ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 32-38, February, 1995.

Logic Partition Orderings for Multi-FPGA Systems

Scott Hauck, Gaetano Borriello
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

One of the critical issues for multi-FPGA systems is developing software tools for automatically mapping circuits. In this paper we consider one step in this process, partitioning. We describe the task of finding partition orderings, i.e., determining the way in which a circuit should be bipartitioned so as to best map it to a multi-FPGA system. This allows multi-FPGA partitioners to harness standard partitioning techniques. We develop an algorithm for finding partition orderings, which includes a method for increasing parallelism in the process, as well as for including multi-sectioning and multi-way partitioning algorithms. This method is very efficient, and capable of handling most of the current multi-FPGA topologies.

Introduction

In the time since they were introduced, FPGAs have moved from being viewed simply as a method of implementing random logic in circuit boards to being a flexible implementation medium for many types of systems. Logic emulation tasks, where ASIC designs are simulated on large FPGA-based structures [Varghese93], have greatly increased simulation speeds. Software subroutines have been hand-optimized to FPGAs to speed up inner loops of programs [Bertin93], and work has been done to automate this process [Wazlowski93]. FPGA-based circuit implementation boards have been developed for easier project construction in electronics education [Chan92]. Custom-computing devices built from FPGAs have demonstrated significant performance improvements over standard general-purpose processors [Lopresti91, Arnold93].

While some very impressive results have been achieved by the hand-mapping of circuits to FPGA-based systems, developing a completely automatic system for mapping to these structures is important to achieving more widespread utility. A good example of this is the Quickturn systems [Varghese93], which offer an integrated approach for mapping ASIC circuits. One of the most difficult steps in the mapping process to a multi-FPGA system is logic partitioning. In logic partitioning, the circuit to be mapped is split into pieces small enough to fit into the individual FPGAs. The partitioner needs to ensure that no FPGA is assigned more logic than it can handle, and it must also restrict the inter-FPGA routing to fit the available resources. It turns out that in today's systems it is the routing resource constraint that is the most restrictive [Babb93], greatly limiting the achieved mapping sizes.

Partitioning has been an area of active research for at least the last 25 years. There have been numerous approaches tried, and several promising approaches have emerged [Donath88]. While many of these have been focused on bipartitioning, or breaking into exactly two partitions [Kernighan70, Fiduccia82, Krishnamurthy84, Hagen92, Bui94, Riess94, Yang94, Hauck95], there has been work on extending them to multi-way partitioning [Sanchis89, Chan93]. 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, 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 suffers from the problem that the first cuts made may be significantly better than later cuts, we can use this to our advantage. If the first 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 argues 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, current systems are offering much greater flexibility. For some, it is primarily the ability to connect together small systems to handle much larger problems [Babb93, Varghese93], 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 in the types and capacities of resources that can be inserted [Aptix93, Giga94, Koch94, Zycad94]. There is also the possibility of much greater flexibility, with systems that allow very complex and customized topologies to be built [Hauck94]. 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 second important trend is towards systems being built by users whose goal is to harness FPGA technology to solve problems in other domains. We have seen multi-FPGA systems developed to examine compilation techniques and processor structure [Lewis93], multi-processor architectures [Barroso94], and neural-net computations [Ferrucci94], and future applications are abundant. There is a real need to develop flexible mapping solutions for these systems, solutions that do not require the user to be an expert in every step of the mapping process. Partitioners for such systems need to be as general as possible, and can benefit greatly from automatic methods to find partition orderings.

The final trend of interest to us 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 when the user requires 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 paper 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 in order to best map a circuit onto a system of chips with fixed interconnections.

Partition Ordering Challenges

Before we discuss some solutions, we need to define some terms. In this paper, 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.

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. However, 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.

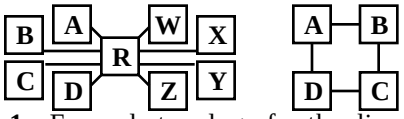

Figure 1. Example topology for the discussion of connectedness of partitions (left), and for the discussion of multiple partition creation (right).

The issue of connected groupings is demonstrated in figure 1 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 comes up 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 1 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, say 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 cap_1 is the total logic capacity of all FPGAs on one side of a cut, and cap_2 is the capacity on the other side, then out of $(cap_1+cap_2)^2$ signals, $2*cap_1*cap_2$ 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, $(2*cap_1*cap_2)/(cap_1+cap_2)^2$ is the percentage of all nets crossing this cut. Since (cap_1+cap_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. 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 $wire_{12}/(cap_1 * cap_2)$. 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 that reflects the most constrained routing (assuming random placement of logic), while ensuring that the partitions formed are connected. It turns out that 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. The iteration of choosing random points and routing 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. Thus, a net whose capacity has been used up is removed from the system, and no longer connects between any of its terminals. 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 1 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 of all, 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. Thus, we can implement the queue as an array of 1025 elements (with element j representing paths of length $j+i*1025$, for $i \geq 0$). In this way, insertions and deletions are *O(1)*.

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 an FPGA or other device (such as an Field-Programmable Interconnects (FPIC) [Aptix93, I-Cube94], a routing-only device) is meant to be used purely for routing, there should be no sources or sinks assigned to that chip. Also, the capacity of the individual chips should also be considered. If one FPGA in the system has twice the logic capacity of another, then twice as much logic will likely be assigned to it, and thus 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 FPGAs in the system. Note that if a route starts and ends in the same FPGA it can be ignored. One final random routing consideration is the handling of external inputs and outputs. In many cases there will be specific ports on the multi-FPGA system that are used for communication with the system's 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.

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 2a. 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 2b). There are two problems with this. First of all, there is the issue of terminal propagation [Dunlop85]. When we do 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 2d, 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 2b, 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 2c, **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 2c, and perform them simultaneously (both cuts labeled 2 in the figure would be considered as one single cut, so all cuts shown could be done simultaneously). These algorithms would be able to recognize a situation such as the mapping in figure 2d, 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 2a, 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 considered. 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 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 cut.

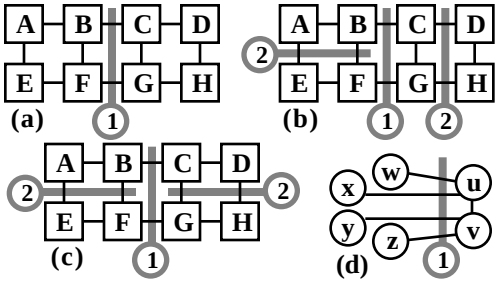

Figure 2. Examples of multiple cuts in a topology (cuts are in gray, with lower numbers indicating earlier cuts). If the initial cut (a) is used to split the entire system, uncoordinated subcuts (b) may occur. Keeping the system together would generate more coordinated cuts (c). An example circuit is also shown (d). Note that in this paper FPGAs will be represented by squares, and logic to be mapped by circles.

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*. 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 introduction) all partitions created must be connected.

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 2a, 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 2b. While the Global algorithm has a good chance of generating the partitioning in figure 2c, 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 3 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.

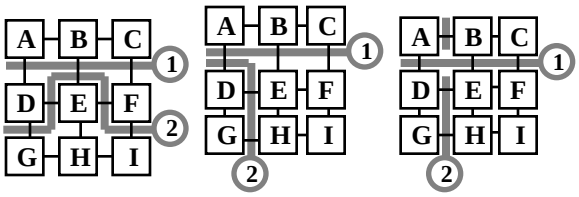

Figure 3. A failure of Global combining (left), and a comparison of ratio-cut metrics (center and right).

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 evaluate a cut based on a multi-partition ratio-cut metric [Chan93]. The cost is:

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

where *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 3 center

have a ratio-cut cost of $(3-1)^{-1}(3/3 + 3/2 + 4/4) = 1.75$, and the cuts in figure 3 right have a cost of $(4-1)^{-1}(3/2 + 2/1 + 3/2 + 4/4) = 2.0$. Thus, the cuts in figure 3 center are preferred by the ratio-cut metric, but the cuts in figure 3 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 3 center, and 2.0/4 = .5 for figure 3 right, which indicates that figure 3 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.

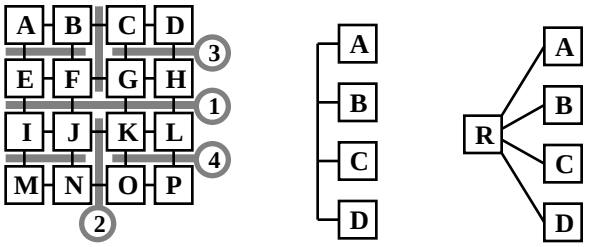

Figure 4. Example of parallelizeable cuts (left), and multi-way partitioning opportunities (center and right).

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 4 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 4 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 4 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 4 center we could not perform this 4-way partitioning, since there would be capacity constraints between individual FPGAs (it turns out that most multi-FPGA systems with buses [Thomae91, Bertin93, Lewis93] also have other connections, so we will ignore the net-cut model for multi-way partitioning). In the topology in figure 4 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 FPIC-based systems [Aptix93]), 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. This group 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 partitioning found causes a clustered node to be unclustered next, we make this multi-way partitioning the first cut performed.

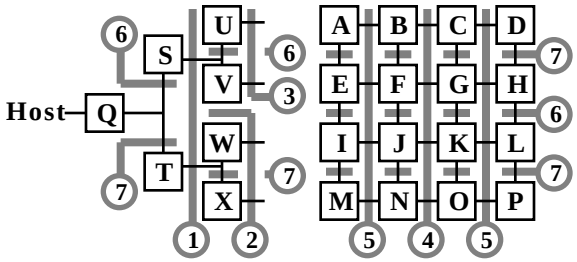

Figure 5. The DECPeRLe-1 board. Connections from **U**,**V**,**W**, and **X** going off to the right connect to each of **A**-**P**. Some connections are not shown.

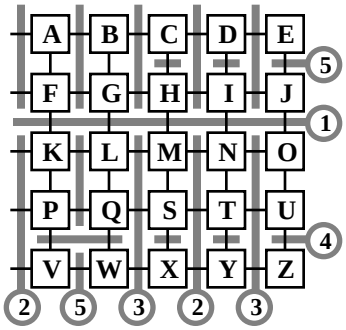

Figure 6. The NTT board. Connections going off the left side connect to the corresponding FPGA on the right side. All chips still connected are cut by the 6th cut line. Note that there is 50% more capacity on the horizontal wires than on the vertical ones.

Experiments

All of the experiments in this section were performed with the algorithms described in this paper. 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.

In figures 5, 6, and 7, we demonstrate our algorithm on three different current topologies. Figure 5 is the DECPeRLe-1 board [Bertin93, Keaney93]. 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 6 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 attempts 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.

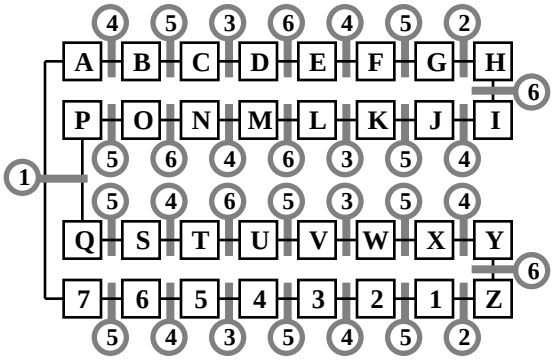

Figure 7. The Splash topology. Note that the cuts are somewhat imbalanced because of limited resources between FPGAs **A** and **7**.

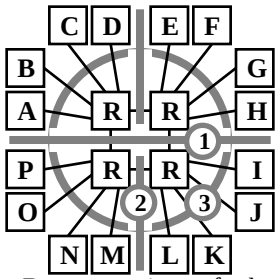

Figure 8. Demonstration of the clustering to discover k-way partitioning opportunities.

In figure 7 we partition the Splash board [Lopresti91]. 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 fix this, 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.

To demonstrate our method for finding k-way partitioning opportunities, we ran our algorithm on the topology in figure 8. There 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 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).

Conclusions

In this paper 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 multi-processing 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 addition capabilities can be included easily, since the software can partition to an arbitrary subset of the full system.

Acknowledgments

This research was funded in part by the Advanced Research Projects Agency under Contract N00014-J-91-4041. Scott Hauck was supported by an AT&T Fellowship.

References

Aptix Corp., "Data Book", San Jose, CA, February 1993.

J. M. Arnold, "The Splash 2 Software Environment", *FCCM*, pp. 88-93, 1993.

J. Babb, R. Tessier, A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators", *FCCM*, pp. 142-151, 1993.

S. Bapat, J. P. Cohoon, "Sharp-Looking Geometric Partitioning", *Euro-DAC*, pp. 172-176, 1991.

L. Barroso, S. Iman, J. Jeong, K. Öner, K. Ramamurthy, M. Dubois, "The U.S.C. Multiprocessor Testbed: Project Overview", USC CENG Technical Report CENG-94-15, August 1994.

P. Bertin, D. Roncin, J. Vuillemin, "Programmable Active Memories: a Performance Assessment", *Symposium on Integrated Systems*, pp.88-102, 1993.

T. N. Bui, B. R. Moon, "A Fast and Stable Hybrid Genetic Algorithm for the Ratio-Cut Partitioning Problem on Hypergraphs", *DAC*, pp. 664-669, 1994.

P. K. Chan, M. Schlag, J. Y. Zien, "Spectral K-Way Ratio-Cut Partitioning and Clustering", *Symposium on Integrated Systems*, pp. 123-142, 1993.

P. K. Chan, M. Schlag, M. Martin, "BORG: A Reconfigurable Prototyping Board Using Field-Programmable Gate Arrays", *FPGA '92*, pp. 47-51, 1992.

W. E. Donath, "Logic Partitioning", in *Physical Design Automation of VLSI Systems*, B. Preas, M. Lorenzetti, Editors, Menlo Park, CA: Benjamin/Cummings, pp. 65-86, 1988.

A. E. Dunlop, B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits", *IEEE Trans. on CAD*, Vol. CAD-4, No. 1, pp. 92-98, January 1985.

A. Ferrucci, M. Martin, T. Geocaris, M. Schlag, P. K. Chan, "ACME: A Field-Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network", *FPGA'94*, 1994.

C. M. Fiduccia, R. M. Mattheyses, "A Linear-Time Heuristic for Improved Network Partitions", *DAC*, pp. 241-247, 1982.

Giga Operations Corporation, "G-800 VL-Bus Board", Berkeley, CA, 1994.

L. Hagen, A. B. Kahng, "New Spectral Methods for Ratio Cut Partitioning and Clustering", *IEEE Trans. on CAD*, Vol. 11, No. 9, pp. 1074-1085, September 1992.

S. Hauck, G. Borriello, C. Ebeling, "Springbok: A Rapid-Prototyping System for Board-Level Designs", *FPGA'94*, 1994.

S. Hauck, G. Borriello, "An Evaluation of Bipartitioning Techniques", *Chapel Hill Conference on Advanced Research in VLSI*, 1995.

I-Cube, Inc., "The FPID Family Data Sheet", Santa Clara, CA, February 1994.

R. A. Keaney, C. H. Lee, D. J. Skellern, J. Vuillemin, M. Shand, "Implementation of Long Constraint Length Viterbi Decoders using Programmable Active Memories", *Australian Microelectronics Conference*, pp. 52-57, 1993.

B. W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning of Electrical Circuits", *Bell Systems Technical Journal*, Vol. 49, No. 2, pp. 291- 307, February 1970.

G. Koch, U. Kebschull, W. Rosenstiel, "A Prototyping Environment for Hardware/Software Codesign in the COBRA Project", *Third International Workshop on Hardware/Software Codesign*, pp. 10-16, 1994.

B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks", *IEEE Trans. on Computers*, Vol. C-33, No. 5, pp. 438-446, May 1984.

D. M. Lewis, M. H. van Ierssel, D. H. Wong, "A Field Programmable Accelerator for Compiled-Code Applications", *ICCD '93*, pp. 491-496, 1993.

D. P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays", *Advanced Research in VLSI 1991: Santa Cruz*, pp. 139-152, 1991.

B. M.. Riess, K. Doll, F. M. Johannes, "Partitioning Very Large Circuits Using Analytical Placement Techniques", *DAC*, pp. 646-651, 1994.

K. Roy, C. Sechen, "A Timing Driven N-Way Chip and Multi-Chip Partitioner", *ICCAD*, pp. 240-247, 1993.

L. A. Sanchis, "Multiple-Way Network Partitions", *IEEE Trans. on Computers*, Vol. 38, No. 1, pp. 62-81, 1989.

P. R. Suaris, G. Kedem, "Standard Cell Placement by Quadrisection", *ICCD*, pp. 612-615, 1987.

D. A. Thomae, T. A. Petersen, D. E. Van den Bout, "The Anyboard Rapid Prototyping Environment", *Advanced Research in VLSI 1991: Santa Cruz*, pp. 356-370, 1991.

J. Varghese, M. Butts, J. Batcheller, "An Efficient Logic Emulation System", *IEEE Trans. on VLSI*, Vol. 1, No. 2, pp. 171-174, June 1993.

G. Vijayan, "Partitioning Logic on Graph Structures to Minimize Routing Cost", *IEEE Trans. on CAD*, Vol. 9, No. 12, pp. 1326-1334, December 1990.

M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh, "PRISM-II Compiler and Architecture", *FCCM*, pp. 9-16, 1993.

Y.-C. Wei, C.-K. Cheng, "Towards Efficient Hierarchical Designs by Ratio Cut Partitioning", *ICCAD*, pp. 298-301, 1989.

K. Yamada, H. Nakada, A. Tsutsui, N. Ohta, "High-Speed Emulation of Communication Circuits on a Multiple-FPGA System", *FPGA'94*, 1994.

H. Yang, D. F. Wong, "Efficient Network Flow Based Min-Cut Balanced Partitioning", *ICCAD*, 1994.

C. W. Yeh, C. K. Cheng, T. T. Lin, "A Probabilistic Multicommodity-Flow Solution to Circuit Clustering Problems", *ICCAD*, pp. 428-431, 1992.

Zycad Corporation, "Paradigm RP", Fremont, CA, 1994.