routing between FPGAs on both sides, such as between **A** and **C**, would be generated. However, the shortest paths found in the algorithm could cross the previous cut. So, if a large number of routes had already used some of the vertical wires **AE** and **BF**, a route from **B** to **F** could choose to go through **C** and **G**. In this way, the two sides of the cut will tend to affect each other, and generating similar cuts in both partitions is likely. Note that it is possible that edges moving across a previous cut will become saturated before others, and we might expect to find this previous bottleneck again. However, our stopping condition on the iteration is when there is no path between the source and sink of a random route. Since we never consider routes between FPGAs on both sides of a cut, the iteration continues until at least one of the partitions is split.

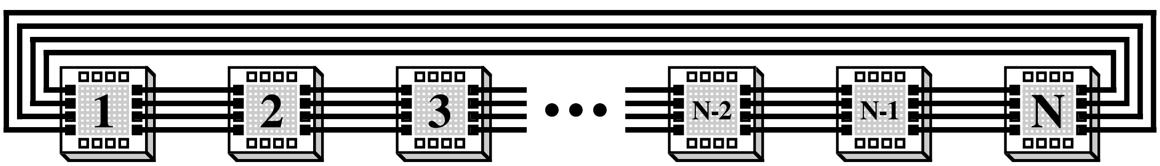

Figure 89. Ring topology, one of the most symmetric topologies.

As was alluded to earlier, it is possible (and quite likely in topologies such as tori and rings) that the sectioning found will split a partition into more than two groups. For example, consider a ring of N FPGAs where FPGA i is connected to FPGAs $(i+1)$ and $(i-1)$, with FPGA N connected to FPGA 1 (see Figure 89). If all the connections and FPGAs have the same capacities, there is no obvious point to cut the topology, and in fact many or all of the connections may be cut in one partitioning step. If we allow partitions to be broken into multiple groups, we would then need an algorithm that can do multi-way partitioning with routing constraints between partitions, the very thing this algorithm is built to avoid. The solution to this is to detect cases where a partition is being broken into multiple groups, and combine groups together until no partition is broken into more than two groups.

To combine groups together, our algorithm selects the two largest groups (based on the total logic capacities of the FPGAs in the group) as seeds. It then iteratively selects the largest remaining subgroup, and combines it with whichever of the two seeds has the greatest ratio-cut with this group (i.e., largest $wire_{ij}/(size_i * size_j)$, where i is the largest remaining group, and j is one of the two seeds). Note that if the largest remaining group isn't connected to either of the seeds, we ignore it and move on, returning to it only once a group is successfully combined with a seed. The reasoning behind this combining algorithm is as follows: we start with the largest groups, since this will hopefully allow us to best balance the sizes of the two final groups created. We merge based on ratio cut, since we are trying to minimize the overall ratio cut. We do not combine unconnected groups, since (as discussed in the chapter introduction) all partitions created must be connected.

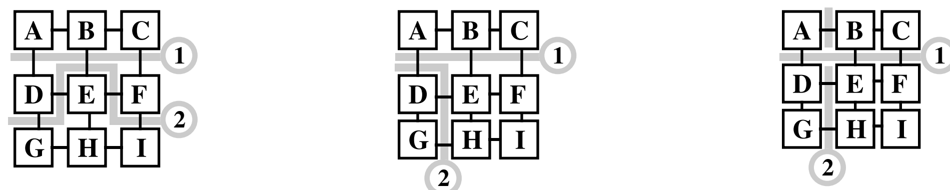

Figure 90. A failure of Global combining (left), and a comparison of ratio-cut metrics (center and right).

This combining algorithm can be turned into either a Global or a Local version. The Global version ignores partitions, and considers the entire topology at once. The groups considered are the sets of connected FPGAs after only applying the final cut. The Local version works on one partition at a time, and the groups considered are the sets of FPGAs in a partition that are connected after all the cuts (including the current one) are applied. The Global version produces better results, since it considers the entire topology at once, but is not guaranteed to successfully combine groups. For example, if we are partitioning the topology in Figure 88a, and have already found cut #1, we could end up breaking both partitions into multiple groups. The Local algorithm would ignore global concerns, and could generate the results in Figure 88b. While the Global algorithm has a good chance of generating the partitioning in Figure 88c, it might also fail to produce a good partitioning. One possibility is that it would simply find cut line #1 again, and the algorithm would make no progress. A second possibility is demonstrated in Figure 90 left. As shown, the second cut breaks the topology into two halves, and could be found by the Global algorithm. However, with the pre-existing cut #1, this breaks the lower partition into three groups, and is thus a failure. The Local algorithm will always ensure that a partition is broken into at most two groups.

A final issue in the basic algorithm is that of controlling the randomness of the process. Since we use a random process to select the cuts to make in the system, it is possible that any specific cut selected by the algorithm will be a poor choice. This is particularly true if the Local combining process is used. To deal with this, we perform multiple runs of the algorithm for each cut, and select the best at each step. Specifically, we make ten passes of the cut selection algorithm, as well as the combining algorithms (if necessary), to find the first cut. The best cut from all these runs is selected, and used as the starting point for ten runs to find the second cut. Each cut is selected from ten individual runs, and the best is used as the starting point for subsequent passes. Since we are trying to find the best ratio-cut partitioning, we could evaluate a cut based on a multi-partition ratio-cut metric [Chan93a]. This cost is shown in Equation 3.

$$\frac{1}{(k-1)}\sum_{i=1}^{k}\left(\frac{RoutingCapacity_i}{LogicCapacity_i}\right)$$

Equation 3. Multi-partition ratio-cut metric.

K is the current number of partitions, $RoutingCapacity_i$ is the amount of routing capacity connected between partition i and any other partition, and $LogicCapacity_i$ is the total logic capacity of partition i. The lower the ratio-cut, the better. One problem with this formulation is that it tends to penalize larger numbers of partitions. For example, the cuts shown in Figure 90 center have a ratio-cut cost of $(3-1)^{-1}(3/3 + 3/2 + 4/4) = 1.75$, and the cuts in Figure 90 right have a cost of $(4-1)^{-1}(3/2 + 2/1 + 3/2 + 4/4) = 2.0$. Thus, the cuts in Figure 90 center are preferred by the ratio-cut metric, but the cuts in Figure 90 right are actually better for our purposes. The reason for this is that the standard ratio-cut metric tends to favor less partitions, while for our purposes it is usually better to break a topology into a larger number of partitions, which reduces the total number of cuts necessary. To fix this, we divide the ratio-cut metric given above by k (the number of partitions) since this will tend to favor more partitions. This results in costs of 1.75/3 = .5833 for Figure 90 center, and 2.0/4 = .5 for Figure 90 right, which indicates that Figure 90 right has the preferred cuts. One final piece is necessary: it is possible in the ten runs that some cuts will be generated directly from the partitioning algorithm, some will result from the Global combining algorithm, and some from the Local combining algorithm. The Global algorithm produces better results than the Local algorithm, and results that need no combining are better than results from the Global algorithm. Thus, we always prefer results generated by the partitioning algorithm above any combining algorithm results, and we prefer results of the Global algorithm above results from the Local algorithm. This preference takes precedence over the ratio-cut metric.

Algorithm Extensions

While the algorithm described so far is capable of handling arbitrary partitioning situations, there are two extensions that can be made to greatly improve the results. These are parallelizing cuts, and clustering for multi-way partitioning.

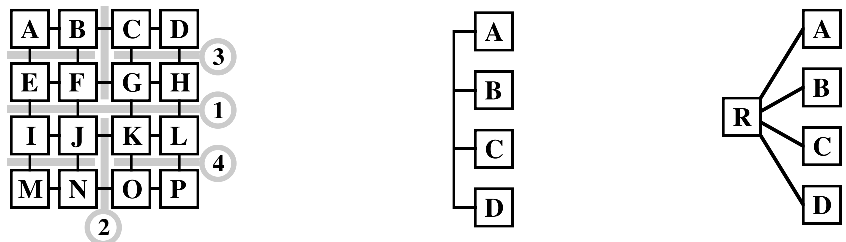

Figure 91. Example of parallelizeable cuts (left), and multi-way partitioning opportunities (center and right).

In parallelizing cuts, we observe that two or more cuts can be performed in parallel if they do not cut the same partition into more than two groups. For example, in Figure 91 left, given the set of cuts shown we would have to run four partitionings in series. We cannot combine cuts 1 and 2 since they cut the initial partition (the entire system) into four groups. We cannot combine cuts 2 and 3 since they cut partition **A-H** into four groups. We do not consider combining cuts that are not sequential since that would disrupt the order of partitionings. However, the last potential pair to combine, namely cuts 3 and 4, can be combined since they do not both cut the same partition. By combining these cuts, we only have to perform three cuts in series, potentially speeding up the partitioning process in a multiprocessing environment. To parallelize cuts, we consider each pair of sequential cuts, from earliest to latest, and combine all pairs that do not cut the same partition. Note that a similar algorithm could parallelize cuts to allow a quadrisection algorithm [Suaris87] to be applied. Quadrisection breaks a partition into four separate groups, while handling the connectivity constraints between the partitions. So, in Figure 91 left we could combine cuts 1 and 2 together, further accelerating the partitioning process.

The final extension we made to our algorithm is clustering to enable multi-way partitioning. As we mentioned earlier, the reason we do not use standard multi-way partitioning algorithms for the multi-FPGA partitioning problem is that we normally have to optimize for inter-FPGA routing capacity constraints, while standard multi-way algorithms do not allow inter-partition constraints. What these algorithms do optimize for is either the total number of nets connecting logic in two or more partitions (the net-cut metric), or the total number of partitions touched by each of these cut nets (the pin-cut metric). For example, if one net touches partitions **A** and **B**, and another touches **A**, **B**, and **C**, then the net-cut metric yields a cost of 2, and the pin-cut metric yields a cost of 5. It turns out that there are places in some topologies where a multi-way partitioning under the net-cut or pin-cut metric is exactly the right solution. For example, in the topology in Figure 91 center, which has only a set of buses connecting the individual FPGAs, there are no specific routing capacities between individual FPGAs. The only thing that is important is minimizing the number of nets moving between the partitions, since each net uses up one of the inter-FPGA buses, regardless of which or how many FPGAs it touches. In this situation, the net-cut metric is correct, and we can partition into all of the FPGAs simultaneously. Note that if there were other wires between a subset of the FPGAs in Figure 91 center we could not perform this 4-way partitioning, since there would be capacity constraints between individual FPGAs (most multi-FPGA systems with buses also have other connections, so we will ignore the net-cut model for multi-way partitioning). In the topology in Figure 91 right, there are four FPGAs connected to only a purely routing chip **R**. In this situation, it is necessary to limit the number of wires going between **R** and the other FPGAs, but there is no other limitation on the number or connectivity of the inter-FPGA nets. For example, if the wires in the topology had a capacity of one, it would make no difference if there was one net connecting all four partitions, or two connections between two different FPGAs (i.e., **A**-**B** and **C**-**D**). In this situation, which is common in two-level topologies (Chapter 5), a multi-way partitioning based on the pin-cut model is the proper way to partition.

We have the following algorithm for finding multi-way partitioning opportunities. Before we do any partitioning, we search for multi-way partitioning opportunities. We remove all routing-only nodes from the system, and find all maximally connected subcomponents of the topology. We examine in order (from smallest to largest in total capacity) each subcomponent, and examine if it can be multi-way partitioned. Specifically, for multi-way partitioning to be performed, this subcomponent must be connected to only one routing-only node. Also, removal of this routing node from the complete topology must break the topology into at least three connected components (if it didn't, the k-way partitioning that could be performed here would be 1-way or 2-way, and thus best left to the normal algorithm to discover). If these constraints can be met, we group together the original component, the routing node, and all but the largest of the other components found in the complete topology minus the one routing-only node. This cluster becomes a new node, which replaces all nodes being grouped, and all edges that were incident to grouped nodes are connected to the new node instead. The cluster node's logic capacity is equal to the total logic capacity of all nodes being grouped. This process of clustering multi-way partitioning opportunities continues until no more clusterings can be performed. During the partitioning process, once one of these clusters is broken off into a partition by itself, or with only other routing-only chips, we add the multi-way partitioning of this cluster to the list of partitionings to be performed. The cluster node is replaced with the nodes it clusters, and the algorithm is allowed to perform further partitioning of the nodes that were in the cluster. Note that if the first cut found causes a clustered node to be unclustered next, we make this multi-way partitioning the first cut performed.

Experiments

All of the experiments in this section were performed with the algorithms described in this chapter. The topologies are a mix of a set of existing multi-FPGA systems, as well as a few hypothetical mesh structures included to demonstrate some specific features of our algorithm. All processing occurred on a SPARC-10. For each cut, we chose the best of ten runs. The flow increment (the amount of capacity used up by each random route) was ten percent of the average capacity of the wires in the system, after all wires connecting the same destinations were combined.

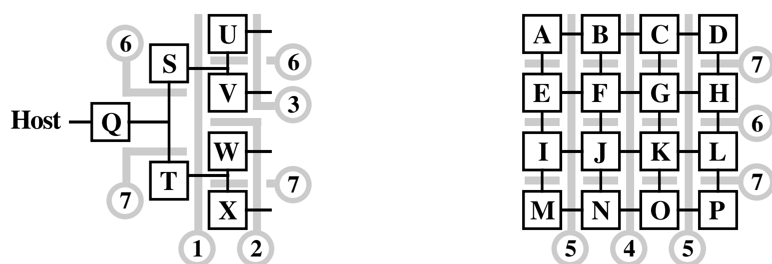

Figure 92. The DECPeRLe-1 board. Connections from **U**,**V**,**W**, and **X** going off to the right connect to each of **A**-**P**. Some connections are not shown.

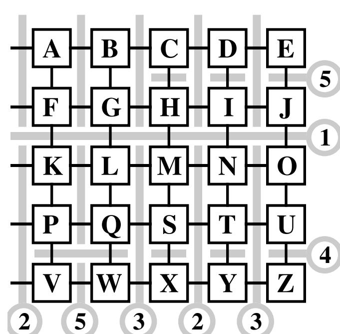

Figure 93. The NTT board. Connections going off the left side connect to the corresponding FPGA on the right side. All chips still connected are cut by the 6th cut line. Note that there is 50% more capacity on the horizontal wires than on the vertical ones.

The first three figures demonstrate our algorithm on three different current topologies. Figure 92 is the DECPeRLe-1 board [Vuillemin95]. Note that the connections dangling off of **U**-**X** are actually buses connecting to all the FPGAs **A**-**P** in the center. As can be seen, the algorithm first partitions three times through the left half, which are crossbars connected throughout the system plus the host interface. It then partitions up the main mesh twice (center), and finishes both halves in the last two cuts.

In Figure 93 is the NTT board [Yamada94], which is a mesh with wrap-around connections from left to right. While most of the cuts are reasonable, note cut #5. The algorithm actually attempted to do both a horizontal and a vertical cut at once. The partition **ABFG** is actually split several times in this step, but the combining algorithm reduces it to a single cut. Note that cut #6 is not depicted - it splits all remaining 2-FPGA partitions in half.

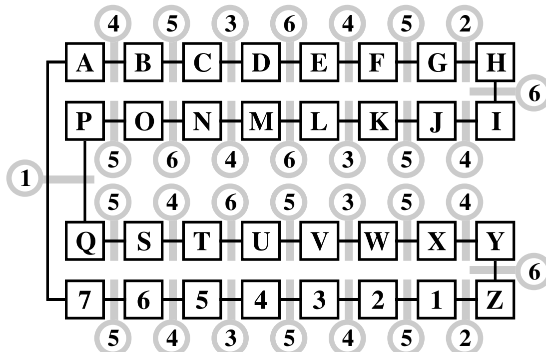

Figure 94. The Splash topology. Note that the cuts are somewhat imbalanced because of limited resources between FPGAs **A** and **7**.

In Figure 94 we partition the Splash board [Gokhale90]. Note that while there are 68 connections between most of the neighboring FPGAs, there are only 35 connections between **A** and **7**. Because of this, the cuts made in steps 2 and 3 are shifted over one FPGA from an even split, and it takes a total of 6 steps to subdivide the entire system. If we change the topology, putting 68 connections between **A** and **7**, the algorithm completes the partitioning in 5 steps, performing even splits in each step. Note that if we did not apply our parallelization of cuts technique to the original topology, it would actually require 17 separate partitionings.

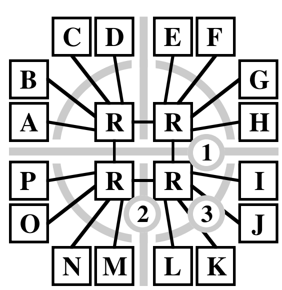

Figure 95. Demonstration of the clustering to discover k-way partitioning opportunities.

To demonstrate our method for finding k-way partitioning opportunities, we ran our algorithm on the topology in Figure 95 . This two-level topology is a 2x2 mesh of routing FPGAs, with each routing FPGA connected to four logic-bearing FPGAs. The algorithm clusters together the logic FPGAs connected to each routing FPGA. The first two cuts separate the routing FPGAs, and then the four 4-way partitionings can all be accomplished simultaneously (the latter was done with the help of the parallelization routines).

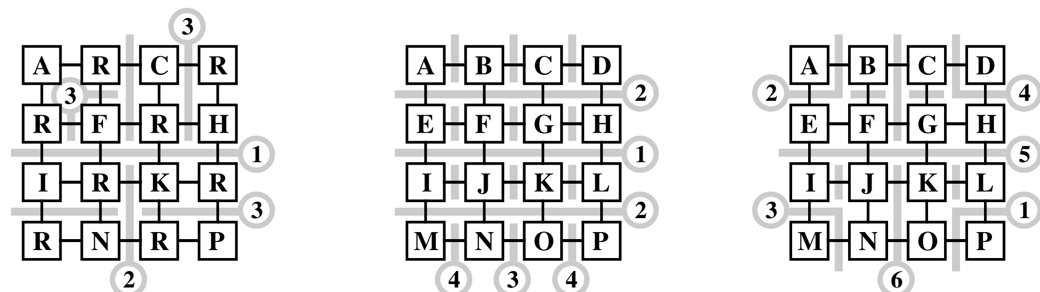

Figure 96. Demonstrations of how the algorithm reacts to different topological parameters. At left, the array is half routing-only chips (those marked with R). At center, a normal array with three times as much routing capacity on the horizontal edges as on the vertical edges. At right, all edges have equal capacity, but the corner four FPGAs have ten times as much logic capacity as the others (to simulate external connections).

As a final test of our system, we took a 4x4 mesh and varied some of the parameters we have discussed previously. Figure 96 left demonstrates routing-only nodes (labeled “R”). As can be seen, since there is no need to separate a logic-bearing node from routing-only nodes, the algorithm stops after 3 steps. In Figure 96 center, we show a topology where the horizontal edges have 3 times the capacity as the vertical edges. As might be expected, the algorithm adjusts by partitioning the vertical edges before it cuts the horizontal ones, since the excess capacity makes the horizontal edges much less critical. Finally, Figure 96 right assigns the same routing capacity to all edges, but gives the four corner FPGAs 10 times as much logic capacity as the other FPGAs (to simulate external connections for example). As can be seen, the algorithm decides it needs to deal with the connections to these FPGAs first, and isolates them in the first four cuts. It then cuts the rest of the topology. Note that the four unlabeled gray lines are cuts from step #7, and the remaining uncut edges are all cut in step #8.

Pseudocode and Computational Complexity

In this section we present the pseudocode for the logic partition ordering algorithm, as well as its computational complexity. The overall algorithm is given in Figure 97.

```
read in topology;
group nets connected to same set of chips, incrementing capacity;
cluster for multi-way partitioning;

While two logic-bearing FPGAs are in the same partition {
   compute probabilities;
   repeat 10 times {
      saturate network;
      if more than two groups are formed {
         apply global combining;
         if global combining failed {
            apply local combining;
         }
      }
   }
   select best of 10 cuts;
   if cluster is isolated {
      select multi-way partitioning of cluster;
      remove clustering;
   }
}
parallelize cuts;
```

Figure 97. Logic partition ordering algorithm.

```
remove routing-only nodes from topology;
find maximally connected components;
replace routing-only node;
foreach component C, smallest to largest {
   if C is connected only to one routing-only node R {
      remove node R from topology;
      if there are 3 or more maximally connected components {
         cluster R and all but largest connected component;
      }
      replace node R;
      restart clustering algorithm on clustered topology;
   }
}
```

Figure 98. Clustering algorithm for multi-way partitioning.

In [Yeh92] the algorithm that formed the basis of this work was presented, and it was shown that it has a complexity of O(R*M*N log N). N is the number of nodes and M the number of nets in the graph being partitioned. Note that for our purposes, M is actually the number of different types of nets, where all nets connecting exactly the same set of nodes are in the same category. Multiple nets in the same category in a multi-FPGA systems simply increase the capacity on the single edge that represents the entire category. R is the number of flow increments necessary to saturate the largest-capacity net in the circuit. This flow increment can easily be modified to always keep the value of R constant, and thus R can be ignored. The N log N is the time to find the shortest path between the random points in the original algorithm's real-valued edge length model. Because of our reformulation of the edge lengths, which allows constant-time access to the queue, the shortest-path search can be done in O(T), where T is the number of terminals in the graph (one terminal exists at each connection of nets to nodes in the graph). In any reasonable multi-FPGA system the number of terminals will be only a small constant factor larger than the number of nets in the system (almost all nets will be 2-terminal nets, with only a few large fanout nets). Thus, one cut in the system can be found in $O(M^2)$. All other portions of the algorithm are dominated by this runtime.

```
total_probability = 0.0;
foreach group G {
   cum = total capacity of nodes in group;
   cum_prob(G) = 0.0;
   foreach node N in group G
      cum_prob(G) = cum_prob(G) + (cum - capacity(N))*capacity(N);
   total_probability = total_probability + cum_probability(G);
}
foreach group G
   probability(G) = cum_probability(G) / total_probability;
```

Figure 99. Algorithm for computing probabilities for random route distribution. A group is the set of nodes that have been on the same side of all previous cuts. Probability(G) is the portion of random routes that should occur within a given group, and is based on the capacity of the nodes in that group. Note that a group with only one logic-bearing node will have a 0 probability of routes.

```
set flow on all edges to 0;
set length on all edges to 1;
do forever {
   randomly pick group G based on probabilities;
   randomly pick source node S in group G based on capacities;
   randomly pick different destination node D in group G
      based on capacities;
   find shortest-path P from S to D using queue structure;
   if no path exists
      return topology;
   increment flow on all edges in P;
   recompute edge length on all edges in P;
}
```

Figure 100. Network saturation algorithm.

The combining algorithm will have to be run multiple times to create a complete partitioning ordering for a multi-FPGA system. While for many situations there will only need to be O(log N) cuts in the multi-FPGA system, it is possible to need up to O(N) cuts. This is true for a linear array of FPGAs, where each of the

N-1 links in the array will need to be cut separately. Thus, the complete runtime for this algorithm is $O(N*M^2)$. While a cubic complexity may not be ideal, one must realize that N and M are unlikely to grow that quickly in future. While FPGAs will increase in internal capacity and external I/Os, neither of these numbers increases N nor M (remember that more nets going to the same location simply increase the flow on the edge in the graph being partitioned, and greater capacity simply alters the distribution of random sources and sinks of routes). Thus, improvements in multi-FPGA systems are likely to come mostly from improved chips, while the number of FPGAs and the types of routing are unlikely to change much in the future.

```
partition topology based only on current cuts;
find largest group L, second-largest group S;
repeat until no groups other than L and S remain {
   find largest group G other than L and S which is connected
      to L and/or S;
   L_ratio = bandwidth(L, G) / (size(L) * size (G));
   S_ratio = bandwidth(S, G) / (size(S) * size (G));
   if S_ratio >= L_ratio
      combine S and G;
   else
      combine L and G;
   if size(S) > size (L)
      swap(L, S);
}
```

Figure 101. Global combining algorithm.

```
foreach group G in topology, ignoring current cut
   apply global combining to group G only;
```

Figure 102. Local combining algorithm.

```
max = total number of cuts in partition ordering;
base = 1;
target = 2;
while next <= max {
   if cut(base) and cut(next) do not subdivide the same group {
      combine cut(base) and cut(next);
      next = next + 1;
   } else {
      base = next;
      next = next + 1;
   }
}
```

Figure 103. Cut parallelization algorithm.

Conclusions

In this chapter we have considered applying standard partitioning algorithms to multi-FPGA systems. We have detailed an algorithm that determines the order to perform bipartitioning by finding the critical bottlenecks, while ensuring that all partitions created are connected. We have also detailed a method for increasing the parallelism, and decreasing the required run time, of partitioning in a multiprocessing environment. This technique is also capable of finding multi-sectioning opportunities. Finally, we have included a method of determining when multi-way partitioning can be used. In this way, we have developed an integrated method for best harnessing the numerous existing bipartitioning, multi-sectioning, and multi-way partitioning algorithms. The algorithm is efficient, and handles arbitrary topologies and heterogeneous FPGAs.

As mentioned earlier, an automatic method for generating partition orderings has several benefits. For current and future multi-FPGA systems, it allows a large amount of flexibility and extendibility to be built into the system, since the software can cope with arbitrary topologies. Thus, small systems can be grouped into even larger machines, arbitrary chips and connections can be introduced, and the topology itself can be dynamically modified. Automatic mapping software can be generated that requires little user intervention, since the designer is not required to determine the location and ordering of cuts to be made. Also, failure recovery and incremental update capabilities can be included easily, since the software can partition to an arbitrary subset of the full system.

By combining the logic partition ordering algorithm from this chapter with the bipartitioning work from Chapter 10, a complete logic partitioning algorithm for multi-FPGA systems can be developed. It is fast and efficient, and automatically adapts to an arbitrary topology. In Chapter 12 we discuss a topology-adaptive algorithm for pin assignment, which handles part of the inter-FPGA routing problem for multi-FPGA systems.