Chapter 12. Pin Assignment

Introduction

A theme running throughout this thesis has been the need for the highest performance automatic mapping solution possible, especially one that can adapt to arbitrary topologies. In Chapter 10 we presented a fast and efficient algorithm for logic bipartitioning. Chapter 11 takes this algorithm and automatically applies it to arbitrary multi-FPGA system topologies. In this chapter we present a topology-adaptive pin assignment algorithm. By applying this algorithm, the overall mapping quality is improved, and the time it takes to place and route the FPGAs in the system is reduced. Current systems can require placement and routing times of seventeen hours or more on a uniprocessor system.

In this chapter we will examine the global routing stage of the automatic mapping process. It is important to remember during the following sections the context we are working in: we are mapping to a fixed structure of FPGAs, where wiring between the pins of the FPGAs are fixed, and while the logic has been partitioned to the individual FPGAs, these individual FPGA mappings have not yet been placed nor routed.

Global Routing For Multi-FPGA Systems

The global routing phase of mapping to multi-FPGA systems bears a lot of similarity to routing for individual FPGAs. Just as in single FPGAs, global routing needs to route on a fixed topology, with strictly limited resources, while trying both to accommodate high density mappings and minimize clock periods.

The obvious method for applying single-FPGA routing algorithms to multi-FPGA systems is to view the FPGAs as complex entities, explicitly modeling both internal routing resources and pins connected by individual external wires (Figure 104 left). A standard routing algorithm would then be used to determine both which intermediate FPGA to use for long distance routing (i.e., a signal from FPGA **A** to **D** would be assigned to use either FPGA **B** or **C**), as well as which individual FPGA pins to route through. Unfortunately, this approach will not work. The problem is that although the logic has already been assigned to FPGAs during partitioning, the placement of logic into individual logic blocks will not be done until the next step, FPGA placement. Thus, since there is no specific source or sink for the individual routes, standard routing algorithms cannot be applied.

The approach we will take here is to abstract entire FPGAs into single nodes in the routing graph, with the arcs between the nodes representing bundles of wires. This solves the unassigned source and sink problem mentioned above, since while the logic hasn't been placed into individual logic blocks, partitioning has assigned the logic to the FPGAs. It also simplifies the routing problem, since the graph is much simpler, and similar resources are grouped together (i.e., all wires connecting the same FPGAs are grouped together into a single edge in the graph). Unfortunately, the routing algorithm can no longer determine the individual FPGA pins a signal should use, since those details have been abstracted away. For example, in Figure 104 left, logic function **a** in FPGA **A** may best be placed in the lower right corner, and logic function **b** in FPGA **B**'s lower left corner. Thus, a connection between **a** and **b** would best be placed on one of the lower wires connecting these FPGAs. A global routing algorithm could not determine this fact. It is this problem, the assignment of interchip routing signals to FPGA I/O pins, that this chapter addresses.

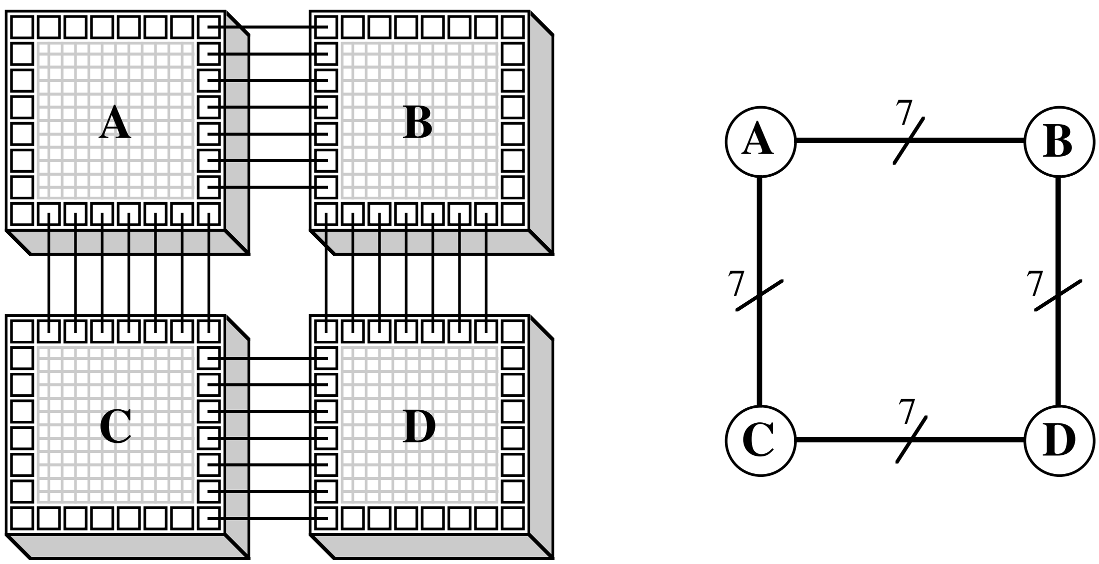

Figure 104. Two views of the inter-FPGA routing problem: as a complex graph including internal resources (left), and a more abstract graph with FPGAs as nodes (right).

Pin Assignment For Multi-FPGA Systems

Currently, the only work done on multi-FPGA pin assignment has been for the restricted case where no two logic-bearing FPGAs are directly connected (Chapter 9). For crossbars, hierarchical crossbars, and two-level topologies, where routing-only chips handle all connections between logic-bearing FPGAs, these algorithms works well. However, many systems map logic to most or all of their FPGAs (Chapter 5), and are beyond the scope of this algorithm. Also, as pointed out in Chapter 9, there is no way to adapt the pin assignment techniques from other technologies (such as ASIC routing channels, general routing for cell-based designs, and printed circuit board routing). This is because these algorithms optimize for features that are fixed in the multi-FPGA system domain, yet ignore the issues critical to multi-FPGA systems.

One solution to the pin assignment problem is quite simple: ignore it. After global routing has routed signals through intermediate FPGAs, those signals are then randomly assigned to individual pins. While this simple approach can quickly generate an assignment, it gives up some optimization opportunities. A poor pin assignment can not only result in greater delay and lower logic density, but can also slow down the placement and routing software, which must deal with a more complex mapping problem.

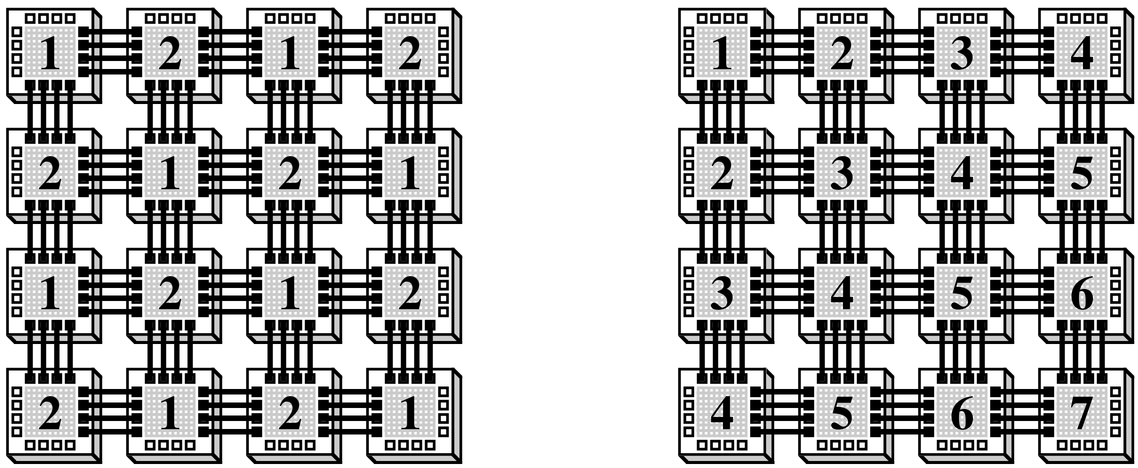

Figure 105. Checkerboard (left) and wavefront (right) pin assignment placement orders.

A second approach is to allow the FPGA placement tool to determine its own assignment. This requires that the placement tool allow the user to restrict the locations where an I/O pin can be assigned (a feature available in tools such as the Xilinx APR and PPR placement and routing tools [Xilinx94]). With such a system, I/O signals are restricted to only those pin locations that are wired to the proper destinations. Once the placement tool determines the pin assignment for one FPGA, this assignment is propagated to the attached FPGAs. It is important to note that this does limit the number of placement runs that can be performed in parallel. Specifically, since the assignment from one FPGA is propagated to adjacent FPGAs only after that entire FPGA has been placed, no two adjacent FPGAs can be placed simultaneously. Since the placement and routing steps can be the most time-consuming steps in the mapping process, achieving the greatest parallelism in this task can be critical. An algorithm for achieving the highest parallelism during placement, while allowing the placement tool to determine the pin assignment, is to find a minimum graph coloring of the FPGAs in the routing graph. Since the structure of a multi-FPGA system is usually predefined, the coloring can be precomputed, and thus the inefficiency of finding a graph coloring is not important (techniques introduced later in this chapter will be able to handle topologies without a predefined structure). Then, all the FPGAs assigned the same color can be placed at the same time, since any FPGAs that are adjacent cannot be assigned the same color. For example, in a four-way mesh (every FPGA is connected to the FPGA directly adjacent horizontally and vertically), the FPGAs could be placed in a checkerboard pattern, with half handled in the first iteration, and half in the second (Figure 105 left). Note that since the pin assignment will be determined during placement, the FPGA routing can be run in parallel with other placements or routings.

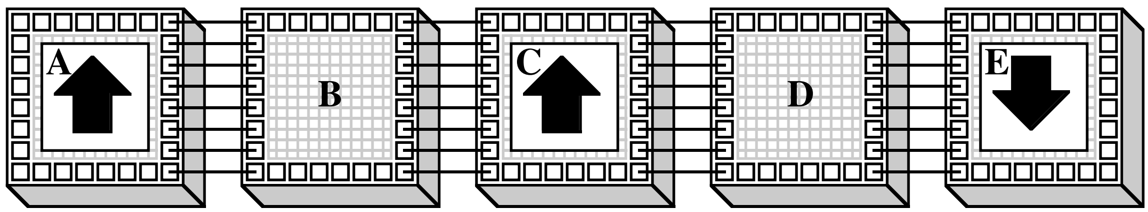

Figure 106. Example problem with the Checkerboard algorithm.

One problem with the maximum parallelism (or "checkerboard") method just described is that while the pin assignment for the FPGAs placed first will be very good, since the placer is mostly free in its placement choices, other FPGAs may be placed fairly poorly. For example, consider the mapping of a systolic circuit to a linear array of FPGAs (see Figure 106). If allowed to map freely, the algorithm will map in a certain manner (the up arrows), or the mirror image around the horizontal (the down arrow). Since the individual FPGAs are mapped independently, some will choose one orientation, while others will choose the other. The problem is that there is a good chance that one of the unmapped FPGAs will be left with some neighbors in one orientation, and others in the other orientation (FPGA D). These FPGAs will have their mapping task complicated by the pin assignment imposed on them by their neighbors.

There is an alternative approach to the checkerboard algorithm, which trades longer mapping run times for better results. The idea is to make sure that FPGAs are not mapped completely independently of one another. In the first step, a single FPGA is placed. The FPGA that is most constrained by the system's architecture (i.e., by special global signals or external interface connections) is normally chosen, since it should benefit most from avoiding extra pin constraints. In succeeding steps, neighbors of previously placed FPGAs are chosen to be placed next, with as many FPGAs placed as possible without simultaneously placing interconnected FPGAs. For example, in a 4-way mesh, where the first FPGA routed is in the upper-left corner, the mapping process would proceed in a diagonal wave from upper-left to lower-right (Figure 105 right). In this way, mappings "grow" from a single "seed" assignment, and will be more related to one another than in the checkerboard approach, hopefully easing the mapping tasks for all of the FPGAs.

Unfortunately, even this "wavefront" placement order may not generate good pin assignments. Most obviously, while the individual FPGA placements attempt to find local optimums, global concerns are largely ignored. For example, while pairs of signals that are used together in an FPGA will be placed together in the mapping, the fact that two of these pairs are used together in a neighbor will not be discovered, and these pairs may be assigned pin locations very distant from each other.

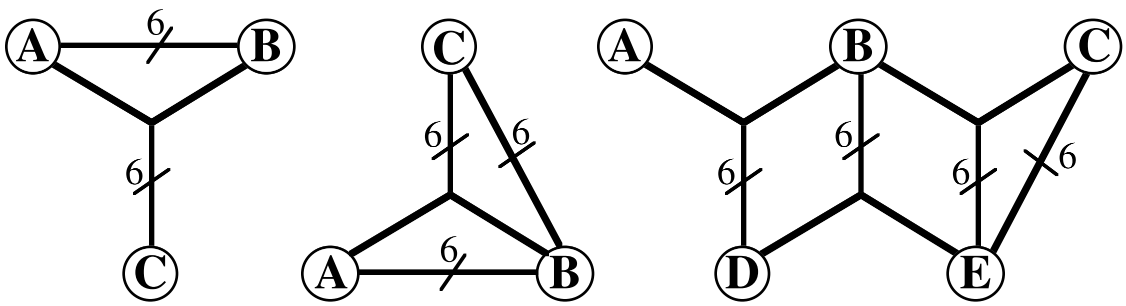

Figure 107. Examples of FPGA topologies that cannot be handled by sequential placement pin assignment techniques. Each of these topologies can be found as subcomponents in current systems, with all of these occurring in [Lewis93], and all but the topology at right in both [Thomae91, Vuillemin95].

Even worse than the speed and quality problems of both the checkerboard and wavefront methods (hereafter referred to jointly as "sequential placement methods") is the fact that there are some topologies that cannot be handled by these methods. Specifically, when the wires that connect FPGA pins are allowed to connect more than two FPGAs there is the potential for conflicts to arise, conflicts that can cause assignment failures with sequential placement methods. For example, consider the topology in Figure 107 left. Assume that the logic we wish to map to this topology has one connection between **A** and **C**, one between **B** and **C**, and six between **A** and **B** (Note that while this and other examples in this paragraph are specifically crafted to best show the underlying problems, each of these specific examples represent more general situations, and most wire and mapping connection counts can be changed without fixing these problems). Since all three FPGAs in this topology are directly connected, both the wavefront and checkerboard approach would place these FPGAs sequentially. If FPGA **A** is placed first, it could assign five of its connections to **B**, as well as its single connection to **C**, on the three-destination wires connecting **A**, **B**, and **C**. This means that there is no longer any way to route the connection between **B** and **C**, and the assignment fails. The same thing can happen when FPGA **B** is placed first. In the specific case of the topology at left, we could place FPGA C first, and avoid the conflicts described previously. Unfortunately, topologies such as the one in Figure 107 center have no order that is guaranteed to work. Assume that for this topology we wish to map a circuit with one connection between **A** and **C**, seven between **A** and **B**, and seven between **B** and **C**. In this situation at least one connection between each pair of FPGAs must be placed on a three-destination wire. However, regardless of the order, the first FPGA placed can use all of the three-destination wires for its own signals, causing the assignment to fail.

Instead of trying to find a placement order that will work, we could instead try to restrict the choices available to the placement tools. For example, in the mapping to the topology in Figure 107 center we know that each FPGA pair must have at least one connection assigned to the three-terminal wires. We could ensure that the placement tool generates a successful mapping by reserving wires for these signals. That is, if we placed FPGA **B** first, one of the three-destination wires would be reserved for a connection between **A** and **C** by disallowing the placement of signals from **B** onto the corresponding pin location. However, not only is this decision arbitrary, generating lower quality mappings since we have no method to pick a good wire to reserve, but it can also require significant effort to determine which and how many wires to reserve. For example, in the topology in Figure 107 right, a connection between **B** and **E** could be assigned either to a **BCE** or a **BDE** wire. If we do nothing to reserve space for it, connections between other FPGAs can fill these wires, making the assignment fail. We cannot reserve space on both wires, since all the capacity in the system may be required to successfully handle the mapping. However, to determine which wire to reserve requires the system to examine not only FPGA **B**'s and **E**'s connections, but also **D**'s, **C**'s, and **A**'s connections, since congestion on one wire can ripple through to the wires between **B** and **E**. Thus, determining the wires to reserve for one FPGA's connections can involve examining most or all FPGAs in the system. Note that while some of these topologies may seem contrived, topologies similar to the one in Figure 107 center are fairly common, and arise when a local interconnect is augmented with global connections, connections that include adjoining FPGAs.

Force-Directed Pin Assignment For Multi-FPGA Systems

As we have shown, pin assignment via sequential placement of individual FPGAs can be slow, cannot optimize globally, and may not work at all for some topologies. What is necessary is a more global approach which optimizes the entire mapping, while avoiding sequentializing the placement step. In the rest of this chapter, we will present one such approach and then give a quantitative comparison with the approaches presented earlier.

Intuitively, the best approach to pin assignment would be to place all FPGAs simultaneously, with the individual placement runs communicating with each other to balance the pin assignment demands of each FPGA. In this way a global optimum could be reached, and the mapping of all FPGAs would be completed as quickly as any single placement could be accomplished. Unfortunately, tools to do this do not exist, and even if they did, the communication necessary to perform this task would be prohibitive. Our approach is similar to simultaneous placement, but we will perform the assignment on a single machine within a single process. Obviously, with the placement of a single FPGA consuming considerable CPU time, complete placement of all FPGAs simultaneously on a single processor is impractical, and thus simplification of the problem is key to a workable solution.

Our approach is to use force-directed placement of the individual FPGAs [Shahookar91]. In force-directed placement, the signals that connect logic in a mapping are replaced by springs between the signal's source and each sink, and the placement process consists of seeking a minimum net force placement of the logic ("net force" indicates that forces in opposite directions cancel each other. Note that the spring force is calculated based on the Manhattan distance). To find this minimum net force configuration, and thus minimize wirelength in the resulting mapping, the software randomly chooses a logic block and moves it to its minimum net force location. This greedy process continues until a local optimum is found, at which point the software accepts the current configuration.

Force-directed placement may seem a poor choice for pin assignment, and is generally felt to be inferior to simulated annealing for FPGA placement. Two reasons for this are the difficulty force-directed placement has with optimizing for goals other than wirelength, and the inaccuracy of the spring approximation to routing costs. However, as will be shown, force-directed placement can handle all of the optimization tasks involved in pin assignment, and the spring metric is the key to efficient handling of multi-FPGA systems.

As implied earlier, we will not simply place individual FPGAs, but will in fact use force-directed placement simultaneously on all FPGAs in the system. We make two alterations to the basic force-directed algorithm: first, we do not restrict logic blocks to non-shared, discrete locations, but instead allow them to be freely placed into any location in the FPGA with no regard to congestion or physical logic block boundaries (however, I/O pins ARE constrained to exact pin locations, and I/Os cannot share a single pin location). Second, we assume that all logic blocks are always at their optimal positions. While the second alteration is simply a change in the movement order of nodes, the first change could cause a significant loss in accuracy. However, the resulting degradation of the local optimum will turn out to be more than made up for by the ability to seek the global optimum. Also, while this assumption allows logic blocks to share physical locations, something that cannot be done in valid FPGA mappings, our force-directed placement is used only to determine I/O pin assignments, and the logic blocks will be placed in a later step by standard FPGA placement tools.

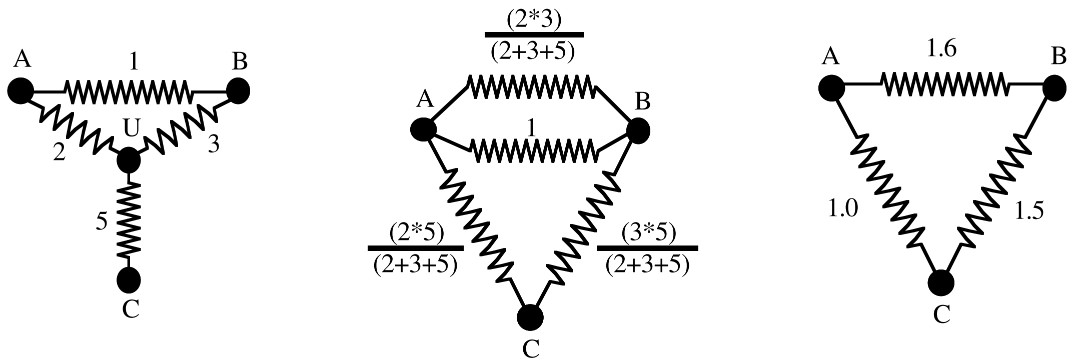

Figure 108. Example of spring simplification rules. Source circuit at left has logic node U replaced at center, and any springs created in parallel with others are merged at right.

By making the two assumptions just given, we now have the opportunity to greatly simplify the mapping process. We can examine the system of springs built for the circuit mapping, and use the laws of physics to remove nodes corresponding to logic functions, leaving only I/O pins. As shown at the end of this chapter, as well as in the example of Figure 108, the springs connected between an internal logic node and its neighbors can be replaced with a set of springs connected between the node's neighbors while maintaining the exact same forces on the remaining nodes. By repeatedly applying these simplification rules to the logic nodes in the system, we end up with a mapping consisting only of I/O pins, with spring connections that act identically to the complete mapping they replace. In this way, we simplify the problem enough to allow the pin assignment of a large system of FPGAs to be performed efficiently. Note that this spring replacement process can take a significant portion of the algorithm's run time, primarily because the replacement time is dependent upon the number of springs connected to the node being removed, and the removal of a node causes all of its neighbors to become connected. This effect can be mostly alleviated by removing nodes in increasing order of the number of springs connected to that node.

There are some details of the force-directed algorithm that need to be considered here. First, there are two ways to perform individual node moves in the algorithm. A simple approach is to pick a node and swap it with whichever node yields the least overall net force. A more complex approach is to perform a ripple move, where a node is moved to its optimum location, and any node occupying this location is then moved. This continues until the destination of the current node being moved is empty, at which point the move as a whole is evaluated. If it is a beneficial move, it is accepted; if not, it is rejected. Note that during this process the nodes already moved are flagged, and are not allowed to move again until a complete ripple move is over. While it is tempting to go with the simplicity of the swap move, there are some topologies for which swap moves yield poor results. Specifically, consider an FPGA with a total of 3 three-terminal wires (Figure 109), one to A and B, one to B and C, and one to A and C. Now consider a mapping onto this structure with a separate connection to each of the neighbor FPGAs. In this situation it is impossible to make a correct swap move, since no two connections have the same two wires as possible assignments.

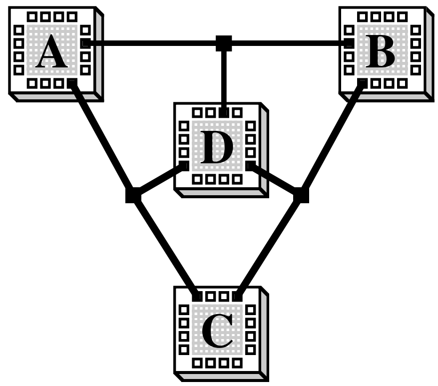

Figure 109. Topology for demonstrating the advantage of ripple moves instead of swap moves.

Another issue with the algorithm has to do with the starting position. While we would like to construct an initial, correct assignment for each wire, this can be very complex. As we indicated in the discussion of the topology in Figure 107 right, determining onto which wire a signal can be placed can require the examination of most or all connections in the system. An easier answer is to start with all wires unassigned, and always accept all valid ripple moves that start with an unassigned connection. In this way the mechanisms already required to perform moves of previously assigned pins can be easily extended to handle unassigned pins as well.

Efficiency in the calculation of the individual moves is another important problem. To determine the best possible destination of a node during a ripple move it is necessary to calculate the change in net force for assigning the connection to each wire that connects the FPGAs the signal moves between. This is necessary because there is not necessarily any relationship between the location of an I/O pin on one FPGA and the location of the pin the wire goes to on another FPGA (in fact, as described in Chapter 7, there is some advantage to be gained by scattering pin connections). Because of this, it is unlikely that there would be any method for pruning the search space of possible move destinations. This can be very troubling, especially because the node replacement method described above causes nodes to be connected to most or all other nodes in the same FPGA. However, as shown in Equation 4, the standard force equation (at left) can be reformulated so that the information from all springs connected to a node can be combined at the beginning of a move, and the calculation of the net force at each location requires only a small amount of computation. Another method for speeding up the process is to remember which connections have been starting points of ripple moves that were determined not to be beneficial, and thus not moved. Then, the algorithm will not try these moves again until another pin on one or more of that connection's FPGAs has been successfully moved.

$$\sum_{i=1}^{n} SpringConst_i \times \left(pos_i - pos_{node}\right) = \left(\sum_{i=1}^{n} SpringConst_i \times pos_i\right) - \left(\sum_{i=1}^{n} SpringConst_i\right) \times pos_{node}$$

Equation 4. Reformulation of the standard force equation.

A final extension necessary is software support to help avoid the problems found in the sequential approaches on some topologies. Just as the topologies in Figure 107 caused problems for the other approaches, they could cause difficulties with the force-directed approach. Imagine that we have the topology in Figure 107 left, and there are six connections between **A** and **C**, and one connection between **A** and **B**. At some point, all the three-destination wires will be occupied by connections between **A** and **C**. If we begin a ripple move with the connection between **A** and **B**, its best position might be on one of the three-destination wires. Unfortunately, while these wires must be occupied by the connections between **A** and **C**, and cannot accept the connection between **A** and **B**, this fact will not be determined until 5 of the 6 connections between **A** and **C** have been moved as part of this ripple move. At that point all of the wires will have connections assigned to them which have been flagged as part of the current ripple move, and the final connection between **A** and **C** will have no allowable assignment. While we can undo the ripple move, there is no way to tell the connection from **A** to **B** that it needs to try one of the two-terminal wires.

Our solution to the problems with the topologies in Figure 107 is to add the notion of equivalence classes to the system. That is, every wire in the system is a member of exactly one equivalence class, and each class contains only those wires that connect exactly the same FPGAs. For example, the topology in Figure 107 center would have three equivalence classes, one for the two-terminal wires between **A** and **B**, another for the two-terminal wires between **B** and **C**, and a final class for the three-terminal wires connecting **A**, **B**, and **C**. For each equivalence class we maintain a count of the number of wires within the equivalence class that have no connection assigned to them. Then, whenever a connection wishes to be assigned to a wire in an equivalence class, it can only do so if either the equivalence class has an unassigned wire, or if one of the (unflagged) connections already assigned to the class can be moved to another equivalence class (moving this connection to another class requires the same check to be performed on that class). Note that much of this information can be cached for greater efficiency, since if at one point in a ripple move an equivalence class cannot accept a connection, it cannot accept a connection at any future point during that move. Thus, in the previous example where it took several moves to determine that the connection between **A** and **B** shouldn't be moved onto a three-terminal wire in Figure 107 left, this fact would be immediately apparent from the equivalence classes. This would allow the search for possible moves to be pruned, and the connection would be forced to move to one of the two-terminal wires between **A** and **B**, the only legal assignments for that connection.

Table 16. Quantitative speed comparison table. Numbers in parentheses are normalized to the force-directed algorithm's results. Time is the time on multiple processors, while Uniproc. is the time on a single machine.

System		Splash		NTT		DECPeRLe-1	
Mapping		DNA Comparator		Telecom		Calorimeter	
Force -	Time	34:25		8:45		29:14	
	Uniproc.	12:05:12		19:31		2:43:18	
Wave -	Time	5:35:40	(9.7530)	20:50	(2.3810)	2:11:33	(4.5000)
	Uniproc.	13:24:58	(1.1100)	22:36	(1.1580)	3:10:02	(1.1637)
Checker -	Time	1:07:00	(1.9467)	12:22	(1.4133)	1:51:24	(3.8107)
	Uniproc.	12:37:36	(1.0447)	21:55	(1.1230)	2:55:24	(1.0741)
Rand -	Time	1:00:18	(1.7521)	8:18	(0.9486)	30:47	(1.0530)
	Uniproc.	17:45:56	(1.4698)	20:59	(1.0751)	2:56:44	(1.0823)

As mentioned earlier, our force-directed approach is capable of handling most of the optimization issues necessary for pin assignment. First, since little information about the individual FPGA architectures is necessary beyond the location of the I/O pins, it is very easy to port this software to different FPGAs. In fact, our current system allows different FPGA architectures and sizes to be intermingled within a single multi-FPGA system. Note that the locations defined in the software's chip descriptions need not be exactly the same as the real chip's geometry. In fact, our definitions of the Xilinx chips used in the comparisons below space pins at the corners closer together than others to reflect the lower expected congestion at these locations. We can also accommodate crossbar chips and FPICs, chips where minimizing wirelength is unnecessary, by setting the spring constant of all springs within such chips to zero. The assignments to pins going to the crossbar can also be removed after the pin assignment is complete. In this way, connections that require pin assignment can be assigned by our tool, while connections to the crossbar can be left unconstrained. More details on this can be found in Chapter 9.

Table 17. Continuation of quantitative speed comparison from Table 16. Numbers in parentheses are normalized to the force-directed algorithm's results. Time is the time on multiple processors, while Uniproc. is the time on a single machine.

System		Virtual Wires		Marc-1	
Mapping		Palindrome		Logic Sim. Processor	
Force -	Time	13:18		26:36	
	Uniproc.	1:44:37		3:38:11	
Wave -	Time	22:01	(1.6554)	Failure	
	Uniproc.	1:27:34	(0.8370)		
Checker -	Time	13:56	(1.0476)	Failure	
	Uniproc.	1:28:19	(0.8442)		
Rand -	Time	10:37	(0.7982)	26:38	(1.0013)
	Uniproc.	1:28:35	(0.8467)	4:36:32	(1.2674)

Table 18. Routing resource usage comparison table. Numbers in parentheses are normalized to the force-directed algorithm's results.

System	Splash	NTT	DECPeRLe-1	Virtual Wires	Marc-1
Force	143690.9	7292.2	45159.4	41255.4	102728.6
Wave	146424.8	7345.6	45661.2	41111.8	Failure
	(1.0190)	(1.0073)	(1.0111)	(0.9965)	
Checker	149860.5	7714.6	46522.5	41900.7	Failure
	(1.0429)	(1.0579)	(1.0302)	(1.0156)	
Random	156038.2	7854.8	48713.4	42225.3	108236.1
	(1.0859)	(1.0772)	(1.0787)	(1.0235)	(1.0536)

To optimize for delay, spring constants can be altered. Although our tool currently does not optimize critical paths, and assigns a spring constant of one to all non-crossbar springs, the tool could easily be extended to increase the spring constant on critical paths. This would cause critical path I/Os to be clustered closer together at the chip edge, improving performance.

Another important consideration is support for predefined pin assignments for some special connections. Specifically, special connections such as clocks, reset signals, fixed global communication lines, and host interfaces need to be assigned to specific pin locations. Handling these constraints in the pin assignment tool is trivial, since the support necessary to flag nodes that participate in the current ripple move can be extended to permanently freeze a connection in a specified position. Note also that any special chip features in an FPGA, such as the source of global distribution lines, can be handled similarly by defining a special pin location at the special feature site, and always preassigning the feature node to that point. For signals within FPGAs that are globally distributed via special resources, resources that make wirelength minimization irrelevant, the corresponding springs can simply be removed, or have their spring constants set to zero. Finally, there are other limited resources within an FPGA that might require special processing. For example, in Xilinx FPGAs [Xilinx94] there are a limited number of horizontal longlines which can be used as wide multiplexers or wired ANDs. To handle these features, the corresponding nodes in the circuit graph could be left unremoved, and force-directed placement could be applied to these elements as well (though these resources would of course require different potential positions than the pin locations). By doing this, a more accurate assignment could be performed without greatly affecting runtimes. Also, for the specific case of the Xilinx longlines, where the horizontal portion of the Manhattan distance is unimportant (the longlines stretch the width of the chip), these nodes could be specially marked, and the force calculation could be set up to properly ignore the horizontal component.

Comparison of Pin Assignment Approaches

To compare the various pin assignment approaches, we tested each approach on mappings for five different current systems. These include a systolic DNA comparison circuit for Splash [Lopresti91], a telecommunications circuit for the NTT board [Yamada94], a calorimeter circuit for DECPeRLe-1 [Vuillemin95], a palindrome circuit for the Virtual Wires Emulation system [Tessier94], and a RISC processor configured for logic simulation on the Marc-1 system [Lewis93]. The pin assignment systems compared are: “random”, which randomly assigns connections to wires; “checkerboard” and “wavefront”, which use sequential placement runs to do pin assignment; and “force”, which uses the force-directed pin assignment technique described above. The results include both mapping time and resulting wirelength (Table 16, Table 17, and Table 18). The time is CPU time on a SPARC-10, and includes the time to perform pin assignment as well as individual FPGA place and routes, and assumes enough machines are available to achieve maximum parallelism. Uniprocessor time, the time it takes to complete all of these tasks on a single machine, is also included. Note that the sequential placement techniques only sequentialize the placement step, while routing is allowed to be performed in parallel with subsequent placements and routings. Also, these placement attempts are begun as soon as possible, so that for example in the checkerboard algorithm, we do not need to wait to begin placing FPGAs of one color until all FPGAs of the previous color are placed, but instead can proceed with a placement once all of its direct neighbors in the previous color(s) have been completed. Also, the random approach is assumed to be able to generate a pin assignment in zero time, so its time value is simply the longest time to place and route any one of the individual FPGAs. The wirelength is determined by summing up all source to sink delays for all signal in the FPGAs. While this is not exact, since some portions of a multi-destination route will be included more than once, it gives a reasonable approximation of the routing resource usage of the different systems.

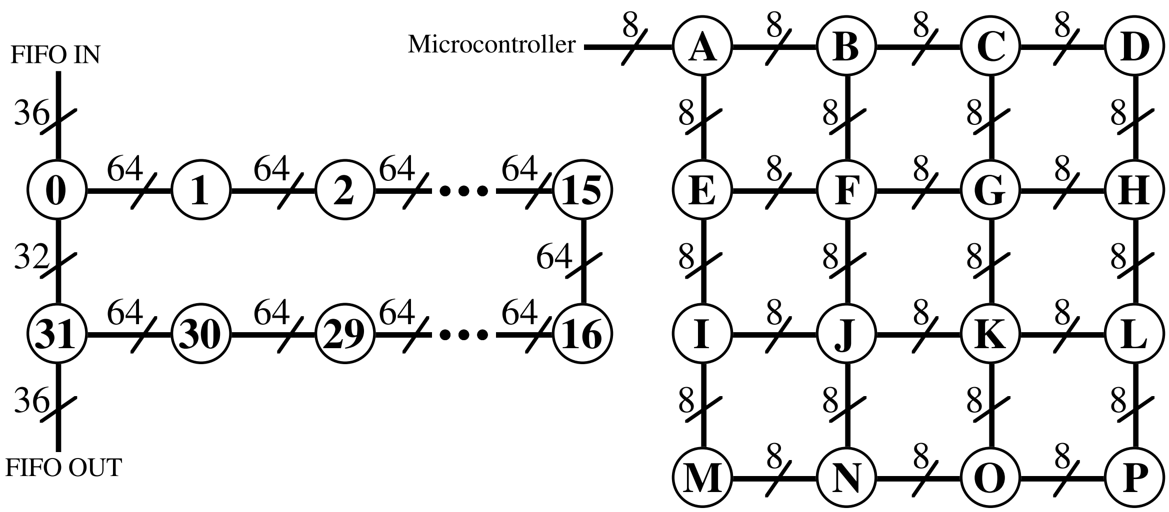

Figure 110. The Splash (left) and Virtual Wires (right) topologies. Note that system features not used in the example mappings, as well as global communication wires and clocks, are omitted.

The Splash system [Gokhale90] is a linear array of Xilinx 3090 FPGAs (Figure 110 left), with memories attached to some of the communication wires. As expected, Random does poorly on the average wirelength, with Checkerboard and Wavefront each doing successively better. However, the Force-Directed approach does the best, since it is able to optimize the mapping globally. More surprising is the fact that the Force-Directed approach is actually faster than all the other approaches, including the Random approach, in both parallel and uniprocessor times. The reason for this is that the poor pin assignment generated by the random approach significantly increases the routing time for some of its FPGAs, up to a factor of more than 4 times as long. Note that the time to perform the force-directed pin assignment does not dominate for either this or any other mapping. For the DNA mapping it is only 9% of the overall runtime, for Palindrome and Telecom it is 6%, and for Calorimeter it is 8%, though for the Marc-1 mapping it is 31%.

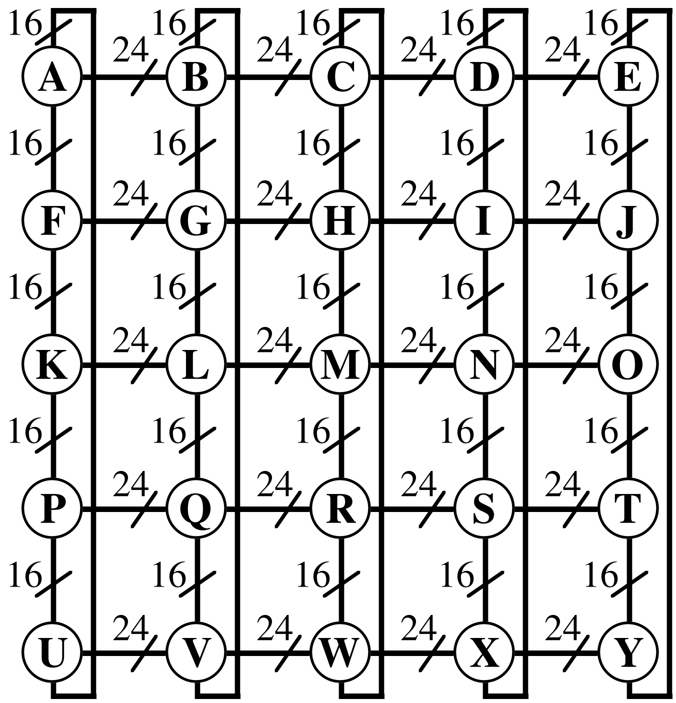

Figure 111. The NTT topology.

The results are almost identical for the DECPeRLe-1 board [Vuillemin95], a four by four mesh of Xilinx 3090 FPGAs with seven additional support FPGAs, and the NTT board [Yamada94], a five by five mesh of 3042 Xilinx FPGAs with wrap-around connections on the top and bottom edges (Figure 111). However, the Checkerboard approach takes much longer on the DECPeRLe-1 board than it did on the Splash board, making it only about 15% faster than the Wavefront approach. The reason is that the DECPeRLe-1 board, which has 23 FPGAs, is only 19-colorable, leaving very little parallelism for the Checkerboard algorithm to exploit.

The Virtual Wires board [Tessier94] is a 4-way mesh of Xilinx 4005 FPGAs, with the connections in a given direction scattered around the FPGA's edge (Figure 110 right). The results for this mapping are less favorable than for the DNA comparator and the calorimeter, with the Wavefront method actually generating a slightly better assignment than the Force-Directed approach. However, we believe the reason for this poor performance is the special techniques applied to the mapping prior to pin assignment. The Virtual Wires system [Babb93, Selvidge95] attempts to overcome FPGA I/O limitations by time-division multiplexing the chip outputs. Thus, each chip I/O is connected to a shift chain, which is connected to the actual I/O signals of the mapping. Unfortunately, the choice of which wires to combine is done arbitrarily, without regard to the mapping structure. Thus, instead of combining signals that are close together in the logic, signals from very distant portions of an FPGA's mapping may be merged. Because of this, there is very little structure left in the mapping, and there is little pin assignment can do to improve things. We believe that a more intelligent combining strategy, possibly driven by the spring constants generated by our spring simplification approach, would generate better mappings, mappings that can also be helped by good pin assignments.

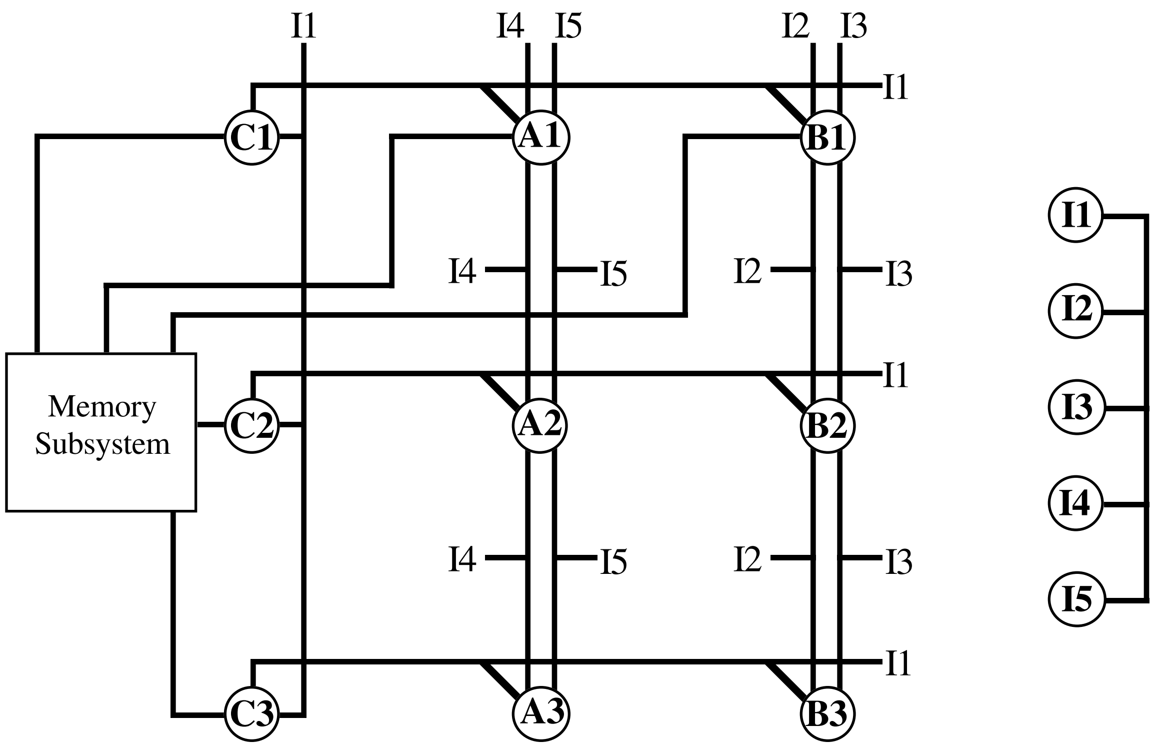

Figure 112. Schematic of the Marc-1 board. The entire board consists of two copies of the subsystem at left, and one copy of the subsystem at right. Labels without circles indicate connections to the corresponding FPGA circle. Note that system features not used in the example mappings, as well as global communication wires and clocks, are omitted.

The Marc-1 board [Lewis93] is a complex system of Xilinx 4005 FPGAs, as shown in Figure 112. As mentioned earlier, this board includes all topologies from Figure 107, topologies that make mapping failures possible in sequential placement approaches regardless of placement order. We attempted to use these sequential approaches four times each on this system, but each time it failed because of resource conflicts (note that even had they succeeded, the topology is only 5-colorable, and the Wavefront method would sequentialize 9 placement runs, which in both cases would greatly increase runtimes). However, our force-directed approach easily handled this topology, and generated a good assignment. Note that while the force-directed result is somewhat less of an improvement than the other examples led us to expect, this may be due to the fact that only 51% of the logic pins are assignable, with the others fixed by memory interfaces and other fixed global signals.

Table 19. Unconstrained pin assignment distances. Numbers in parentheses are normalized to the force-directed algorithm's results. The NTT results show the "optimal" value is worse than what we achieve. This is due to the random nature of the placement process.

Splash	138899.9	(0.9667)
NTT	7380.7	(1.0121)
DECPeRLe-1	37373.3	(0.8276)
Virtual Wires	39789.1	(0.9645)
Marc-1	91267.4	(0.8884)

While the previous comparisons demonstrate the advantages of the force-directed approach over the simple approaches described earlier, it would be useful to find out how close to optimal the force-directed approach is. In general, we can expect that the best a pin assignment can do is to equal the placement qualities the placement and routing tools can achieve without fixed pin assignments. That is, if we place and route the individual FPGAs and only require that logic pins be assigned to chip pins going to the correct neighbor (though we do not require that connected logic pins on neighboring FPGAs match up, that they occupy connected FPGA pins), this is probably as good as any pin assignment can hope to achieve, and probably better than is achievable. We did exactly this experiment, and the results are in Table 19. As can be seen, our pin assignments achieve results within 3.5% of "optimal" on half of the examples, with the geometric mean of the lower bound results being 93% of the force-directed results. In one case our results are even better than the lower bound results; however, we believe this is due mainly to the random nature of the placement process. It is unclear exactly how much can be inferred from these results, especially since it is not clear any approach can achieve results as good as those shown in Table 19. One important observation to note is that most likely not much improvement (at least in routing resource usage) can be achieved beyond what our algorithm provides.

Pseudocode and Computational Complexity

In this section we present pseudocode for the force-directed pin assignment algorithm, as well as a discussion of the computational complexity. The overall algorithm is presented in Figure 113.

```
read in topology and circuit description;
                     // Apply spring reduction rules
foreach FPGA F {
   foreach logic node L in F, from low to high connectivity {
      T = total spring constants of springs incident to L;
      foreach pair of L's neighbors N and M {
         spring(N, M) = spring(N, M)
            + (spring(L, N)*spring(L, M))/T;
      }
      foreach of L's neighbors N
         remove spring(N, L);
   }
}

create equivalence classes;
while moves possible {
   randomly pick IO signal S;
   perform ripple move for signal S;
   if S was originally unassigned, or ripple move is beneficial
      accept ripple move;
   else
      undo ripple move;
}
```

Figure 113. The force-directed pin assignment algorithm.

```
S = IO signal starting ripple move;
Compute constants from Equation 4 for each FPGA connected to S;
foreach wire W connecting between S's FPGAs {
   if equivalence classes allow S to be assigned to W {
      compute net force of S assigned to W;
   }
}
pick W with lowest net force;
temporarily assign S to W;
alter equivalence class so that S cannot be moved again in ripple;
if W is unoccupied {
   return change in force of assigning S to W;
} else {
   perform ripple move for signal assigned to W;
   return change in force of assigning S to W plus rest of ripple;
}
```

Figure 114. Algorithm for applying ripple moves.

Unfortunately, it is difficult to quantify the computational complexity of the force-directed pin assignment algorithm. The primary problem is bounding the number of iterations for the algorithm to find a minimum net-force state. It is possible to believe that degenerate situations exist where the system will take a significant number of moves to find a local minimum. Also, it is possible for a single ripple-move to move every I/O node in the system. Specifically, imagine a system that only has three-terminal wires, and a mapping with two-terminal connections. A connection between FPGAs A and B might move to a three-terminal wire between A, B, and C. This will displace a connection from B to C, which goes to a three-terminal wire between B, C, and D, displacing a connection from C to D. Thus, the worst-case complexity is significant. Also, the spring reduction process can be $O(N^3)$ in the worst-case. Specifically, imagine a mapping to an FPGA where every logic and I/O node is connected to a single signal (such as a reset or clock signal that does not use the FPGA's global distribution resources). When the first node is replaced, all pairs of nodes in the FPGA will become connected. Thus, every subsequent step in the process will have to modify a spring between each pair of nodes in the FPGA, resulting in $O(N^3)$ performance. Note that it is unlikely that such a situation will occur, since most huge-fanout nets will be handled by global routing resources, and thus will be removed from consideration by the spring reduction algorithm. For real circuits, the runtimes should be more on the order of linear or quadratic because of the sparse, localized connection patterns. Thus, while the worst-case complexity of this algorithm is quite large, and hard to bound, the program runs efficiently on current systems, and should scale reasonably to future machines.

Conclusions

As we have shown in this chapter, pin assignment for multi-FPGA systems is a difficult problem that has not been previously studied. We presented some approaches for using existing tools, as well as a new force-directed algorithm that can help improve mapping quality. Placement and routing with the force-directed approach is almost always faster than the random approach, by up to 43%, and almost always delivers superior routing resource usage, by up to an 8% decrease in total resource usage in the entire system. We have also shown that this is within at least 7% of optimal. Given these results, it is clear that pin assignment improves the mapping process for multi-FPGA systems.

Several possible extensions have been mentioned earlier for improving mapping quality, and these are worth further study. More importantly, the spring cost metric is not necessarily tied to force-directed placement, but could instead be used within a simulated annealing algorithm. While we believe simulated annealing would not yield significant improvements, since the cost metric is so abstracted from the real situation, and since simulated annealing is so time-consuming, a study of its actual benefits should be performed. More interesting is the approach mentioned above for improving Virtual Wires mappings by guiding wire combining by the spring forces our system generates. We hope that such an approach would not only improve the overall placements achieved by Virtual Wires, but also show the benefits of good pin assignments in this domain.

Spring Replacement Rules

As discussed earlier, we wish to reduce the complexity of the force-directed placement algorithm by replacing all springs touching non-I/O nodes with equivalent springs only involving I/O nodes. To do this, we iteratively remove individual internal nodes from the system, and replace the attached springs with new springs between the removed node's neighbors.

Since we are using a Manhattan distance metric, the forces along the X and Y dimension are independent, and are given in Equation 5.

$$F_x = C \times (X_1 - X_2) \quad F_y = C \times (Y_1 - Y_2)$$

Equation 5. Spring forces in the X and Y direction.

C is the spring constant, and (X_1,Y_1) and (X_2,Y_2) are the locations of the nodes connected by the spring. There are two simplification rules necessary for our purposes: parallel spring combining, and node removal. For parallel springs, springs which connect the same two endpoints, the springs can be merged into a single spring whose spring constant is the sum of the parallel springs.

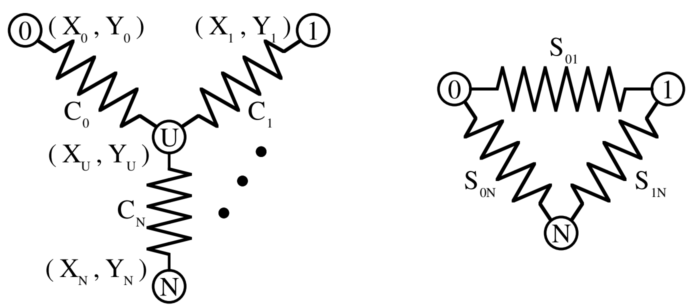

Figure 115. Spring system before node **U** is removed (left), and after (right).

In node removal, we remove an internal node, and replace the springs connected to that node with springs connected between the node's neighbors. As shown in Figure 115, we have node **U** with $N+1$ neighbors labeled **0**…**N** (note that if **U** has only one neighbor, the node and the spring can simply be removed, since it will never exert force on its neighbor). Node i is located at (X_i,Y_i), and is connected to node **U** by a spring with spring constant C_i. As discussed earlier, we assume that internal nodes are always placed at their optimal location, which is the point where the net force on the node is zero. Thus, we can calculate the location of node **U** as shown in Equation 6 and Equation 7 (note that from now on we will work with only the X coordinates of all nodes. Similar derivations can be found for the Y coordinates as well).

$$0 = \sum_{i=0}^{n} C_i \times \left(X_i - X_U\right) = \left(\sum_{i=0}^{n} C_i \times X_i\right) - X_U \times \sum_{j=0}^{n} C_j$$

Equation 6. Total forces on node **U**, assuming **U** is placed at its optimal location.

$$X_U = \frac{\sum_{i=0}^{n} C_i \times X_i}{\sum_{j=0}^{n} C_j}$$

Equation 7. Optimal X position of node **U** expressed in terms of its neighbors' positions.

To replace the springs connected to node **U**, we must make sure the new springs provide the same force as the old springs. So, we start with the force equation from Equation 5, and substitute in the location found in Equation 7. The results are Equation 8-Equation 11.

$$F_k = C_k \times (X_U - X_k) = \frac{C_k \times \left(\sum_{i=0}^{n} C_i \times X_i\right)}{\sum_{j=0}^{n} C_j} - C_k \times X_k$$

Equation 8. Substitution of Equation 7 into Equation 5.

$$F_k = \frac{C_k \times \left(\sum_{i=0}^{n} C_i \times X_i\right) - C_k \times X_k \times \sum_{l=0}^{n} C_l}{\sum_{j=0}^{n} C_j}$$

Equation 9. Simplification of Equation 8.

$$F_k = \frac{\left(\sum_{i=0}^{n} C_k \times C_i \times X_i\right) - \left(\sum_{l=0}^{n} C_k \times C_l \times X_k\right)}{\sum_{j=0}^{n} C_j}$$

Equation 10. Simplification of Equation 9.

$$F_k = \sum_{i=0}^{n} \left(\frac{C_k \times C_i}{\sum_{j=0}^{n} C_j} \times (X_i - X_k) \right)$$

Equation 11. Simplification of Equation 10.

From Equation 11 it is now clear how to replace the springs incident on node **U**. We can replace all of these springs, and insert a new spring between each pair of neighbors of node **U**. The new spring between nodes *I* and *K* will have a spring constant S_{ik} as given in Equation 12.

$$S_{jk} = \frac{C_k \times C_i}{\sum_{j=0}^{n} C_j}$$

Equation 12. Spring constants for spring replacement, derived from Equation 11.