Northwestern University, Dept. of ECE, Technical Report, 1996.

Software Technologies for Reconfigurable Systems

Scott Hauck
Department of ECE
Northwestern University
Evanston, IL 60208 USA
hauck@ece.nwu.edu

Anant Agarwal
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
agarwal@lcs.mit.edu

Abstract

FPGA-based systems are a significant area of computing, providing a high-performance implementation substrate for many different applications. However, the key to harnessing their power for most domains is developing mapping tools for automatically transforming a circuit or algorithm into a configuration for the system. In this paper we review the current state-of-the-art in mapping tools for FPGA-based systems, including single-chip and multi-chip mapping algorithms for FPGAs, software support for reconfigurable computing, and tools for run-time reconfigurability. We also discuss the challenges for the future, pointing out where development is still needed to let reconfigurable systems achieve all of their promise.

1.0 Introduction

Reconfigurable computing is becoming a powerful methodology for achieving high-performance implementations of many applications. By mapping applications into FPGA hardware resources, extremely efficient computations can be performed. In [Hauck98] we presented numerous reconfigurable systems, and described the varied roles of these systems. However, hardware is only one part of a complete system. In order for multi-FPGA systems to achieve widespread use, they not only require an efficient hardware implementation medium, but also a complete, automatic software flow to map circuits or algorithms onto the hardware substrate. Similar to a compiler for a standard programming language, the mapping software for a multi-FPGA system takes in a description of the circuit to be implemented, and through a series of transformations creates an implementation of the circuit in the basic primitives of the hardware system. In a multi-FPGA system, this implementation is programming files or bitstreams for the FPGAs in the system.

For the developers and users of reconfigurable systems there is a tradeoff to be made between hand mapping to the system, creating high-quality implementations but requiring a very hardware-savvy user, and automatic tools which free the user from some implementation details, but which can (sometimes substantially) impact the resulting quality. In this paper we focus primarily on software tools to partially or completely automate the mapping process. While these algorithms are still in their infancy, our hope is that they will lead to an efficient, automatic mapping solution for reconfigurable systems. However, it is important to realize that there will always be a role for hand-mapped implementations, just as in the compiler-dominated world of general-purpose computing there is still a role for assembly-language programming where the required quality demands the highest performance implementation possible.

Before we discuss the specific steps necessary to map onto a multi-FPGA system, it is necessary to consider some of the features of a multi-FPGA system that impact this software flow. One of the most important concerns in multi-FPGA systems is that while the FPGAs are reprogrammable, the connections between the FPGAs are fixed by traces on the circuit board. Thus, not all FPGAs may be interconnected, and communication between FPGAs must be carried on these limited resources. If the source and destination of a route are on different FPGAs, and the FPGAs are not directly connected, this signal will need to traverse one or more intermediate FPGAs. This adds extra delay to the routing, and uses up multiple FPGA I/O pins. This latter constraint can be the major bottleneck in multi-FPGA systems, with I/O resource constraints in some cases (especially logic emulation) limiting achieved logic utilization of FPGAs to 10%-20%. Thus, the primary concern of mapping tools for a multi-FPGA system is limiting the amount of I/O resources needed by a mapping. Not only does this mean that most signals should be kept within a single FPGA, but also that those signals that need to be communicated between FPGAs should be communicated between neighboring FPGAs. Thus, the mapping tools need to understand the topology of the multi-FPGA system, and must optimize the mapping to fit within this routing topology. Note that some topologies

use crossbars or FPICs[1], devices meant to ease this I/O bottleneck. However, even these chips have finite I/O resources, and the connections between them and the FPGAs are fixed. Thus the restrictions of a fixed topology occur even in these systems.

Not only must the tools optimize to a fixed topology, but in some circumstances they must also be fully automatic. In the case of logic emulation, the user of the system has no desire to learn all the details of a multi-FPGA system, and will be unwilling to hand-optimize their complex design to a multi-FPGA system. The system is meant to speed time-to-market of the system under validation, and without an automatic mapping solution the speed benefits of emulation will be swamped by the mapping time and human effort. Also, for a multi-FPGA system targeted to general software acceleration, the users of the system will be software programmers, not hardware designers. While there are situations where a hand-optimized solution may be the right answer, for many other domains a complete, automatic mapping system is a necessity. Merging these two demands together, developing an automatic system with the ability to also take user input to partially or completely guide the mapping process will be an important concern for reconfigurable systems.

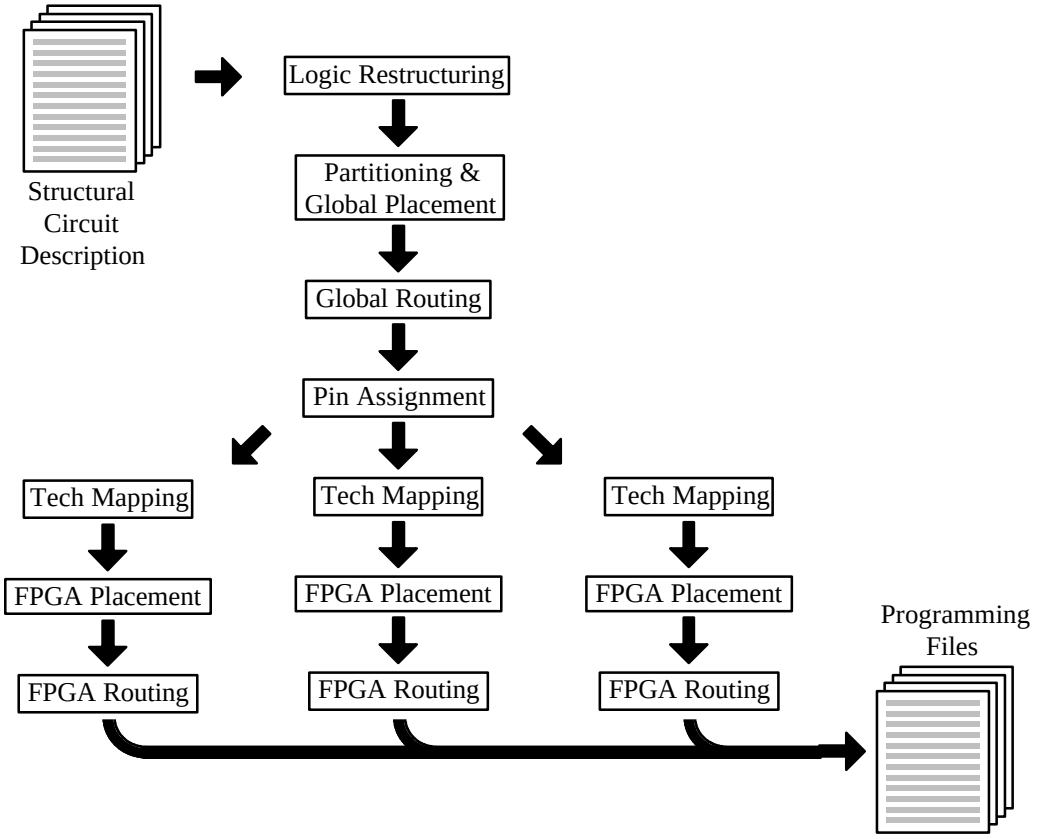

Figure 1. Multi-FPGA system mapping software flow.

The final major constraint is that the performance of the mapping system itself is an issue. A multi-FPGA system is ready to use seconds after the mapping has been developed, since all that is necessary is to download the FPGA configuration files to the system. Thus, the time to create the mapping dominates the setup time. If the mapping tools take hours or days to complete, it is difficult and time-consuming to make alterations and bug fixes to the mapping. This is especially important for rapid-prototyping systems, where the multi-FPGA system is part of an iterative process of bug detection, correction, and retesting. If the mapping time is excessive, the multi-FPGA

[1] FPICs are reconfigurable, routing-only devices which can perform nearly arbitrary interconnctions between their inputs [Aptix93, I-Cube94].

system will be used relatively late in the design cycle (where bugs are few), which greatly limits their utility. With a mapping process that takes only minutes, it becomes possible to use the benefits of logic emulation much earlier in the design cycle, increasing the usefulness of a multi-FPGA system.

2.0 Overview

The input to the multi-FPGA mapping software may be a description in a hardware description language such as Verilog or VHDL, a software programming language such as C or C++, or perhaps a structural circuit description. A structural circuit description is simply a representation of a circuit where all the logic is implemented in basic gates (ANDs, ORs, latches, etc.), or in specific, premade parts (i.e., microprocessors, memories, etc.). Programming language descriptions (Verilog, VHDL, C) differ from structural descriptions in that the logic may be described more abstractly, or "behaviorally", with the functions described by what needs to be done, not by how it should be implemented. Specifically, an addition operation in a behavioral description would simply say that two values are added together to form a third number, while a structural description would state the exact logic gates necessary to perform this computation. To implement the behavioral descriptions, there are automatic methods for converting such descriptions into structural circuit descriptions. Details of such transformations can be found elsewhere [McFarland90]. Our discussion of the mapping process starts with structural circuit descriptions, while sections 5 and 6 will overview methods for handling higher level specifications.

To convert from a structural circuit description to a multi-FPGA realization requires a series of mapping steps (Figure 1). These steps include logic restructuring, partitioning & global placement, global routing, pin assignment, and FPGA technology mapping, placement and routing. FPGA technology mapping, placement and routing are identical to the tools used for single FPGAs. In the pages that follow, we will first give a brief overview of the process. A more in-depth discussion of each of the steps appears later in this paper.

The first mapping step is logic restructuring. Logic restructuring alters the input netlist so that it can be mapped to the multi-FPGA system. Asynchronous components and gated clocks may need to be carefully considered or isolated, and RAM usage must be made to fit into the physical RAM chips available in the system.

After logic restructuring, partitioning takes the single input circuit description and splits it into pieces small enough to fit into the individual FPGAs in the system. The partitioner must ensure not only that the partitions it creates are small enough to fit the logic capacity of the FPGAs, but must also ensure that the inter-FPGA routing can be handled within the constraints of the multi-FPGA routing topology. Commonly global placement, the process of assigning partitions to specific FPGAs in the system, is combined with the partitioning stage. Otherwise, it is unclear where a given partition resides within the multi-FPGA topology, and thus it can be difficult to properly optimize the interpartition connectivity.

After partitioning and global placement, global routing handles the routing of inter-FPGA signals (i.e., the signals that need to be communicated between partitions). This phase can be broken up into abstract global routing and detailed global routing (pin assignment). Abstract global routing (hereafter referred to simply as global routing) determines through which FPGAs an inter-FPGA signal will be routed. Pin assignment then decides which specific I/O pins on each of the FPGAs will carry the inter-FPGA signals.

Once partitioning, global placement, global routing, and pin assignment are completed, all that is left is the technology mapping, placement and routing of the individual FPGAs in the system. When these steps are completed, there will be configuration files prepared for each FPGA in the system. Downloading these files to the multi-FPGA system then customizes the multi-FPGA system, creating a complete realization of the desired functionality.

In the sections that follow, we will discuss techniques for implementing each of these operations. Section 3 covers single-chip mapping tools, reviewing techniques from standard FPGAs for use in reconfigurable systems. Then, section 4 discusses multi-chip tools, showing what techniques are necessary to convert the multi-FPGA mapping problem into a set of independent single-chip mappings. Finally, we consider tools and techniques specialized for FPGA-based computation, including reconfigurable computing techniques in section 5, and run-time

reconfigurability in section 6. We then sum up the paper, providing an overview of current systems, as well as a description of areas for future innovation.

3.0 Single-Chip Mapping Tools

While some circuits are designed by hand, in many cases automatic mapping software is critical to logic development. This is particularly true for technologies such as FPGAs, where in general the complete mapping process is carried out by mapping software. In the following sections we will discuss some of the most important steps in mapping to LUT (lookup-table) based FPGAs. Note that the process is similar, but not identical, for other types of FPGAs. First is technology mapping, which restructures the input netlist into the logic blocks of the FPGA. Next, placement decides which specific logic blocks inside the FPGA will contain the logic functions created by technology mapping. Finally, routing determines what routing resources inside the FPGA will be used to carry the signal from where they are generated to where they are used. A more detailed treatment of all of these tasks can be found elsewhere [Venkateswaran94]. These single-chip FPGA mapping tools provide a method for implementing arbitrary logic within an FPGA, and form an important part of all mapping methodologies for FPGA-based systems.

3.1 Technology Mapping

The user of an FPGA provides as input a circuit specified as some interconnection of basic logic gates and functions. These functions may have more or less inputs than the LUTs in the FPGA that will implement them. When the logic gate has too many inputs, it must be split into smaller functions which can fit inside the LUTs in the FPGA. If the gates have too few inputs, several interconnected gates could be combined to fit into a single LUT, decreasing the amount of LUTs needed to handle the mapping. By reducing the number of LUTs, more logic can be fit in the same sized FPGA, or a smaller FPGA could be used. The process of restructuring the logic to best fit the logic blocks in an FPGA is called technology mapping.

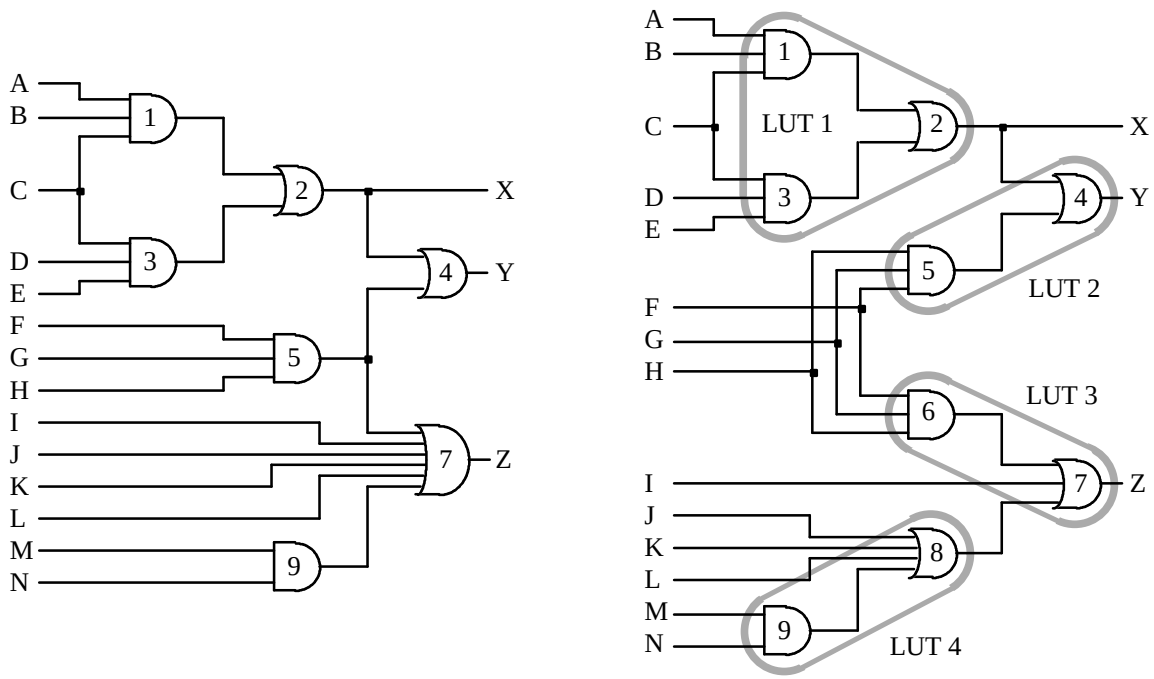

Figure 2. Example of 5-input LUT technology mapping. The input circuit (left) is restructured and grouped together into 5-input functions (right). The gray loops at right indicate individual LUTs. The numbers on the gates are for identification.

There are many different methods and approaches to the technology mapping of circuits for FPGA implementation [Brown92a, Vincentelli93]. An example of this process is shown in Figure 2. The circuit at left is restructured

into four 5-input LUTs, designated by the gray loops at right. Some of the logic functions, such as gate **7** at left, have more inputs than the LUT can handle. Thus, the gate will be split ("decomposed") into two gates (**7** and **8** at right). Then, the gates are grouped together ("covered") into LUTs, while trying to minimize the total number of LUTs required. Note that this grouping can be complex. For example, even though gates **1** and **3** each have three inputs, and thus should not fit into a single LUT with gate **2**, since input **C** is shared between the two gates a single 5-input LUT can handle all three gates. Finding this reconvergent fanout can be difficult. Also, this grouping process can cause the logic to be restructured. For example, gate **5** at left is duplicated, creating gates **5** and **6** at right. Although this seems like it would increase the hardware cost, replicating the logic can actually reduce the logic cost. In the example shown, if gate **5** was not replicated, it could not be grouped with either of its fanouts, since by grouping a gate with its fanout the gate's output is no longer available to other functions. However, by duplicating the gate, the duplicates can each be grouped with one of the fanouts, reducing the total LUT count.

Many approaches and optimizations are possible for technology mapping. For example, Chang et al. [Chang96] provides an approach that is capable of optimizing either the number of lookup tables or the latency of the resulting circuit. Similarly, methods for area and speed optimization are found in [Abouzeid93]. Instead of just mapping for LUT count, some algorithms optimize for performance or routeability (i.e., how easy it is to route the logic generated by technology mapping). Also, real FPGAs usually have logic blocks that are more complex than a single n-input LUT, and thus require more complex mapping algorithms. Recognizing that the use of LUT-based function units and restricted interconnect requires non-traditional logic synthesis approaches, Hwang et al. [Hwang94] present a logic synthesis technique based on communication complexity analysis. Numerous approaches to these and other mapping issues are presented in the literature, including work on logic decomposition [Murgai94, Chen95, Huang95a, Sawada95, Shen95, Stanion95, Wurth95, Legl96], covering [Cong92, Cong93, Farrahi94], and combined approaches [Francis90, Murgai90, Francis91a, Francis91b, Murgai91a, Murgai91b].

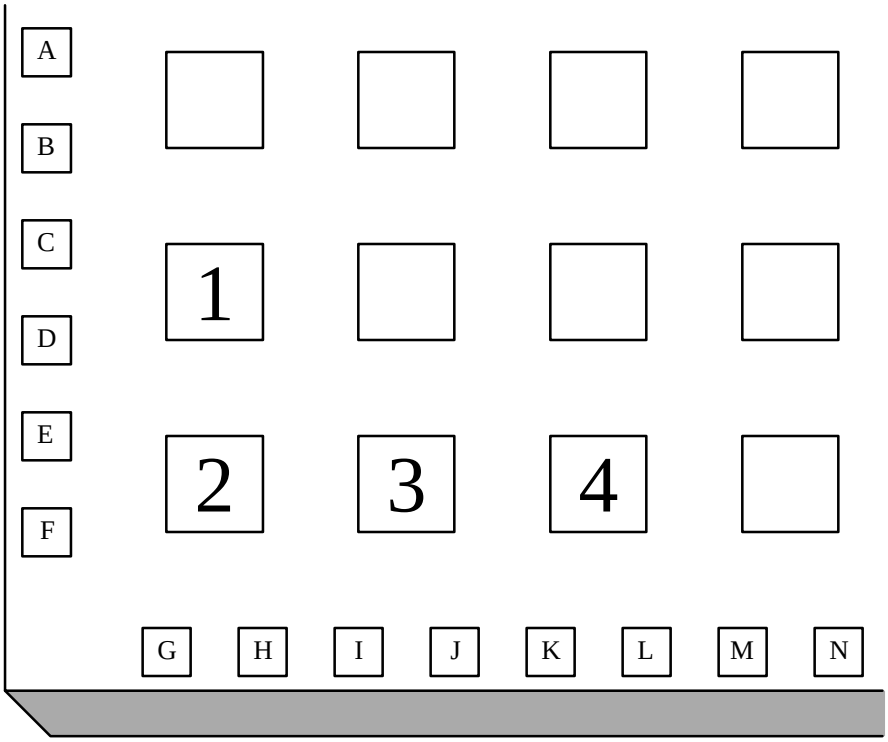

Figure 3. Placement of the circuit from Figure 2. The numbers are the number of the LUT (from the technology mapping) assigned to each logic block, while the letters are the assignment of input signals to I/O blocks.

Technology mapping can fit at several points within a complete multi-chip mapping system. As one example technology mapping can be performed after all multi-chip mapping steps (partitioning, global routing, and pin assignment) are completed. This allows the technology-mapping of the FPGAs to be performed in parallel (perhaps distributed across a set of workstations), speeding up the mapping process. However, if a circuit is partitioned before it is technology mapped, the partitioner must rely on estimates of circuit and component sizes, perhaps introducing inaccuracy into the mapping process. Recent work [Hauck95a] demonstrates that this inaccuracy is more than made up for by the added flexibility that pre-technology mapping partitioning can exploit, and thus we believe that technology mapping should in fact be performed only after all multi-chip mapping steps are completed.

3.2 Placement

Placement takes the logic functions formed by technology mapping and assigns them to specific logic blocks in the FPGA. This process can have a large impact on the capacity and performance of the FPGA. Specifically, routing between distant points in an FPGA requires a significant amount of routing resources. Thus, this path will be much slower, and use up many valuable resources. Because of this, the primary goal of placement is to minimize the length of signal wires in the FPGA. To do this, logic blocks that communicate with one another are placed as close together as possible. For example, Figure 3 shows a placement of the logic functions created by technology mapping in Figure 2. Since function **2** takes the output of function **1** as an input, and shares inputs **F**, **G**, and **H** with function **3**, function **2** is placed between function **1** and function **3**. Also, **F**, **G**, and **H** are assigned to I/O blocks close to function **2**.

Placement is a complex balancing act. Logic circuits tend to have a significant amount of connectivity, with many different functions communicating together. Trying to find the best two-dimensional layout of these elements can be quite difficult, since many functions may want to be placed together to minimize communications, while only a small fraction will fit within a given region of the FPGA. Thus, the placement tool must decide which functions are most important to place together, not just to minimize the distances of communications between these functions, but to minimize the total communication in the system.

The most common technique for performing placement for FPGAs (as well as other technologies) is simulated annealing [Shahookar91]. In order to use simulated annealing to solve an optimization problem, the programmer must generate a cost function and a move function. A cost function looks at a state of the system and assigns a value to the desirability of that state, with a lower value indicating a better result. For placement, a state of the system would be an assignment of logic functions to logic blocks, and I/O connections to I/O blocks. A cost function could be the total wirelength necessary to route in this configuration. Estimates are often used, since exact numbers are time-consuming to calculate, and simulated annealing requires the cost metric to be quickly computed. Alternatively, exact costs can be used if incremental costs can be quickly computed for a given move. Thus, states that have the smallest cost would require the least amount of routing, and would be better placements. More complex cost metrics, which take into account issues such as critical paths, are also possible. A move function is simply a method of transforming the current state of the system into a new state. Through repeated applications, this function should be capable of transforming any state of the system to any other. For placement, a move function could be to randomly pick two logic blocks in the FPGA and swap their contents.

One way to perform placement once a cost and move function are defined is to first pick a random starting point. The algorithm then repeatedly applies the move function to the current state of the system, generating a new state. If this state has a lower cost than the current state, it is accepted, and replaces the current state. Otherwise the current state is retained. Thus, this algorithm will greedily accept good moves, and move into a local minimum in the cost function's state space. The problem is that most cost functions have a huge number of local minima, many of which are much worse than the optimal placement (Figure 4). Specifically, from a given state there may be no way to swap two logic functions and reduce the cost (thus, we are in a local minima), though two or more pairs of swaps can greatly improve the placement.

Simulated annealing avoids the problem of getting caught in local minima. Like the greedy algorithm, simulated annealing takes the current state, uses the move function to generate a new state, and compares the cost metric in both states. If the move is a good move (that is, the cost function is lower for the new state than the old state) the

new state replaces the current state. However, instead of rejecting all bad moves (moves that increase the cost function), simulated annealing accepts some bad moves as well. In this way, the algorithm can get out of a local minima by accepting one or two bad moves. Subsequent good moves will then improve the results again, hopefully finding better results than the previous local minima.

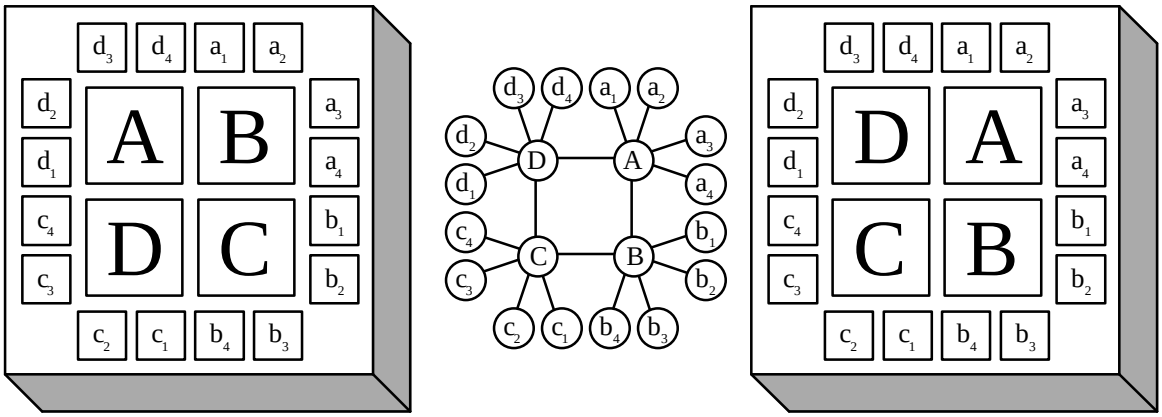

Figure 4. An example of a local minima in placement. The circuit at center, placed as shown at left, is in a local minima. No swap of logic or I/O functions will reduce the total wirelength. However, the placement at right is significantly better.

The method of how to determine what bad moves to accept is critical. The probability of accepting a bad move is usually $\exp(-\Delta C/T)$, where ΔC is the difference between the current and the new state's cost functions, and T is the temperature, a parameter that allows more or less bad moves to be accepted. Whenever the algorithm finds a bad move, it calculates the different between the current and new state. The algorithm then randomly determines whether to accept this bad move, and it is more likely to accept moves causing small increases in the cost function than big increases in the cost function (that is, the worse the move, the less likely it is to be accepted). Also, over time it gets pickier, accepting less and less bad moves. This is done by lowering the temperature parameter *T*. At the beginning of the annealing the algorithm accepts many bad moves, and randomly wanders around the search space. Since it always accepts good moves, it tends to stay in the portion of the search space where better states are found, but the large amount of bad moves accepted keep it from getting stuck in any one place. As time goes on, *T* is decreased, and the algorithm accepts less and less bad moves. As this happens, it gets harder for the algorithm to wander away from the areas in the cost function where the better states are found. Thus, it is stuck in one region of the state space based on very coarse-grain measures of goodness, and it begins to stay in parts of this region where better states are found. The algorithm continues to accept less and less bad moves, until eventually it accepts only good moves. At this point, the algorithm is zeroing in on a local minima, though this minima tends to be much better than the average local minima in the search space, since the algorithm has slowly gravitated to areas in the search space where better states are found. In this way, simulated annealing can find much better results than greedy approaches in complex search spaces.

By applying simulated annealing to a placement problem, the complex relationships between the logic in the mapping can be considered. The algorithm will slowly optimize the state, coming up with a good final placement. Note that simulated annealing can be quite time-consuming. This is because the algorithm must be allowed to accept many good and bad moves at each acceptance level, so that the algorithm can explore much of the search space. Thus, multiple hour annealing runs are not unusual.

3.3 Routing

Routing for FPGAs is the process of deciding exactly which routing resources will be used to carry signals from where they are generated to where they are used. Unlike many other technologies, FPGAs have prefabricated routing resources. Thus, instead of trying to limit the size of routing channels (the goal in standard cell routing), an FPGA router must work within the framework of the architecture's resources. Thus, the router must consider

the congestion of signals in a channel, making sure than no more routes are made through a region than there are resources to support them. Otherwise, if too many resources are required the routing fails, while in other technologies the region could just be enlarged.

An example of the routing of the placement from Figure 3 is shown in Figure 5. The router must decide which inputs and outputs of the logic blocks to connect to, which channels to route through, and how to connect through the switchboxes in the architecture. It is allowed to choose which terminal on the logic block to connect to because the logic is implemented in LUTs, and all inputs to a LUT are equivalent (assuming the programming of the LUT is modified accordingly). In deciding which channels and wires to use, and how to connect through the switchboxes, the router must ensure that there are enough resources to carry the signal in the chosen routing regions, as well as leaving enough resources to route the other signals in the system. One algorithm for performing this routing is presented in [Brown92b]. Here, the algorithm is divided into a global and a detailed router. The global router picks which routing regions the signal will move through. Thus, it will select the routing channels used to carry a signal, as well as the switchboxes it will move through. In this process, it takes care not to assign more signals to a channel than there are wires. However, the global router does not decide the wire assignment (i.e., which specific wires to use to carry each signal). The detailed router handles this problem, making sure that it finds a connected series of wires in the channels and switchboxes chosen by the global router that connects from the source to all the destinations. Both algorithms worry about congestion-avoidance, making sure that all signals can be successfully routed, as well as minimizing wirelength and capacitance on the path, attempting to optimize the performance of the circuit. By running both algorithms together, a complete routing solution can be created.

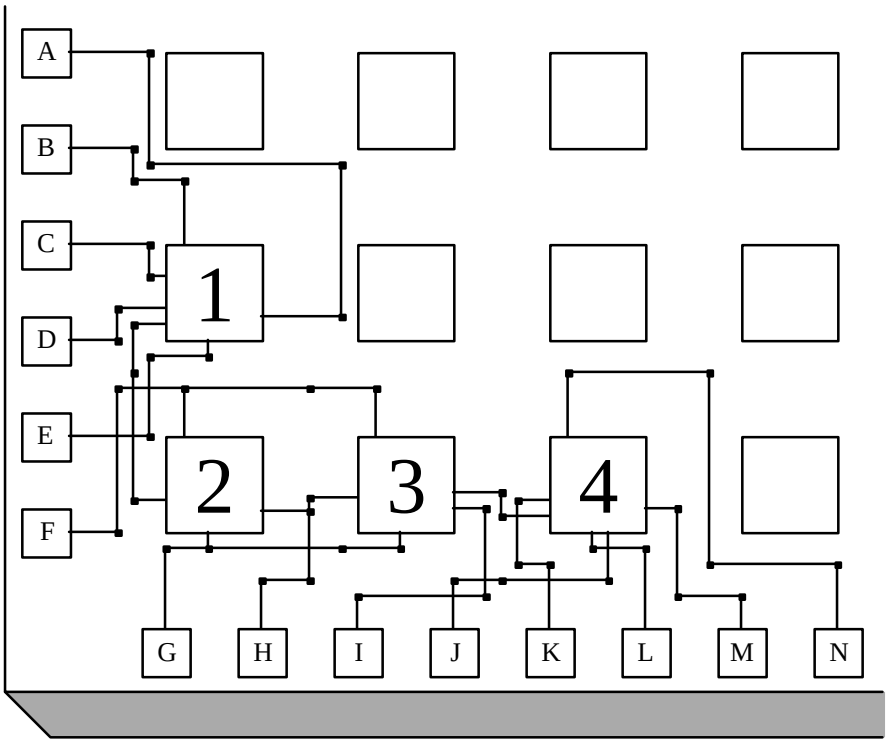

Figure 5. Routing of the placement from Figure 3. The small black squares are the configuration points used for this mapping.

Once the circuit has been technology mapped, placed, and routed, a complete implementation of the circuit on a given FPGA has been created. A simple post-processing step can now turn this implementation into a configuration file for the FPGA which can be downloaded to the reconfigurable system to implement the desired circuit. In the sections that follow we will describe how a large circuit design can be broken into multiple, single-chip mappings.

4.0 Multi-Chip Systems

FPGA vendors have provided tools that can take a mapping intended for a single FPGA, and automatically create a configuration file, and thus an FPGA-based realization of the design. However, many FPGA-based systems do not consist of simply a single FPGA, but may have multiple FPGAs connected together into a multi-FPGA system, as well as perhaps memory devices and other fixed resources. Mapping a circuit to such a system is much more complex than the single-chip mapping process, including considerations and constraints not found within a single FPGA. In this section we discuss techniques for solving these problems, presenting the current state-of-the-art in multi-FPGA mapping. These techniques, in conjunction with those described in the previous section, are capable of taking a complex circuit (one that is much larger than the capacity of a single FPGA) and create a complete implementation.

4.1 Software Assist for the Hand-Mapping Process

Perhaps the simplest method for mapping to a multi-FPGA system (at least from the tool developer's standpoint) is to have the user produce the mapping from scratch, specifying the complete implementation details. This requires the user to have a good understanding of their application and the reconfigurable system, but will potentially allow the user to create a much better implementation than is possible with an automatic mapping system.

Producing a mapping completely from scratch, with no software support, is often impossible. At a minimum, for most FPGAs it is necessary to rely on a software tool to create the bit patterns to load into the FPGA from a placed and routed circuit, since the information to do this by hand is often not provided by the FPGA vendors. However, there are several other ways in which software automation can help in even a hand-mapping environment. For example, when laying out a regular structure like an adder circuit, a user need not specify the exact location of each individual adder cell, and would much rather just state that the adder cells are placed next to one-another down a specific column in the FPGA. Also, a user may wish to create a specific hardware template for a commonly occurring component in their design and then replicate it as needed. Several such systems, which act as assistants to the hand-mapping process, and automate some of the less critical portions of the mapping process, have been produced [Gokhale95, Vuillemin96]. These systems remove some of the complexity of the hand-mapping process, allowing the designer to concentrate on the more important aspects of the design. However, these systems do not replace the designer, and require a user with a high degree of knowledge and sophistication. For less experienced users, or less demanding tasks, higher levels of automation are often essential.

In the rest of the paper we concentrate on the automation of the mapping process, attempting to develop a "compiler" for an FPGA-based system. However, with each of these systems it is important to realize that hand-mapping is often an alternative, and the quality losses due to software automation must be balanced against the design complexity of performing that optimization by hand. It is important to note that most high-performance systems based on FPGAs have largely been mapped to manually. Hopefully, in the decade to come we will transition to a model more in line with general-purpose computing, where most programs are translated from a high-level language into machine instructions via a completely automatic mapping process, though where there is still the potential for assembly-language programming (the hand-mapping of the software world) in situations where the performance gains are large enough or important enough to make such efforts worthwhile.

4.2 Partitioning and Global Placement

The user of a multi-FPGA system usually specifies the desired functionality as a single, large structural circuit. This circuit is almost always too large to fit into a single FPGA, and must instead be split into pieces small enough to fit into multiple FPGAs. When the design is split up, there will be some signals that need to be communicated between FPGAs because two or more logic elements connected to this signal reside on different FPGAs. This communication is a problem for a multi-FPGA system, because the amount of I/O resources on the FPGAs tends to be used up long before the logic resources are filled.

Because I/O resources are the main limitation on logic capacity in a multi-FPGA system, the primary goal of the partitioner, the tool that splits logic up into FPGA-sized partitions, is to minimize the communication between partitions. There have been many partitioning algorithms developed that have as a primary goal the reduction of

inter-partition communication, and which can split a design into multiple pieces, including algorithms designed specifically to map into multiple FPGAs [Kuznar93, Woo93, Kuznar94, Weinmann94, Chan95, Huang95b, Kuznar95, Fang96]. One such algorithm [Chou94] is also designed to efficiently handle the large circuits encountered during logic emulation. Instead of partitioning the entire circuit, it instead focuses only on a small portion of the circuit at a time. Although this speeds up the partitioning, it may cause the algorithm to ignore more global concerns, including assigning logic to more than one partition. To fix these problems, the algorithm uses an technique inspired by the ESPRESSO II logic minimization package [Brayton85] to recombine and restructure the partitioning. If logic is assigned to multiple partitions, the algorithm will restructure the partitions to remove the overlap. Also, more global concerns will be handled by breaking the partitions initially created into smaller portions and recombining them into (hopefully) a better final partitioning.

Unfortunately, many previous multiple-FPGA algorithms are not designed to work inside a fixed topology. Specifically, most multi-way partitioning algorithms, algorithms that break a mapping into more than 2 parts, assume that there is no restriction on which partitions communicate. They only seek to minimize the total amount of that communication, measured as 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). These are reasonable goals for when the partitioner is run before the chips are interconnected, and is used in cases where someone is building a custom multi-FPGA system for a specific application. However, most multi-FPGA systems are prefabricated, with the FPGAs connected in a routing topology designed for general classes of circuits, and these connections cannot be changed. Thus, some FPGAs may be interconnected in the topology, while others will not, and the partitioner needs to understand that the cost of sending a signal between pairs of FPGAs depends on which FPGAs are communicating. Even when the multi-FPGA system is custom designed for an application, later modifications to the circuit will require similar, topology-aware partitioning tools.

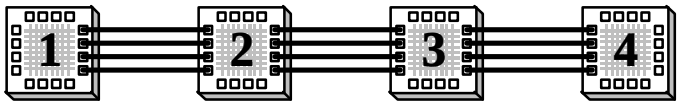

Figure 6. Example system for topological effects on multi-FPGA system partitioning.

For an example of topological issues in multi-FPGA system partitioning, consider a linear array of 4 FPGAs, numbered **1**-**4** (Figure 6). A signal communicated between FPGAs **1** & **2** will consume two I/Os, one on each FPGA, since the FPGAs are directly connected. A signal between FPGAs **1** & **4** will consume six I/Os, two between **1** & **2**, two between **2** & **3**, and two between **3** & **4**. Thus, when partitioning onto this linear array of FPGAs, it is better to have two signals being communicated, one between **1** & **2**, and one between **2** & **3**, than it is to have a single signal communicated between **1** & **4**. In order to do a reasonable job of partitioning onto a multi-FPGA system, the partitioner needs to be aware of the topology onto which it is partitioning. For this reason the step of global placement, the assigning of partitions to specific FPGAs in the system, is often done simultaneously with partitioning. Otherwise it is hard for a partitioner to understand the cost of communication within a given topology if it does not know where the partitions it is creating lie within this topology.

One method for performing partitioning onto a multi-FPGA systems is to handle it as a placement problem. Specifically, we can apply simulated annealing to the partitioning problem, allowing the tool to assign logic to specific FPGAs in a multi-FPGA system [Roy94, Roy-Neogi95]. Just as in placement for single FPGAs, the tool randomly assigns logic to functional units, which in this case are the logic-bearing FPGAs. The logic assigned to an FPGA is restricted to the FPGA's capacity. The partitioner then uses the simulated annealing algorithm to seek better placements through random moves under an annealing schedule. However, one problem with partitioning onto a multi-FPGA topology is that it can be difficult to estimate the routing costs of a given assignment. In an FPGA, routing distances tend to obey a simple geometric cost metric, with the physical distance between source and sink directly relating to the routing costs. In a multi-FPGA system, there may be an arbitrary connection pattern between FPGAs, with no simple geometric cost metric accurately capturing the actual routing complexity. However, in meshes and other regular architectures, geometric routing cost metrics can be developed, and simulated annealing or other algorithms can be applied. Another possibility for such a system is the partitioner for the Anyboard system [Thomae91, Van den Bout92, Van den Bout93], which applies a variant of the Kernighan-

Lin algorithm [Kernighan70] to a bus-structured multi-FPGA system where interconnect costs are easily determined.

In order to handle the partitioning problem on topologies that do not admit a simple routing cost metric, it is necessary to come up with some other method of estimating routing complexity. One method is to model the routing problem as a multicommodity flow problem and use this as part of the partitioning process [Vijayan90]. The multicommodity flow problem can estimate the routing complexity on a general graph, meaning that this approach can be used on an arbitrary multi-FPGA topology. However, the flow calculation can be very complex, and thus cannot be used in the inner loop of a complex partitioning algorithm. Thus, the flow calculation is only run after a significant number of iterations of a normal algorithm, which uses the previous flow calculation as an estimate of current routing conditions. However, even with this modification, their algorithm has an exponential worst-case complexity.

Although the approaches discussed previously can develop reasonably accurate estimates of partitioning quality, they take a significant amount of runtime to complete. The approach in [Vijayan90] has an exponential worst-case complexity, and simulated annealing is notoriously slow. An alternative to both of these algorithms is to adapt the standard partitioning algorithms [Alpert95], which ignore interconnection limitations, to partitioning onto a topology. For crossbar topologies, multi-way partitioning algorithms can be directly applied. In a crossbar topology [Hauck98] routing costs are adequately captured by the amount of inter-FPGA signals connected to each logic-bearing FPGA in the system, which is the optimization metric of many multi-way partitioning algorithms. For a hierarchical crossbar (which has multiple standard crossbar topologies connected together by higher levels of crossbar interconnections) multi-way partitioning algorithms can be recursively applied. In a hierarchical crossbar there are N subsets of the topology interconnected in the highest level in the crossbar. Minimizing the communication between these N subsets is the primary concern in a hierarchical crossbar, because the connections at the highest level of a hierarchical crossbar tend to be the primary routing bottleneck. Thus, a multi-way partitioning algorithm can split the circuit into N pieces, one per subset of the topology, and optimize for this bottleneck. At this point each of these subsets can be handled independently, either as a crossbar or a smaller hierarchical crossbar. In this way, standard multi-way partitioning algorithms can be recursively applied to a hierarchical crossbar, yielding a fairly fast partitioning algorithm.

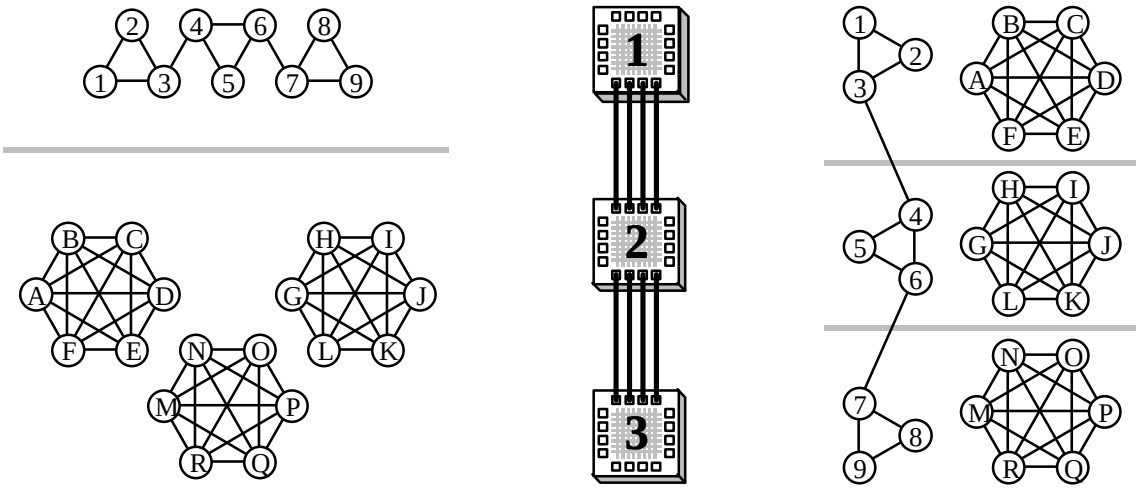

Figure 7. An example of the suboptimality of recursive bipartitioning. We are partitioning to the multi-FPGA system at middle, and the best first cut is shown at left (gray line). The next partitioning would have to cut nine signals. We can instead partition as shown at right, cutting a total of two signals.

A similar recursive partitioning approach can be applied to other multi-FPGA system topologies. A split in the multi-FPGA topology is chosen that breaks the system into two or more subsets, and the logic is split into a similar number of pieces. These subsets are then subdivided repeatedly until the mapping has been broken into N pieces, one per FPGA in the system. By using a fast bipartitioning or multi-way partitioning algorithm, a complete

partitioning of the circuit onto the multi-FPGA topology can be performed. Such an approach has been used to quickly partition circuits onto the Teramac system [Snider95].

There are two problems with recursive bipartitioning onto a multi-FPGA system: it is greedy (overlooking more global concerns), and it ignores the multi-FPGA topology. It is greedy because the first bipartitioning attempts to find the best possible bipartitioning. While it may find a good way to split a mapping if it is only going to be broken into two parts, it may be a poor choice as a starting point for further cuts. The first split may require only a small amount of communication between the two halves of the system, but later cuts may require much more communication. We may have been better served having a somewhat larger initial cut, which might ease congestion on later cuts, and thus minimize the overall amount of inter-partition communication. An example of this is shown in Figure 7.

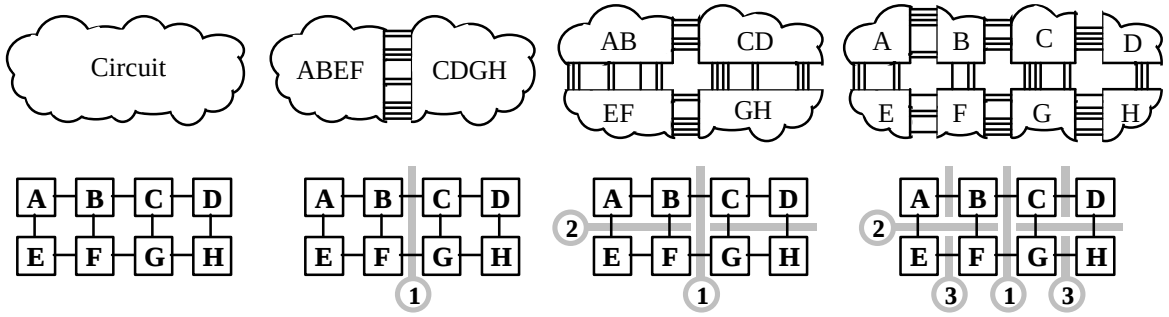

Figure 8. Example of iterative bipartitioning. The circuit (top left) is partitioned onto the multi-FPGA topology (bottom left) in a series of steps. Each partitioning corresponds to the most critical bottleneck remaining in the multi-FPGA system, and after each partitioning the placement of the logic is restricted to a subset of the topology (labeling on circuit partitions).

The greediness of iterative bipartitioning can be taken advantage of in order to map onto a specific multi-FPGA topology [Hauck95b]. Specifically, the multi-FPGA topology itself will have some bottleneck in its topology, some place where the expected communication is much greater than the routing resources in the multi-FPGA topology. For example, going back to the linear array of FPGAs discussed earlier (Figure 6), it is clear that if the number of wires connecting adjacent FPGAs is the same throughout the system, then the wires between the middle FPGAs, numbers **2** & **3**, will be the most heavily used, since with an equal number of FPGAs on either side of this pair of FPGAs the communication demand should be the highest. Not all multi-FPGA systems will be linear arrays of FPGAs, but within most of them will be some set of links that are the critical bottleneck in the system. If we iteratively partition the circuit being mapped onto a topology such that the first cut in the circuit falls across this critical bottleneck, with the logic in one partition placed on one side of the bottleneck, and the other partition on the other side of the bottleneck, then this greedy early cut is performed exactly where the best cut is necessary. We can continue this process, performing partitionings corresponding to the remaining bottlenecks in the system. In this way we take advantage of the greedy nature of iterative bipartitioning, performing the early, greedy cuts where necessary to relieve the most significant bottlenecks in the topology, while later cuts (which are hurt by the greediness of earlier cuts) are performed where lower quality cuts can be tolerated, since they correspond to locations that are not critical bottlenecks in the system. An example of this process is shown in Figure 8. While this does not totally avoid the suboptimality of iterative bipartitioning, it does help limit it.

There are other ways of improving on the basic hierarchical greedy scheme. In [Kim96], Kim and Shin present algorithms in which a first cut hierarchical partitioning phase is followed by a second pass iterative optimization phase in which all constraints are satisfied. Chou et al. [Chou94] present a scheme that uses a local ratio-cut clustering followed by a set covering partitioning method. [Kuznar94] uses functional replication to improve the quality of partitioning.

4.2.1 Partitioning for VirtualWires

The gate versus IO requirements for partitioned blocks are often mismatched with the gate versus IO balance in commercial FPGAs. Generally, experience has shown that for a given number of gates, partitioned blocks tend to have more IOs than FPGAs do. A compilation technique called VirtualWires [Babb93] attempts to bridge this gap through a multiplexing and pipelining approach that enables superfluous gates on FPGAs to be traded for pins, thus allowing more efficient use of FPGA resources. The basic idea is that when the number of logical signals that must cross partitioned boundaries is greater than the number of physical pins on an FPGA, the VirtualWires technique multiplexes and pipelines several logical signals on each physical FPGA pin. Consequently, the multiplexors allow more logical signals to cross FPGA boundaries, but incur a gate cost that is proportional to the number of logical signals and largely independent of the number of physical pins. Multiplexing and pipelining of multiple logical signals onto a single physical pin is orchestrated against a discrete time-base provided by a new clock called the virtual clock. A VirtualWires compiler schedules both logic evaluation and communication on specific cycles of the virtual clock. Scheduling is discussed further in Section 4.3.3.

Partitioning algorithms for VirtualWires are modified forms of those used in traditional systems, differing largely in the cost constraints or the optimization criteria they use during the optimization process [Agarwal95]. Let us represent partitioning constraints as follows. Suppose G_i is the number of gates in partition i, and P_i is the number of pins in partition i. Also suppose that G_{max} and P_{max} are the maximum number of gates and pins (respectively) that an FPGA can support.

In a traditional system, the constraints faced by the partitioner are: $G_i \quad G_{max}$ and $P_i \quad P_{max}$. The constraints are graphically represented in the left hand side of Figure 9 showing the feasible partitioning region in terms of gates and pins. Partitioning attempts to divide a set of gates into the minimum number of partitions that satisfy the above constraints. Typically, the traditional process involves dividing the number of gates into some number of partitions in a manner that satisfies the gate constraint, and then moving gates between partitions to minimize the number of pins in an attempt to satisfy pin constraints.

Now, let us consider the VirtualWires case. Let c denote the "cost" in terms of logic gates of a virtual wire. In other words, let c represent the average number of overhead logic gates in a VirtualWires system that are required to implement a single logic signal that must be sent or received by an FPGA (for simplicity, we assume that both inputs and outputs cost the same). In our experience c is roughly four gates. In a VirtualWires system, the constraint faced by the partitioner is $G_i + c\ P_i \quad G_{max}$. The resulting constraint space for the VirtualWires case is denoted in Figure 10. As can be seen, because pin constraints can be folded into a gate constraint, the feasible space is larger with VirtualWires. The intuition why the feasible space is larger is that gates can be traded for pins. If a given partition has more pins than the others, then the increased number of pins can be compensated for by placing fewer gates on that FPGA.

As before, partitioning attempts to divide a set of gates into the minimum number of partitions that satisfy the above constraint. VirtualWires partitioning involves dividing the number of gates into some number of partitions and then moving gates between partitions to reduce the number of IO so that the above gate constraint is satisfied for each partition. The partitioning algorithm itself can be one of those described earlier.

4.2.2 Global Placement

The placement of partitions on FPGAs can be accomplished as a separate, standalone phase in the compilation process, or it can be combined with partitioning. A separate placement phase simplifies the software system, but the overall results are generally poorer. The placement process assigns partitions to FPGAs. Placement attempts to put partitions that communicate intensely or partitions that are in timing critical paths as close together as possible. Because the placer must attempt to optimize several criteria, such as global communication volume and the critical path length, simulated annealing is a commonly used technique.

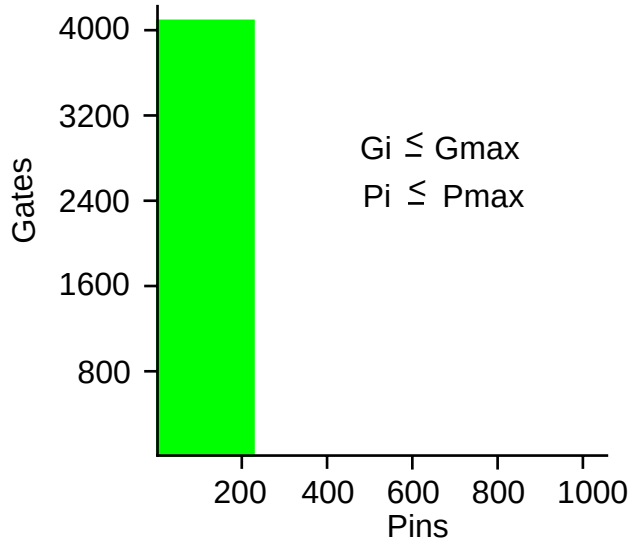

Figure 9. Constraints used by a traditional partitioner.

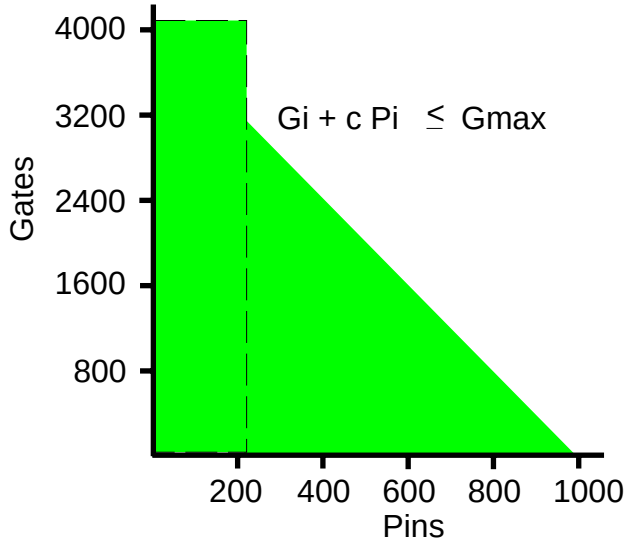

Figure 10. Constraints used by a VirtualWires partitioner.

To apply simulated annealing to the multi-FPGA placement problem requires both a cost function and a move function. The move function for a multi-FPGA system usually involves simply swapping the contents of two FPGAs. Developing a cost function for this problem is more difficult, since considerations such as critical path length and routing congestion must often be considered.

An important factor in such algorithms is that the move generator be able to estimate the move cost quickly - a move function that requires recalculating the routing of the entire system after each move is impractical. Unfortunately, to compute the true routing complexity or critical path length for a given configuration of the system requires just such an action - the swap of two partitions between FPGAs can significantly change most routing-related costs. Estimates can be made by ignoring congestion, and assuming all routes can take the minimum path between FPGAs. However, if there is routing congestion due to a given placement of the logic, this routing estimator will ignore this effect, and yield an overly optimistic cost metric. Thus, the simplicity of separating the partitioning and global routing steps must be balanced against the potential for lower-quality results due to inaccurate cost metrics.

4.3 Global Routing

Global routing is the process of determining through which FPGAs or interconnection switches to route inter-FPGA signals. Note that the related problem of determining which specific pins to use to carry a signal is often handled separately; it will be discussed later in this paper. In cases where a signal needs to connect between only two directly connected FPGAs, this step is usually simple. No intermediate FPGA is needed to carry this signal, and the routing can be direct. However, in some cases the wires that directly connect these FPGAs may be used up by other signals. The signal will need to take alternate routes, routes that may lead through other FPGAs. For long-distance routing, there may be many choices of FPGAs to route through. A global router attempts to choose routes in order to make the most direct connections, thus minimizing delay and resource usage, while making sure that there are sufficient routing resources to handle all signals.

Routing is a problem in numerous domains, including standard cells, gate arrays, circuit board layout, and single FPGAs. Global routing for multi-FPGA systems is similar to routing for single FPGAs. Specifically, in an FPGA the connections are fixed and finite, so that the router cannot add resources to links that are saturated, and the connections may not obey strict geometrical distance metrics. The same is true for a multi-FPGA system. In the cases of standard cells and gate arrays, the size of routing channels can (usually) be increased, and in circuit board routing extra layers can be added. Thus, for multi-FPGA global routing many of the techniques from single-FPGA routing are also applicable.

In the sections that follow we will discuss routing algorithms for several different types of systems, including crossbar and direct-connect systems. We will also consider global routing for Virtual Wires systems.

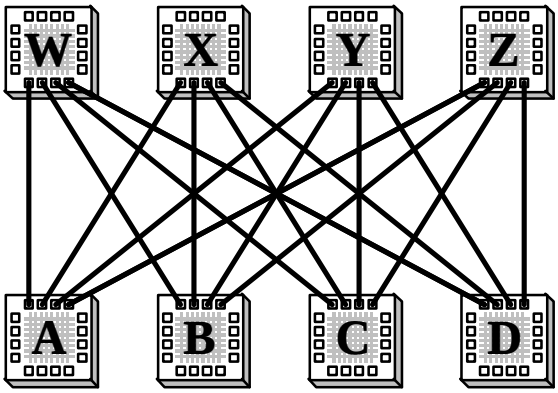

Figure 11. Crossbar routing topology. The chips at top are used purely for routing, while the FPGAs at bottom handle all the logic in the system.

4.3.1 Global Routing for Crossbar Topologies

In a crossbar topology (Figure 11), a signal that needs to be routed between FPGAs will always start and end at logic-bearing FPGAs, since routing-only chips have no logic in them. Thus, routing any signal in the system, regardless of how many FPGAs it needs to connect to, requires routing through exactly one other chip, and that chip can be any of the routing-only chips. This is because each routing-only chip connects to every logic-bearing FPGA, and the routing-only chips have exactly the same connectivity. Thus, routing for a crossbar topology consists simply of selecting which routing-only chip an inter-FPGA signal should route through, and a routing algorithm seeks only to route the most number of signals. Note that this does require some effort, since some assignments of signals to routing-only chips allow much less communication than others. For example, assume that each of the connections between chips in Figure 11 consist of 3 wires, and assume that we are attempting to route three-terminal wires, with connections evenly distributed between the logic-bearing FPGAs. If we route signals between FPGAs **ABC** through W, **ABD** through X, **ACD** through Y, and **BCD** through Z, we will be able to route three signals through each routing-only chip, for a total of 12 signals. At this point, no further routing can

be performed, even of two-terminal wires. We can instead route one **ABC**, one **ABD**, one **ACD**, and one **BCD** wire through each of the routing-only chips, achieving a total of 16 routed signals.

There have been several algorithms proposed for the routing of signals in crossbar topologies. [Mak95b] presents an algorithm that is optimal for 2-terminal routing, as well as proof that routing for multi-terminal nets is NP-Complete. The routing algorithm is based on the idea that if too many routes are assigned to a given routing-only chip, there must be some routing-only chip that is underutilized. Otherwise, there would be no possible routing for this mapping. Given this fact, the routes going through these two chips can be balanced, so that there is almost an identical number of nets going to each of these routing-only chip from each logic-bearing chip. Thus, to perform the routing for a crossbar topology, simply assign all of the nets to a single routing-only chip, and then iteratively balance the demand on routing-only chips until the routing is feasible. Since the problem of routing multi-terminal nets in a crossbar topology is NP-Complete, heuristic algorithms have been proposed [Butts91, Kadi94]. These greedily route signals through routing-only chips based on their current utilization. Because of this, they may not always find a routing solution in situations where such a solution exists, even in the case of purely 2-terminal routing. Thus, there may be some benefit in combining a heuristic approach to multi-terminal net routing with the optimal approach to 2-terminal net routing, though it is not clear how this could be done. An alternative is to break all multi-terminal nets into a spanning tree of only two-terminal nets. Then, the optimal algorithm can be applied to routing these 2-terminal nets. [Mak95a] proposes an algorithm which will perform such a decomposition, as well as make sure that the I/O capacity of the FPGAs are not violated by the splitting of multi-terminal nets. However, this decomposition requires extra I/O resources on some FPGAs in the system to accomplish this splitting, and if such resources are not available, this algorithm cannot find a routing, even though such a routing may in fact exist for non-decomposed multi-terminal nets.

4.3.2 Global Routing in Direct-Interconnect Systems

Thus far we have looked at routing for indirect interconnects. Indirect interconnection systems are those in which logic bearing FPGAs are distinct from FPGAs or switches that handle routing tasks. Routing with direct interconnects is a little more complicated. In a direct interconnect hardwire system, routing is the problem of selecting the set of FPGAs through which to route a signal between a pair of partitions. The following algorithm describes a routing approach for direct interconnect hardwire systems. We first present a simple approach and then suggest methods to improve the route quality.

This routing method begins by constructing the *FPGA channel graph*. This is a directed graph that represents the FPGA topology in which each node represents an FPGA and each directed edge represents a channel. A channel is the set of directed connections from one FPGA to another. Each channel has a width representing the number of physical wires in the channel. The use of directed channels with predetermined widths is a simplification because the IO pins in typical FPGAs can be configured to behave as either outputs or inputs. Figure 12 shows an example input circuit that we wish to map onto a mesh connected FPGA topology. The channel graph for the FPGA topology is shown in Figure 13. The latter figure also shows the initial values of the available channel capacities.

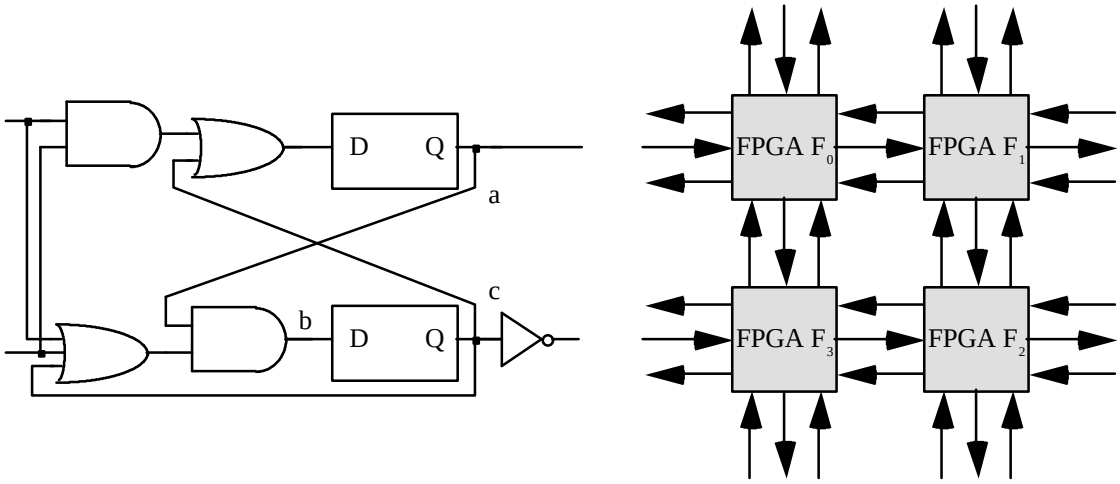

Figure 12. Input circuit (left) and FPGA topology (right).

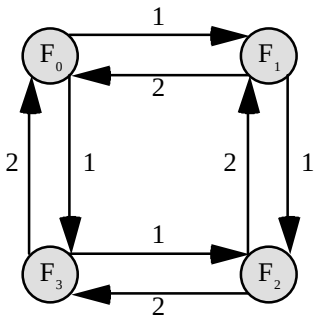

Figure 13. FPGA channel graph showing the initial values of the available channel capacities.

The algorithm also relies on a *distance weight* and the *available channel capacity* of each channel in the channel graph. For channel *C*, let C^d represent the distance weight, let C^w represent the physical channel width, and let C^a represent the available channel capacity. Distance weights are used to turn off channels that are full. Assign a default distance weight of 1 to each channel and initialize the available channel capacity to the channel width. In other words, initialize,

$$C^d = 1$$

$$C^a = C^w$$

The algorithm works by repeated applications of the following step: pick a wire and find a route comprising a sequence of channels from its source partition to its destination partition. If a net has $d > 1$ destination terminals, then the router can treat the net as d two-terminal wires, each of which connects the source terminal to one of the destination terminals.

More specifically, the algorithm applies the following step until all nets have been assigned a route.

1. Pick a wire and find the partition that sources the net. Let s denote the source partition and let s_f denote the FPGA on which the source partition is placed. Similarly, let d be the destination partition for the net and let d_f be the FPGA on which the destination partition lies.

2. Find the shortest path P_{sd} (using a standard shortest path algorithm, where $\begin{matrix} d \\ i \end{matrix}$

$C_i^a = 0$, for any $i = 0 \ldots n\text{-}1$, then set $C_i^d = \infty$.

4. Associate the path P_{sd} with the wire and mark the wire so it is not picked again.

As an example, suppose the design in Figure 12 is partitioned and placed as in Figure 14. Also suppose that the wire *a* is the first wire picked to be routed. The source FPGA s_f for this wire is F1 and the destination FPGA d_f is F3. The initial state of the channel graph is as shown in Figure 13. The state of the graph after wire *a* is routed is shown in Figure 15. Figure 16 shows a possible routing of all the nets.

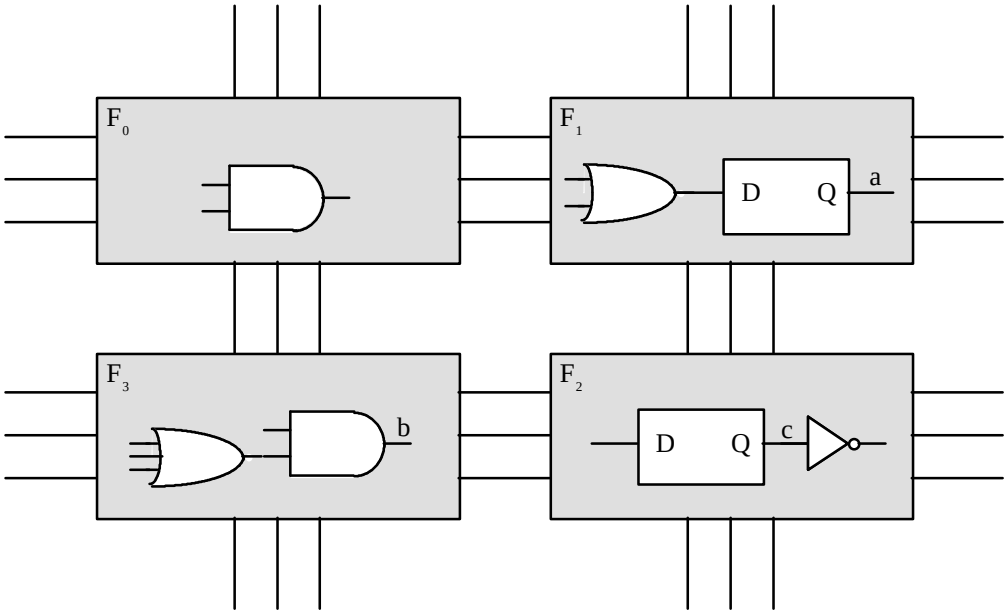

Figure 14. A partition and placement of the input circuit.

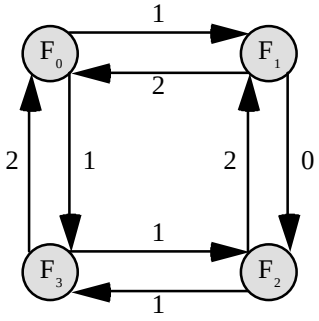

Figure 15. Channel graph showing available channel capacities after routing the net *a*.

The above algorithm can be optimized in many ways. For example, consider this optimization for multi-terminal wires *n* that are represented as *d* two-terminal wires $n_1, n_2, n_3, \ldots n_d$. Suppose that the two-terminal wires $n_1, n_2, \ldots n_{i-1}$ have been previously routed, and we are attempting to route wire n_i. We can choose as the source FPGA for n_i any of the FPGAs that are connected by wires $n_1, n_2, \ldots n_{i-1}$, potentially saving valuable channel capacity.

In a direct-interconnect system, the channels can sometimes run out of capacity and the global router can fail to find any path to route a given wire. The router can attempt to make further progress by ripping out one or more previously assigned routes to make space in the congested channel and unmarking the related wires. However, because pin usage for routing hops in a direct interconnect system is related to the characteristics of the input

netlist, providing guarantees of routing success is impossible. Repartitioning the netlist for a larger number of FPGAs can exacerbate the problem because of the increased routing distances and the associated increase in hop-related pin requirements. Consequently, large multi-FPGA systems often resort to indirect interconnection networks.

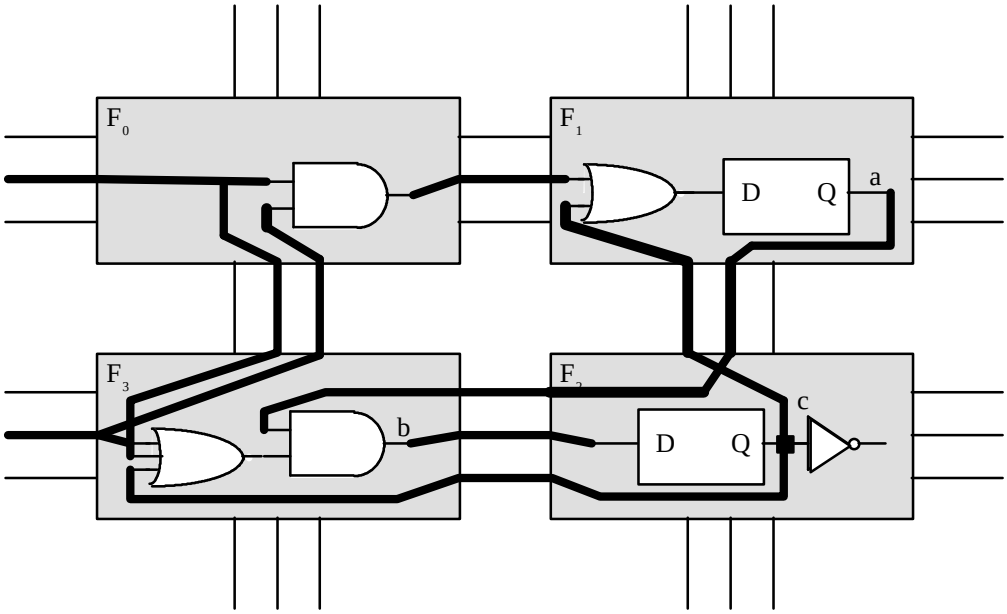

Figure 16. A routing of the input circuit on the target topology.

The global router in hardwire systems must also attempt to satisfy hold time constraints in gated clock designs. For correct operation, the edge-triggered flip-flop discipline requires that the input signal be valid for a period of time — called the *hold time* — following the occurrence of the edge on the flip-flop clock terminal. In a typical VLSI design, the user or the layout tool is responsible for laying out the signal wires and controlling delays so that hold times are met. FPGAs provide special controlled delay clock signal lines for routing clocks to flip-flops in FPGAs so they meet hold times when the design is mapped onto FPGAs. However, when flip-flops are clocked by a gated clock signal derived as some combinational function of a clock and user design signals and is routed over normal FPGA routing logic there is the danger that the hold times will not be met. This situation is exacerbated when the design spans multiple FPGAs, where it is even harder to control delays. In such situations, the global placer and router must arrange the lengths of the signal and gated clock paths (perhaps padding the input signal path with extra delay) so that hold times are satisfied.

4.3.3 Global Routing and Scheduling in VirtualWires systems

Global routing for VirtualWires systems is also called global scheduling because it includes a time dimension and produces a source-destination path in time and space. The scheduler in VirtualWires systems is also responsible for the correct ordering of logic evaluation and communication events, and relies on a digital abstraction of time and space for both simplicity and reliability.

Time and Space Digitization Recall that VirtualWires systems multiplex the physical wires in the inter-FPGA channels so that the number of signals that can be communicated within a single cycle of the user's clock is not bounded by the channel capacities. Sharing of physical wires and scheduling of computation and communication is governed by a virtual clock that provides a discrete time-base. Computations and possible registration of signals within an FPGA take place during a specific sequence of virtual clock cycles prescribed by the scheduler. Thus each cycle of a user's clock is digitized into multiple virtual clock cycles.

In addition to time discretization, VirtualWires systems digitize space as well. Each signal is registered at FPGA boundaries in a flip-flop synchronous to the virtual clock, and is communicated between a pair of FPGAs over a

physical wire on a given virtual clock cycle. Thus, all combinational timing paths are entirely within individual FPGAs. Timing paths also involve inter-FPGA crossings between directly connected FPGA pairs. Such a digital space abstraction not only provides a simple model to the scheduler, but yields system composability in which local timing correctness guarantees global timing correctness.

Timing Resynthesis To facilitate complete event ordering control, all clocked elements in the system are resynthesized to use the virtual clock or signals synchronous to the virtual clock. As part of its routing process, the scheduler establishes the specific virtual cycle on which each storage element is enabled.

The TIERS Routing Approach The goal of the scheduler is to orchestrate all events in the system. In other words, it must prescribe the specific set of virtual clock cycles and the physical channels on which inter-FPGA signals are communicated. Because a physical channel wire is allocated to a given signal only during a specified virtual clock cycle, and not dedicated for all time, the scheduler must also determine the virtual cycle on which the given signal reaches its legal value and can be sampled for transmission outside the chip. The following is a simplified version of the TIERS algorithm for scheduling VirtualWires systems. For more details on TIERS (Topology Independent Pipelined Routing and Scheduling) see [Selvidge95].

The TIERS routing approach comprises the following steps:

1. Construct the *FPGA channel graph*.
2. Order inter-partition wires through a dependence analysis.
3. Determine the time-space path for each wire.

As with direct-interconnect systems, we first construct the FPGA channel graph. Each channel comprises a given number of wires. Although each wire is assumed to be able to carry a single signal during a given virtual clock cycle, each wire has an unbounded temporal extent. In other words, a given wire can transmit more signals by using additional virtual cycles. Because channels can yield more capacity by using more virtual clock cycles, the simplest VirtualWires scheduling algorithm does not require distance weights on the channels. Recall the distance weights are used by hardwire routers to disconnect full channels.

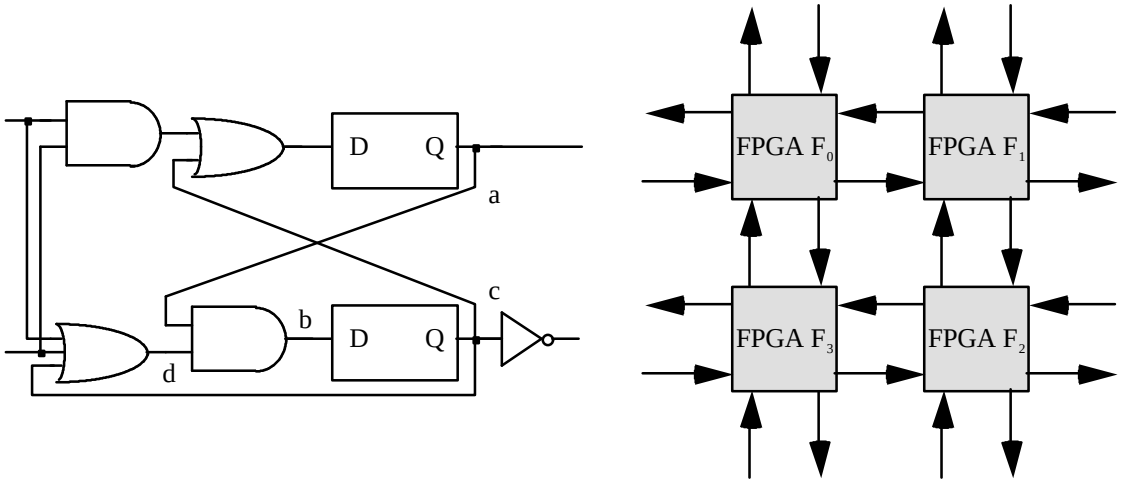

Figure 17. Input circuit (left) and FPGA topology (right).

The available channel capacity in VirtualWires systems is a function of the virtual clock cycle. In other words, each channel has a specific available channel capacity for a given virtual clock cycle. We use a dynamic array indexed by the virtual clock cycle for the available channel capacity, and initialize each element to the channel width. In other words, let C^w represent the physical channel width of channel C, and let $C^a(t)$ represent the available channel capacity for time slot t. Furthermore, for all channels, initialize

$$C^a(t) = C^w \quad \forall t$$

Consider the same input circuit as before, but a FPGA topology with fewer physical connections as in Figure 17. The channel graph for this topology for use in VirtualWires routing is shown in Figure 18.

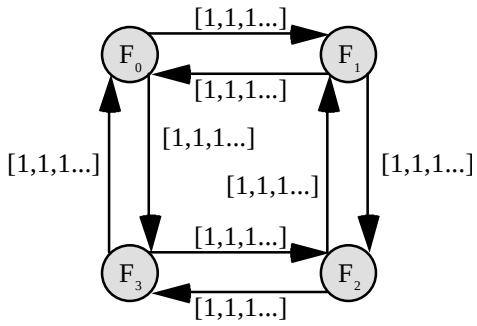

Figure 18 FPGA channel graph showing the initial values of the available channel capacities as a function of the virtual cycle. Initially, $C^a(t)$ for each channel is 1 for all values of t.

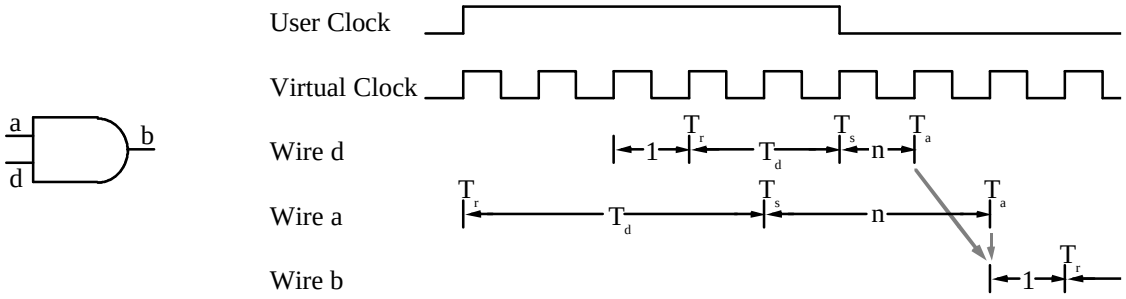

Figure 19. The relationship between the various scheduling times for signals *a*, *d*, and *b*. Wire *b* depends combinationally upon *a* and *d*, and *a* is produced as the output of a flip flop.

The second step involves ordering inter-partition wires through a dependence analysis. In a discrete event system, the scheduler must guarantee that signals that are inputs to a combinational logic block must be available before the outputs of the combinational logic block can be routed. It does so by routing all input wires *before* routing any of the output wires of any combinational block. System input wires and flip-flop output wires comprise the initial conditions for the recursive process implied by the above precedence constraint. A routing order consistent with the above constraint is facilitated by creating a sorted list of inter-partition wires in which a wire w_i precedes another wire w_j only if w_i does not depend combinationally on w_j. We also associate with each output wire of a partition the set of partition input wires on which it depends. As an example, wire *a* (shown in Figure 17) must be scheduled before wire *b* because *b* depends combinationally upon *a*.

The third step involves finding time-space routes for inter-partition wires. The routing algorithm does so by performing the following sequence of operations to wires picked in order from the sorted list of wires, until all wires have been assigned a time-space route. A time-space route comprises a sequence of channel and virtual-cycle pairs from the source FPGA to the destination FPGA. As before, the router treats a net with $d > 1$ destination terminals as d two-terminal wires, each of which connects the source terminal to one of the destination terminals.

1. Find the partition that sources the wire. Let s denote the source partition and let s_f denote the FPGA on which the source partition is placed. Similarly, let d be the destination partition for the wire and let d_f be the FPGA on which the destination partition lies.

2. Find the shortest path P_{sd} (using a standard shortest path algorithm) between FPGAs s_f and d_f. P_{sd} is simply the sequence of channels C_i: $i = 0 \ldots n\text{-}1$ in the shortest path between FPGAs s_f and d_f.

The next set of steps determines the scheduling of the signals along the sequence of channels, and establishes the ready time, arrival time, and send time for the signal (see Figure 19).

3. Determine the *ready time* T_r for the wire. The ready time of a wire is the earliest virtual cycle on which the wire attains its legal value. The ready times on system input wires and wires sourced by flip-flops is initialized to 1. The ready times for other wires is one plus the maximum of the *arrival times* of the wires on which they depend. The one extra cycle allows for local combinational computation and registration of the signal in a flip-flop just prior to transmission through an FPGA output pin.

   The *arrival time* T_a of a wire at an FPGA is the time (or virtual cycle) at which the signal on the wire will arrive at that FPGA. Assuming there are no combinational cycles in the circuit, picking wires according to the order in the sorted wire list guarantees their ready time can be calculated.

4. Compute the *departure delay* T_d for the wire. The departure delay is the number of cycles the wire is delayed at the FPGA as it waits for the availability of a sequence of available time slots along the path P_{sd}.

   One variant of the routing algorithm [Selvidge95] adds delays to a wire only at the source FPGA. That is, once a signal is scheduled for routing it suffers no additional delays till it reaches its destination FPGA. With the source-delay algorithm, a workable route can be obtained by looking for available channel capacity in each of the channels in P_{sd} during successive time slots starting with the ready time T_r of the wire (i.e., for $T_d = 0$). If the signal is not routable starting at T_r, then the departure delay is increased until a workable route is found.

   More formally, a wire is routable with departure delay T_d if the following is true:

$$C_i^a(T_r + T_d + i) \geq 0\,,\ i = 0 \ldots n-1$$

   The above assumes the each hop is accomplished in one virtual cycle.

5. Compute the *send time* T_s for the wire. The send time for a wire is the sum of the ready time and the departure delay for the net.

$$T_s = T_r + T_d$$

6. Compute the *arrival time* of the net at the destination FPGA. Because our routing model assumes that delays are added only at the source, the arrival time is simply the sum of the send time and the number of hops n. In other words, if the number of channels in P_{sd} is n,

$$T_a = T_s + n$$

7. Decrement the available channel capacity for each channel in P_{sd} for the time slot on which it is used by the net.

$$C_i^a(T_s + i) = C_i^a(T_s + i) - 1\,,\ i = 0 \ldots n-1$$

   When the available channel capacity of any channel i during a time slot t is exhausted, i.e., $C_i^a(t) = 0$ $C^a(t) = 0$, that channel will not be selected for further routing during time slot t.

8. Associate the path P_{sd}, the send time T_s, and the arrival time T_a with the net. The T_s time is needed by a future VirtualWires synthesis step that actually implements FPGA-specific state machines and enables appropriate flip-flops and multiplexing logic on the appropriate virtual cycle. T_a, on the other hand, is needed to compute the ready time of other nets that depend on it.

As an example, consider wires *a*, *b*, and *c* shown in Figure 17. One possible ordering of the wires in the depend sorted wire list is *a*, *c*, *b*. T_r for both *a* and *c* is 1 because both are sourced by a flip flop in the original design. Since both can be scheduled for transmission immediately T_d for both is 0, and therefore T_s for both is 1 As depicted in Figure 21, the wire *a* can be scheduled on the channel from F1 to F2 on virtual cycle 1 and on the

channel from F2 to F3 on the following virtual cycle. Thus, T_a for *a* is 3 since *n* (the number of hops) is 2. The channel graph following the scheduling of *a* is as shown in Figure 20. Similarly, wire *c* can be scheduled on the channel from F2 to F3 on virtual cycle 1. T_r for wire *b* can be calculated as 3 (one plus the arrival time for *a*). Because the channel from F3 to F2 is available, T_d for *b* is 0, and *b* can be scheduled on virtual cycle 3.

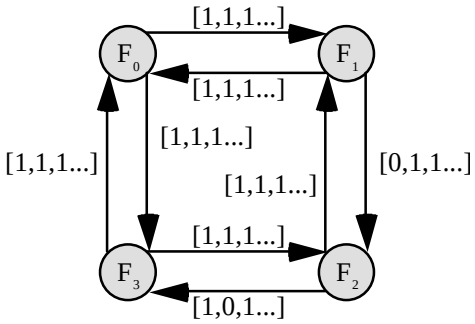

Figure 20. FPGA channel graph showing the values of the available channel capacities after wire *a* has been scheduled.

As part of its routing process, the scheduler also establishes the specific virtual cycles on which the resynthesized storage elements must sample their inputs. It particular, it uses the arrival times of signals at an FPGA to compute the ready times of signals at inputs of storage elements, and thereby obtains a lower bound on the time those storage can sample their inputs. Similarly, it also ensures that flip flops hold their values at least until the send time T_s of the corresponding signal. Because the scheduler exercises complete control on when flip flops are enabled for gated-clock designs, and when signals arrive as inputs to flip-flops, hold time problems are also eliminated [Selvidge95]. See [Agarwal95] for a more detailed discussion on this material.

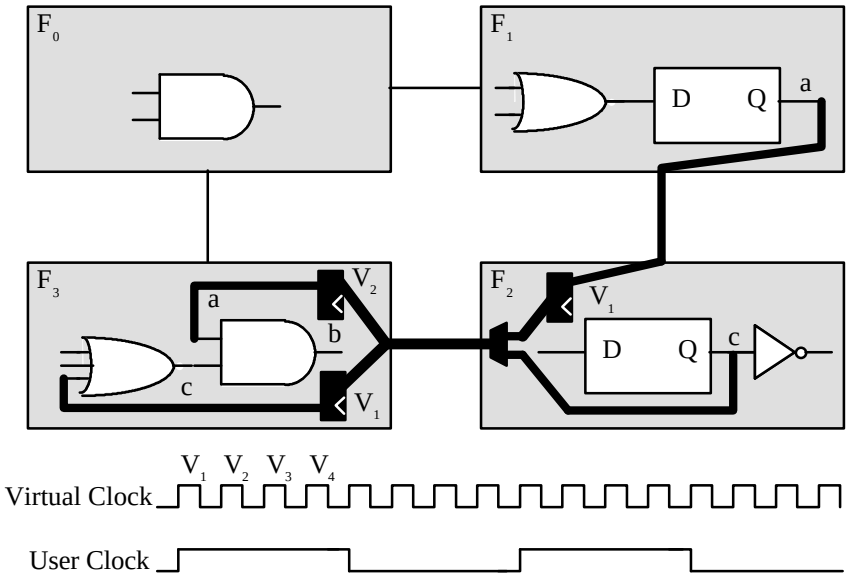

Figure 21. Virtual routing of wires *a* and *c*. On virtual cycle V1, the signal on wire *a* is routed from F1 to intermediate hop F2, and the signal on wire *c* is routed from F2 to F3. The figure also shows the multiplexing and hop logic introduced by VirtualWires to implement the routing and scheduling.

Improvements to the basic routing technique discussed above are described in [Selvidge95]. One improvement, called critical path sensitive routing, attempts to route critical path nets before other nets. Recall that the wires are scheduled in dependence partial order. When several candidate wires are routable according to the partial order, critical path sensitive routing selects the wire with the greatest depth, that is, one that belongs to a combinational path with the largest number of FPGA crossings. Experience has shown that critical path sensitive TIERS

typically produces schedules that come close to the lower bound established by the length of the longest critical path in the design.

4.4 Pin Assignment

Pin assignment is the process of deciding which I/O pins to use for each inter-FPGA signal. Since pin assignment occurs after global routing, the FPGAs through which long-distance routes will pass have already been determined. Thus, if we include simple buffering logic in these intermediate FPGAs (this buffering logic can simply be input and output pads connected by an internal signal), all inter-FPGA signals now move only between adjacent FPGAs. Thus, pin assignment does not do any long-distance routing, and has no concerns of congestion-avoidance or resource consumption.

The problem of pin assignment has been studied in many other domains, including routing channels in ASICs [Cai91], general routing in cell-based designs [Yao88, Cong91], and custom printed circuit boards (PCBs) [Pförtner92]. Unfortunately, these approaches are unsuitable to the multi-FPGA pin assignment problem. First, these algorithms seek to minimize the length of connections between cells or chips, while in a multi-FPGA system the connections between chips are fixed. The standard approaches also assume that pin assignment has no effect on the quality of the logic element implementations, while in the multi-FPGA problem the primary impact of a pin assignment is on the quality of the mapping to the logic elements (individual FPGAs in this case). Because of these differences, there is no obvious way to adapt existing pin assignment approaches for other technologies to the multi-FPGA system problem.

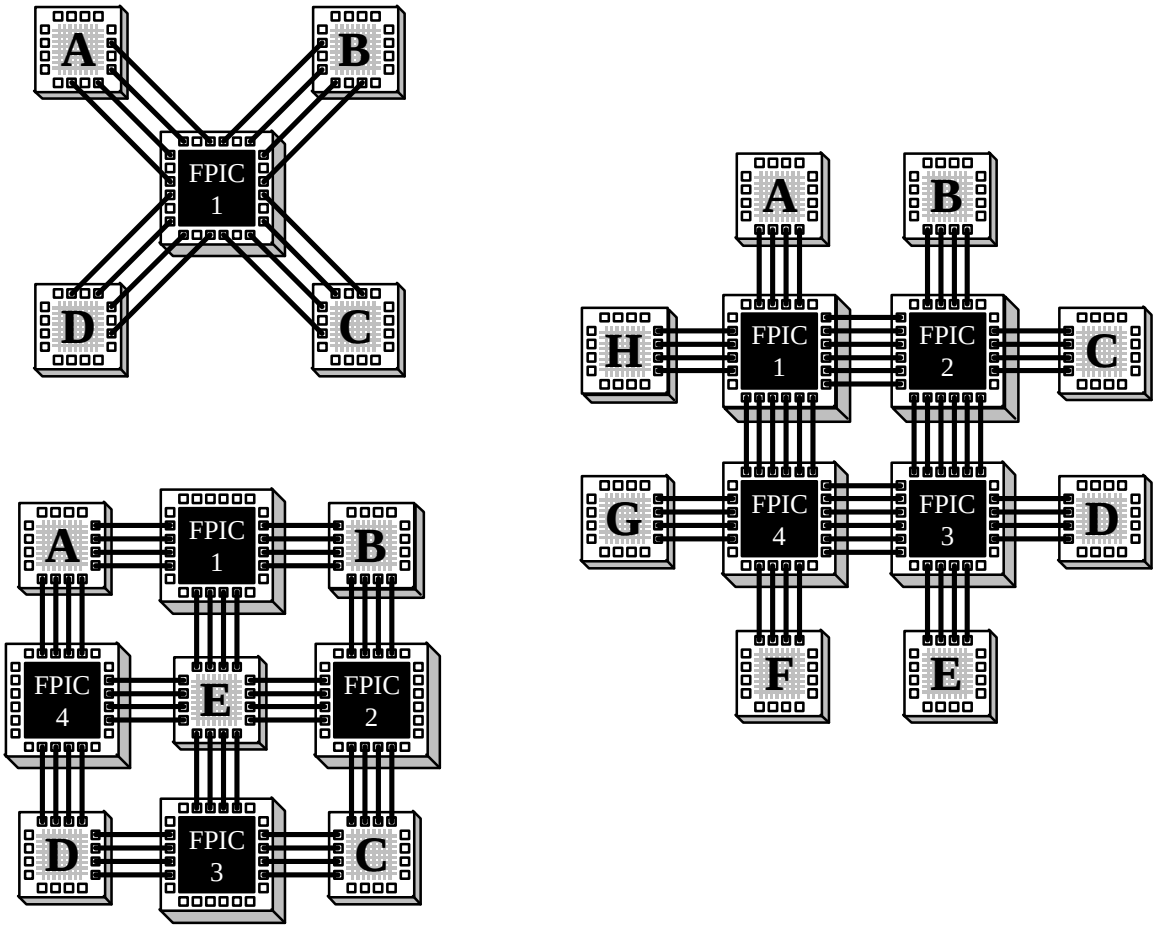

Figure 22. Multi-FPGA topologies with simple pin assignments.

There are several methods of pin assignment for multi-FPGA topologies. The most obvious is for systems connected only through a single FPIC, crossbar chip, or routing-only FPGA (Figure 22 top left). In these situations, it is assumed that the routing chip can handle equally well any assignment of signals to its I/O pins, and thus the only consideration for pin assignment is the routeability of the logic-bearing chips. Since no logic-bearing chips are directly connected, because all routing is through the single routing chip, we can place these FPGAs independently. These placement runs can be almost completely unconstrained, allowing inter-FPGA signal pins the freedom to be assigned to any I/O pin going to the routing chip. Once the individual placements have completed, the pin assignments created by the placement tools induce a pin assignment onto the routing chip. That is, pins on the routing chip must be assigned such that they lie on the opposite end of the board trace connected to the appropriate pin on the FPGAs. In general this is a trivial problem, since there are one-to-one connections between FPGA pins and routing chip pins. Since the routing chip is capable of handling any pin connection pattern equally well, the routing chip need only be configured to handle this connection pattern, and the mapping to the multi-FPGA system is complete. A similar approach works for two-level topologies [Hauck98] where there are multiple routing chips, but no two logic-bearing chips are directly connected, and no logic-bearing chip is connected to more than one routing chip (Figure 22 right). Again, the logic-bearing chips can be placed independently, and this induces a pin assignment onto the routing-only chips.

In a multi-FPGA topology where no two logic-bearing chips are directly connected, but with logic-bearing FPGAs connected to two or more routing-only chip (Figure 22 bottom left), the algorithm must be modified slightly. Depending on which I/O pin on an FPGA a signal is assigned, it may go to a different routing-only chip. However, its ultimate destination may only be connected to one of these routing-only chips, and so the choice of which chip to route through is critical. Even if the destination of the route is connected to the same chips as the source, if the placements of these two FPGAs independently choose different routing-only chips through which to send the signal, there will need to be an additional connection between these routing-only chips. Such connections may not exist, or the chips may be too heavily congested to handle this routing. Thus, the placements cannot be free to choose the routing-only chip to which to connect a signal. However, this is not a problem. The global routing step determines through which chips to route inter-FPGA signals, and is capable of picking which routing-only chip is the best to use for any given route. Thus, all that is necessary is to make sure the pin assignment respects these connections. This is easy to accomplish, since it is clear which I/O pins on a logic-bearing FPGA are connected to which routing-only chip. This establishes constraints on the placement tool, telling it to assign individual logic pins to certain subsets of the chip's I/O pins. In this way, we can again place all the FPGAs independently, and let their placement induce a pin assignment on the routing-only chips.

As described in [Chan93], there are some topologies that can avoid the need to have global routing determine the routing-only chip through which to route, and instead everything can be determined by the placement tool. The simplest such topology is a crossbar topology with two logic-bearing and two routing-only chips (Figure 23 left). The pin connection pattern of the logic-bearing chips has alternating connections to the two different routing-only chips (Figure 23 right), so that each connection to one routing-only chip is surrounded by connections to the other routing-only chip.

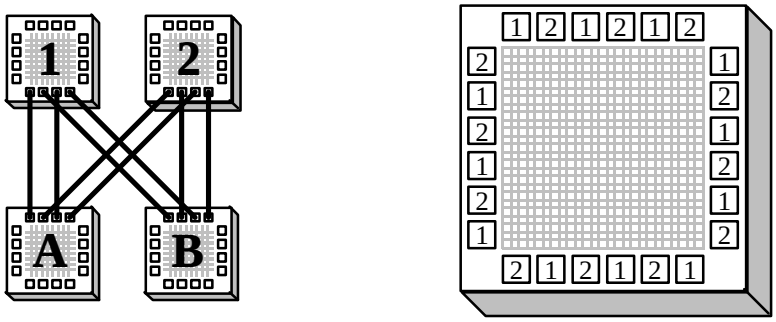

Figure 23. Topology for integrated global routing and pin assignment (left), with the FPGAs at bottom used for logic, and the FPGAs at top for routing only. The detailed pin connections of the logic-bearing FPGAs are given at right. The numbers in the pin positions indicate to which routing-only chip that pin is connected.

In a topology such as the one described, the logic-bearing FPGAs can be placed independently, without any restriction on the routing-only chip through which the signals must move. The inter-FPGA routing pins can be placed into any I/O pin connected to a routing-only chip, and are not constrained to any specific routing-only chip. While this gives the placement tool the flexibility to create a high quality placement, the source and destination pins of an inter-FPGA signal may end up connected to different routing-only chips. Obviously, this is not a valid placement, since there are no inter-routing chip wires to connect these terminals. However, since the problem terminals connect to different routing-only chips, and since the neighboring I/O pin positions go to different routing-only chips, moving one of the logic pins to one of the neighboring I/O pins fixes the routing problem for this signal. While this move may require moving the logic pin that already occupies the neighboring I/O pin, an algorithm for efficiently handling this problem is presented in [Chan93] (hereafter referred to as the *Clos routing algorithm*). It will require moving pins at most to a neighboring I/O pin location, and can fix all routing problems in the topology presented above. Since this is only a slight perturbation, the placement quality should largely be unaffected. The alternate solution, where global routing determines which routing-only chip a signal should be routed through, can have a much more significant impact. Assume that in the optimum placement of the logic in the chip, where inter-FPGA signals are ignored, a set of 10 logic pins end up next to each other in the FPGA's I/O pins. Under the method just presented, these pins might be slightly shuffled, but will still occupy pins within one I/O pin position of optimum. Under the global router solution, the router might have assigned these 10 pins to be routed through the same routing-only chip. This would require these I/O connections to be scattered across every other pin position (since only half the pins in the region go to any given routing-only chip), forcing some of the connections a significant distance from the optimum.

While the Clos routing algorithm requires somewhat restricted topologies, it can be generalized from the topology in Figure 23. Specifically, there can be more than two logic-bearing chips, and more than two routing-only chips. The main requirements are that the logic-bearing chips are connected only to routing-only chips, the routing-only chips only to logic-bearing chips, and there are the same number of connections between each pair of logic-bearing and routing-only chips. The pin connections are similar to those shown in Figure 23 right, with connections interspersed around the logic-bearing FPGA's periphery. Specifically, if there are *N* routing-only chips in the system, then there are *N*-pin regions on the logic-bearing FPGA's periphery that contain connections to each of the *N* routing-only chips. In this way, when the pin assignment is adjusted after placement, a pin need only move within this group to find a connection to the appropriate routing-only chip, yielding a maximum movement of *N-1* pin positions from optimal.

The pin assignment methods described above are adequate for systems where logic-bearing FPGAs are never directly connected. However, as show in [Hauck98], most multi-FPGA systems have logic-bearing FPGAs directly connected. An algorithm for handling such systems is described in [Hauck94]. The main idea is that pin assignment is just a process designed to simplify the interaction between the placements of connected FPGAs. A good pin assignment balances the routing demands in all the FPGAs in the systems. Thus, one way to perform pin assignment for a multi-FPGA system is to simultaneously place all the FPGAs in the system, with the placement tool's cost function incorporating routing complexity metrics for all the FPGAs. Thus, it can accurately determine

whether the benefit of a pin positioning in one FPGA is worth the routing problems it causes in connected FPGAs. However, since placement of a single FPGA is a very complex, time-consuming process, complete placement of tens to thousands of FPGAs simultaneously is intractable. The solution adopted is to simplify the placement process during the pin assignment stage, retaining only enough information to produce a reasonable assignment. After this step is done the individual FPGA mappings are developed independently by standard FPGA placement and routing tools, since once pin assignment is completed all inter-chip issues have been resolved.

The placement simplification process for pin assignment is based on properties of force-directed placement. Force-directed placement is a circuit placement technique that attempts to minimize wirelength by replacing all nets in the system by springs, and then seeking the minimum net force placement [Shahookar91]. This tends to make the springs shorter, which tends to make the nets shorter. In [Hauck94] it is shown that by removing the restriction that logic must be put into exact, unique function block locations, the rules of physics can be used to remove all the logic from the mapping. In this process, the springs connected to a logic node during force-directed placement can be replaced with springs connected only to the logic node's neighbors. However, these neighbors see exactly the same forces as if the logic block was still part of the mapping. Thus, by repeated application of these spring reduction rules, all of the logic blocks can be removed from consideration. This leaves IOBs (connections from an FPGA to the rest of the system) interconnected by springs as if the entire mapping was still there. Thus, force-directed placement can then be applied to this simplified mapping quite efficiently, and a good quality pin assignment can be developed. Note that by removing the restriction on logic node placement, this pin assignment approach uses an inexact placement model. However, this reduction in local accuracy tends to be more than made up for in the ability to seek the global optimum. Also, subsequent single-FPGA placement runs will be made to fix up this inaccuracy.

4.5 Logic Resynthesis

Multi-FPGA systems often require a *logic resynthesis* step before the other mapping phases. One view of logic resynthesis is that it is akin to technology mapping. Logic resynthesis is a transformation step that attempts to create a match between components in the user design and those available in the multi-FPGA hardware substrate. For example, user designs typically contain wide, deep, multiported memories, while those available in the hardware substrate are often single ported and narrow. This section discusses typical logic restructuring transformations for memory and tristate nets.

In many systems, small single-ported memories can be treated much like any other combinational block to be mapped to the system (for example, see [Sample94, Butts91]). Memory can be implemented using either FPGA memory capabilities or external RAMs. However, memory sometimes presents implementation difficulties when it is too wide, multiported, or deep to fit within the memory of an FPGA.

When deep memories do not fit within the FPGA memory limits, external RAMs must be used. This can often exacerbate pin limits. Wide memories can sometimes be handled by *slicing* wide memories into many narrower memories, each of which is implemented in a narrower physical RAM and associated FPGAs. Slicing distributes the data signals to multiple FPGAs, and can reduce the pin requirements of individual FPGAs. Slicing still necessitates the address to be routed to multiple FPGAs and memories. Multiported memories can be implemented in physical multiported memories, or they can be implemented by making multiple copies of the physical memory to allow simultaneous access. Karchmer and Rose [Karchmer94] describe packaging algorithms for area efficient and delay-minimal packing of logical memories into physical memories.

An alternative approach might make use of a memory clock to simulate the effect of a multiported memory from a single physical memory. The memory clock makes multiple transitions during each cycle of the user's design clock and allows a port to be accessed on each memory clock cycle. The memory clock can also be used to simulate a wide memory using a narrower physical memory. Note, however, that systems must support the full width of the memory bus between the clocked memory system and the user's logic, and also provide a mechanism for synchronizing user access with the memory clock.

VirtualWire systems tightly integrate memories with the virtual clock and multiplex the FPGA-memory path [Agarwal95]. Memories have an impact on various stages of VirtualWire systems software. A wide, multiported

user memory is first sliced into many multiported memory slices, each of which has a data bus width that is the same as that of the physical RAMs. The address bus and control signals are connected to each of the slices. The placer locates the memory slices in physical RAMs and a corresponding associated FPGA that is physically connected to the RAM. The placer must satisfy the constraint that the sum of the logical memory sizes assigned to each physical memory do not exceed the size of physical memory.

The VirtualWire scheduler integrates memory scheduling into the normal scheduling and routing flow. When the address is available on the FPGA physically connected to the RAM, the compiler schedules memory-slice read or write cycles. The memory scheduler is also responsible for collecting the full memory word potentially from multiple slices from multiple physical RAMs into a VirtualWire register and routing the data word so it is available where needed. The associated memory scheduling logic and state machines created according to the scheduling step are placed in FPGAs that are physically connected to the RAMs.

Memories are not the only components of an input netlist that may need to be altered to fit into a multi-FPGA system. Inside an FPGA, most signals are unidirectional, with a single source. In an input netlist, some signals (particularly buses and other tristate structures) may have multiple possible sources, and mapping such structures directly to a multi-FPGA system may be difficult, especially if the sources of the net are computed on different chips. Butts and Batcheller [Butts91] present a method for translating tristate nets into unidirectional nets. Suppose each tristate gate driving a tristate bus has a signal output O and a tristate enable signal E. All outputs O of the tristate gates are combined into a single net. Such tristate nets are restructured using AND-OR gates, by replacing each tristate gate with an AND gate whose inputs are E and O, and whose output is connected to the input of a wide OR gate (see Figure 24). The output of the OR gate becomes the value of the tristate signal. In this way, non-unidirectional signals can be fit into normal FPGA routing structures.

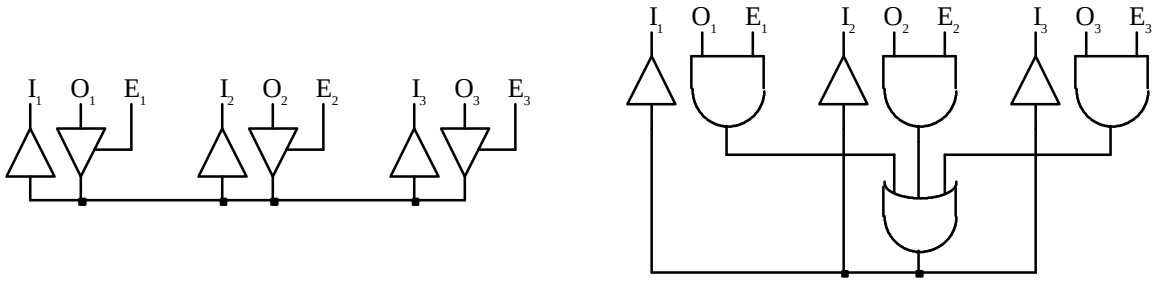

Figure 24. Tristate buses (left) in circuits mapped to multi-FPGA systems can be converted to a logic structure with only single-source nets (right) [Butts91].

5.0 Software Support for Reconfigurable Computing

Normally, digital hardware designers create circuits by manipulating Boolean logic, by interconnecting logic gates or transistors, or perhaps by writing a program in a hardware description language such as Verilog or VHDL. These methods of expressing hardware concepts are all applied to FPGA implementation. Each of these description methods is designed to represent hardware concepts efficiently, and can model most important aspects of circuit functionality.

Reconfigurable systems offer the ability to create extremely high-performance implementations for many different types of applications. While techniques such as logic emulation provide a new tool specifically for logic designers, many other uses of FPGA-based systems serve as high-performance replacements for standard computers and supercomputers. For such applications, the creators of these implementations will largely be software programmers, not hardware designers. However, if these systems hope to be useable by software programmers, the systems must be capable of translating applications described in standard software programming languages into FPGA realizations. Thus, mapping tools that can synthesize hardware implementations from C, C++, FORTRAN, or assembly language descriptions must be developed.

Although techniques exist to transform specifications written in hardware description languages into electronic circuits, translating a standard software program into hardware presents extra challenges. Hardware description

languages focus mainly only constructs and semantics that can be efficiently translated into hardware (though even these languages allow the creation of non-synthesizeable specifications). Software programming languages have no such restrictions. For example, hardware is inherently parallel, and hardware description languages have an execution model that easily expresses concurrency. Most standard software languages normally have a sequential execution model, with instructions executing one after another. This means that a hardware implementation of a software program is either restricted to sequential operation, yielding an extremely inefficient circuit, or the mapping software must figure out how to parallelize an inherently sequential specification. Also, there are operations commonly found in software programs that are relatively expensive to implement in hardware. This includes multiplication and variable-length shifts, as well as floating point operations. Although hardware can be synthesized to support these operations, software that makes extensive use of these operations will result in extremely large designs. Finally, software algorithms operate on standard-sized data values, using 8, 16, 32, or 64 bit values even for operations that could easily fit into smaller bit-widths. By using wider than necessary operands, circuit datapaths must be made wider, increasing the hardware costs. Thus, because we are using a language designed for specifying software programs to create hardware implementations the translation software faces a mapping process more complex than for standard hardware description languages.

There have been many research projects that have developed methods for translating code in C [Athanas93, Wazlowski93, Agarwal94, Wo94, Galloway95, Isshiki95, Peterson96, Clark96, Yamauchi96], C++ [Iseli95], Ada [Dossis94], Occam [Page91, Luk94], data parallel C [Gokhale93, Guccione93], Smalltalk [Pottier96], Assembly [Razdan94a, Razdan94b] or other special hardware description languages [Singh95, Brown96] into FPGA realizations. These systems typically take software programs written in a subset of the programming language, translate the data computations into hardware operations, and insert multiplexing, latches and control state machines to recreate the control flow. As mentioned earlier, software programming languages contain constructs that are difficult to handle efficiently in FPGA logic. Because of this, each of the previous techniques restricts the language constructs that can be present in the code to be translated. Most do not allow multiplication, division, or floating point operations. Some ban the use of structures, pointers, or arrays [Wazlowski93, Agarwal94, Galloway95], eliminate recursion [Galloway95] or do not support function calls [Wo94] or control flow constructs such as case, do-while, and loops without fixed iteration counts [Athanas93]. Some techniques, which are intended primarily to compile only short code sequences, may restrict data structures to only bitvectors [Iseli95], or not support memory accesses at all [Razdan94a, Razdan94b]. Other techniques, such as that of Bertin and Touati [Bertin94], extend C++ for use as a hardware description language.

Translating straight-line code from software languages into hardware is relatively simple. Expressions in the code have direct implementations in hardware that can compute the correct result (they must, since a processor on which the language is intended to run must have hardware to execute it). Variables could be stored in registers, with the result from each instruction latched immediately, and one instruction executing at a time. However, this loses the inherent concurrency of hardware. We can instead combine multiple instructions together, only latching the results at the end of the overall computation. For example, the instructions “A = B + C; D = A - E;” can be translated into “A = B + C; D = B + C - E;”, with all instructions occurring within a single clock cycle. The computation of A could also be dropped if it is not used later in the code. Note that the reassignment of a value to a variable already used, such as “B = A + 1; A = A - C; D = A + 3;”, can be a problem, since the value of A in the first and third instruction are different. Renaming is a standard compiler technique for parallelizing such code, a variant of which is contained in the Transmogrifier C compiler [Galloway95]. The compiler moves sequentially through a straight-line code sequence, remembering the logic used to create the value of each assignment. Variables in an assignment that comes from outside this piece of code draw their values from registers for that variable. Variables that have been the target of an assignment in the code sequence are replaced with the output from the logic that computes its value earlier in the sequence. Thus, the sequence “B = A + 1; A = A - C; D = A + 3;” becomes “B = A + 1; A’ = A - C; D = (A - C) + 3;”. With this logic construction the computation for all variables can occur simultaneously. All the results would then be latched simultaneously, allowing them to be stored for further computations.

Because of the way assignments are handled, simple “if” statements can also be folded into a logic equation. For example, the C statement “if (condition) A = B;” becomes “A’ = (condition and B) or (not(condition) and A);”. The condition of the “if” statement turns into a multiplexor, assigning the result of the “then” clause to the variable

if the condition is true, or the "else" clause (or old value if there is no "else" clause) if the condition is false. In this way, the jumps that would be necessary to implement the "if" statement are removed, increasing the size of straight line code segments. Obviously, if the "if" clause contains function calls or memory accesses (operations which may have side effects) this transformation cannot be done, and the "if" statement must be handled as more general control flow.

Loop unrolling can be used to turn loops with a constant number of iterations into straight line code. If a loop will be executed N times, it is converted into N copies of the loop body, concatenated from iteration 1 to iteration N. If the loop index is used inside the code, the index is assigned the appropriate value before each loop body. Also, local variables may need to be renamed within each loop body. The transformations described above can then be applied to execute all of the instructions from the loop bodies in parallel.

For more complex control flow, such as loops which execute a variable number of times or "if" statements containing function calls, control state machines must be implemented (see Figure 25). The code is broken into blocks of straight-line code. Each of these code segments is represented by a separate state in the control state machine. The straight-line code is converted into logic functions as described above, with the series of operations collapsed into a combinational circuit. Variables whose scope spans code segments are held in registers. The input to the register is a mux which combines the logic for assignment statements from different code sequences. For example, if the assignment statement "A = X;" is in code sequence 1, and "A = Y;" is in code sequence 2, then the logic driving the mux is "A = (X and state1) or (Y and state2) or (A and not (state1 or state2));", where state1 is the state corresponding to code sequence 1, and state2 corresponds to code sequence 2. If a given code sequence uses that variable, the register's output is fed to the corresponding logic.

```
/* state s1 */
x = y;
while (x) {
   /* state s2 */
   x = x - 2;
}
/* state s3 */
x = 12;
```

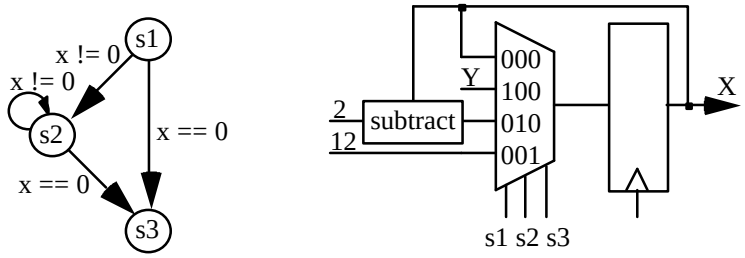

Figure 25. Example of variables and complex control flow. The code segment (left) is converted into a control state machine (center) and logic for controlling the variable's register (right). Although the states are shown in the code, these are automatically inferred from the control flow.

The control state machine is constructed to mimic the control flow in the source code. A "for" loop (where the loop test is executed before every iteration of the loop, including the first) is translated into a state for the code before the loop, one for the loop test, another for the loop body, and a final state for the code following the loop. The state machine has the test state always following both the pre-loop and loop body state, while the loop body state follows the test state if the loop test is true, and the post-loop state follows the test state if the test is false. In this way, loops without fixed loop bounds can be directly implemented in hardware. Complex "if" statements can be handed similarly; The "then" and "else" bodies each have their own state, with the "then" state occurring after the pre "if" state if the clause is true, and the "else" state occurring otherwise. Both of these states then lead to the post "if" state.

Function calls in the code require more careful construction. One might be tempted just to inline the function, making it part of the caller's code. However, this means that some frequently used code sequences will be duplicated in many places, increasing the size of the hardware. An alternative is contained in the Transmogrifier C compiler [Galloway95]. Each function has its own control state machine, with a unique entry and exit state. The caller has a state before and after the function call, and once it reaches the state before the function call it tells the function's state machine to start in its entry state. The caller then waits for the function to reach its exit state, at which point the caller moves on to its state for after the function call. Although multiple callers will all see the function reach its exit state, since only one of the callers will be active in its pre function call state, only the current caller will proceed to its next state. Recursion is normally not allowed because of this construction (since multiple

callers of the same function might be waiting in the function call state at the same time), as well as the need to have separate logic for each recursive call to the function body.

```
func (int X, int Y)
{
   /* state f1 */
   x = y;
   /* state f2 */
}

main()
{
   /* state s1 */
   func(a, 12);
   /* state s2 */
   . . .
   /* state s100 */
   func (b, 19);
   /* state s101 */
}
```

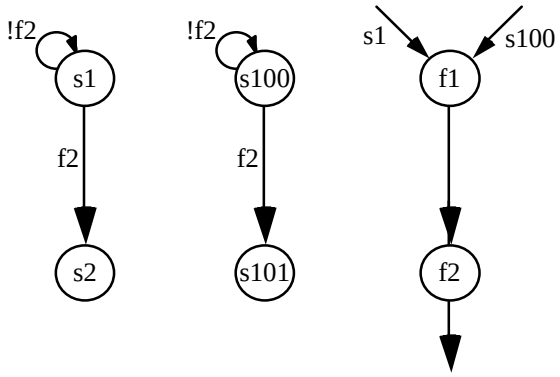

Figure 26. A code segment with a function call (left) is converted into a separate FSM for each procedure (right), with appropriate communication between components.

Once the controller for the hardware is constructed, techniques can be applied to simplify this state machine [Agarwal94]. States can be combined together sequentially or in parallel, allowing greater concurrency in the hardware, as well as minimizing the hardware cost of the controller. However, an even more important issue is the simplification of the datapath, something that hasn't yet been adequately addressed. In the construction given above every operation in the source code generates unique hardware. For simple computations this is fine. However, complex operations such as multiplication and division will be scattered throughout the source code, implying that a huge amount of hardware will be needed to implement all of these computations. However, each of the multipliers will only be used within the state corresponding to that portion of the source code, and otherwise will sit idle. The circuit's size could be greatly reduced if these hardware multipliers were reused in different places in the code. A single hardware multiplier can be used for many separate multiplication operations from the source code, as long as each of the multiplications occurs in different states. Thus, a 1000 line program, which has 50 multiplications, would require 50 separate hardware multipliers in the current scheme. However, the 1000 line program might be broken into 100 different states, with at most 4 multiplications occurring in any one state. With the proper muxing of the inputs to the hardware multipliers based on the state, only 4 hardware multipliers are actually required. Methods for performing such alterations are already available for the compilation of hardware description languages, and similar transformations are important for efficiently converting software programming languages into FPGA realizations.

There are several different systems that convert code sequences in software programming languages into hardware realizations. While they all support the translation of control flow and datapath operations into circuits, they differ on the amount of code they convert, and the model of operation they assume. Perhaps the most straightforward is the Transmogrifier C system [Galloway95]. This system takes a complete program written in the C programming language and translates it into a circuit directly implementable in the CLBs of Xilinx FPGAs. Special pragma (compiler directive) statements are included to declare external inputs and outputs, including the assignment of these communications to specific FPGA pins. This yields a system where the resulting hardware implementation is expected to be a complete, self-contained system implementing the entire functionality of the desired behavior.

Most of the systems for translating software programs into hardware algorithms assume that only the most time-critical portions of the code are mapped to the FPGAs [Athanas93, Wazlowski93, Agarwal94, Razdan94a, Razdan94b, Wo94, Iseli95, Clark96]. These systems use the FPGA(s) as a coprocessor to a standard CPU. The processor implements most of the program, handling much of the operations that are necessary to implement

complex algorithms, but which contribute little to the total computation time. The truly time-critical portions of the algorithm are translated into hardware, using the FPGA to implement the small fraction of the total code complexity which accounts for most of the overall runtime. In this way the strengths of both FPGAs and standard processors are combined into a single system. Processors can easily implement a large variety of operations by working through a complex series of instructions stored in high-density memory chips. Mapping all of these instructions into FPGA logic means that the complete functionality of the entire program must be available simultaneously, using a huge amount of circuit space. However, we can implement only the most frequently used portions of code inside the FPGA, achieving a significant performance boost with only a small amount of hardware. During the execution of the program the processor executes the software code until it hits a portion of the code implemented inside the FPGA coprocessor. The processor then transfers the inputs to the function to the FPGA coprocessor and tells it to begin computing the correct subroutine. Once the FPGA has computed the function, the results are transferred back to the processor, which continues on with the rest of the software code. An added benefit of the coprocessor model is that the software to hardware compiler does not have to support all operations from the software programming language, since complex functions such as multiplication or memory loads can be handled instead by the host processor. This does limit the portions of the code that can be translated into hardware. However, a system which converts the complete program into hardware must either convert these operations into FPGA realizations, yielding much larger area requirements, or must ban these constructs from the source code, limiting the types of operations and algorithms that can be supported.

Although compiling only the critical regions of a software algorithm can reduce the area requirements, and avoid hardware-inefficient operations, it does introduce problems unique to these types of systems. One problem is that there must be some mechanism introduced for communicating operands and results between the processor and the coprocessor. For systems like PRISC [Razdan94a, Razdan94b], which view the FPGA as merely a mechanism for increasing the instruction set of the host processor, instructions are restricted to reading from two source registers, and writing one result register, just like any other instruction on the processor. However, other systems have much less tightly coupled FPGAs, and require protocols between the two systems. In the PRISM systems [Athanas93, Wazlowski93, Agarwal94] up to 64 bits may be sent across the bus to the coprocessor, which then begins computing. In both of these systems, the communication mechanism puts a hard limit on the amount of information that can be communicated, and thus the amount of the computation that can be migrated to hardware. For example, if only two input words and a single output word are allowed, there is obviously only so much useful computation that can be performed in most circumstances.

A second important issue with compiling only a portion of a program into hardware is determining which portions of the code to so map. Obviously, the code that gets mapped to the hardware needs to contain a large portion of the runtime of the overall algorithm if we hope to achieve significant performance improvements. However, it is difficult to determine where the critical portions of a program are strictly from the source code. In general, the solutions are to profile the execution of the software on a sample input set to find the most frequently executed code sequences [Razdan94a, Razdan94b], or to have the user pick the code sequences to use by hand [Athanas93, Wazlowski93, Agarwal94, Wo94]. However, simply identifying the most often executed code sequences may not yield the best speedups. Specifically, some code sequences will achieve higher performance improvements than others when mapped to hardware. For example, even though a given code sequence occurs twice as much as another, if the achieved speedup is only a quarter of what is possible for the other sequence the other sequence is the better routine to optimize. Also, some code sequences may be too complex to map into hardware, not fitting within the logic resources present in the FPGA. The PRISC compiler [Razdan94a, Razdan94b] performs some simple checks to see whether a given code sequence will fit in the FPGA resources, and will reduce the amount of code translated if the current selection will not fit. However, in general greater automation of this decision process is necessary.

6.0 Software Support for Run-Time Reconfigurability

Perhaps one of the types of systems most unique to FPGAs is run-time reconfigurability. The idea is that since an FPGA is reconfigurable, its programming can be changed while the system is still executing. Thus, like the virtual memory in a normal processor, a run-time-reconfigurable FPGA can be viewed as “Virtual Hardware”, paging

hardware into the FPGA only when it is needed, and allowing large systems to be squeezed into a relatively small amount of physical hardware.

While the concept of run-time reconfigurability is quite promising, there are many issues that must be considered in developing such a system. Since the configuration of an FPGA can change dynamically, and in fact some portions of the FPGA may be changing while others are performing useful work, it becomes difficult to understand the exact behavior of the system. The logic must be designed so that it only accesses those logic subcircuits that are currently loaded into the FPGA, and must properly ignore spurious signals from unconfigured or reconfiguring sections of the FPGA.

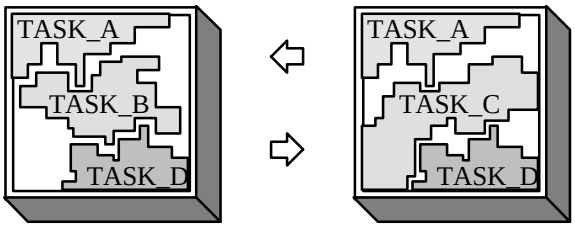

Figure 27. Conceptual diagram of a run-time reconfigurable system, with fixed logic tasks "A" and "D", and tasks "B" and "C" which are swapped in and out during the operation of the circuit [Lysaght96].

As an aid in the development of run-time reconfigurable systems, Lysaght and Stockwood have developed a simulation approach for these systems [Lysaght96]. In their methodology, a circuit's VHDL description is annotated with information on what signals will cause a given subcircuit to be loaded into hardware, and how long the reconfiguration process takes. This initial description is then automatically transformed into a new description with explicit isolation switches between the subcircuits controlled by schedulers based upon the circuit annotations. Whenever a given component should not be present in the FPGA, or has not yet been completely loaded into the chip, these isolation switches set the value of all of that subcircuit's signals to unknown ("X"). Thus, any circuitry that is (erroneously) sensitive to the value of signals from circuitry that is not currently loaded will also go to the unknown state, demonstrating the logical flaw. When the circuit is properly loaded into the FPGA, the isolation switches simply forward on all signal values. This new representation of the circuit can now be simulated normally, allowing standard software simulators to support the execution of run-time-reconfigurable systems. Another method for simulating run-time reconfigurable systems in VHDL has been proposed by Kwait and Debany [Kwait96].

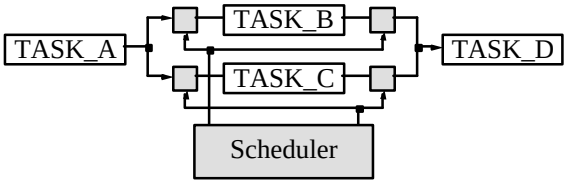

Figure 28. Simulation model for the circuit of Figure 27. Isolation switches (grey squares) and a reconfiguration scheduler are added to the base VHDL description to mimic the reconfiguration operations [Lysaght96].

The modeling of unconfigured or reconfiguring outputs with the unknown value allows the user to test the logical correctness of their design. However, one issue that is not addressed by this is the electrical correctness of a circuit. Specifically, in many FPGAs there are long-distance communication lines which have tristate drivers allowing multiple different logic blocks to drive the line. Normally, when developing a configuration for an FPGA the design software ensures that at most one logic block is configured to drive a given wire. Otherwise, with multiple writers to a single wire a direct power-ground connection can be set up, potentially overheating and destroying the chip. However, with dynamic reconfiguration the writer of the wire may change, since different configurations may require a different interconnection scheme. As long as configurations with different writers to the same line are never allowed to coexist in the chip, this is a legal design. However, it may be difficult to determine whether

two configurations, with conflicting writer assignments, will ever coexist in the chip. Even worse, when the system switches between configurations, parts of the new and the old configuration may coexist briefly, since reconfiguring an FPGA is not an instantaneous, atomic process. To deal with this, the Lysaght and Stockwood work [Lysaght96] suggests that an old configuration first be overwritten by a "safe" configuration before loading the new configuration. Thus, this "safe" configuration would turn off all drivers to shared resources, avoiding the partial configuration conflicts. However, the more general issue of how to ensure that two configurations with conflicting writer assignments are never loaded simultaneously is still an important open problem.

There are many other open problems in the design of run-time reconfigurable systems, with almost no current solutions. Current software support for standard reconfigurable systems is largely useless for this domain, since they do not have any support for time-varying mappings. To some extent, the challenges of run-time reconfigurability are similar to those of self-modifying code. Since the configuration changes over time, under the control of the configuration itself, there needs to be some way to make sure that the system behaves properly for all possible execution sequences. This is in general a challenging problem to address.

Beyond just issues of making sure that a run-time reconfigurable system behaves properly, producing a functionally and electrically correct implementation of the desired behavior, there are some major issues in how to produce the best possible implementation of the functionality. With normal FPGA-based systems, one wants to map the logic spatially so that it occupies the smallest area, and produces results as quickly as possible. In a run-time reconfigurable system one must also consider the time to reconfigure the system, and how this impacts the performance of the system. Configuration can take a significant amount of time, and thus reconfiguration should be kept to a minimum. Thus, it will be critical to group logic together properly so that a given configuration can do as much work as possible, allowing a greater portion of the task to be completed between reconfigurations. One possibility already implemented is to group together all instructions from the same function in a software algorithm and put them into their own separate context [Gokhale95]. However, what to do when function boundaries are too large or too small is still unclear. Techniques also need to be developed for limiting the amount of data that must be changed during reconfiguration. One possibility is to better encode the configuration information, requiring fewer bits to configure the FPGA, and thus reduce the reconfiguration time [DeHon96]. Another is to maximize similarity between configurations. For example, if several configurations all require an 8-bit multiplier, and that multiplier is placed into the same location in the FPGA in all configurations, then this portion of the FPGA will not need to be reconfigured when switching between these configurations. Such techniques may be able to significantly reduce the reconfiguration overhead, improving the performance of run-time reconfiguration, and thus increasing its applicability.

7.0 Conclusions

While hardware substrates are an important area of research for reconfigurable computing, efficient software support for automatically mapping applications to these systems is critical to their being widely accepted. For some of these tasks standard single-FPGA mapping tools can be harnessed for reconfigurable systems. Reconfigurable systems must perform technology mapping, placement and routing, which requires similar optimizations to normal FPGAs. However, for complete reconfigurable systems other optimizations must be performed, and these optimizations must handle the constraints of reconfigurable systems. For example, multi-FPGA systems have fixed inter-FPGA routing structures, requiring careful consideration of partitioning, global placement, and global routing tasks. Also, logic restructuring may need to be performed to alter the input netlist into a form that can be accommodated within the reconfigurable system. Finally, limited inter-chip bandwidth and scalability considerations may require special optimizations, such as the VirtualWires methodology for time-multiplexing FPGA I/O pins and scheduling signals according to a discrete time base.

Reconfigurable mapping software differs from standard hardware compilers and CAD algorithms not just because of the constraints of FPGA-based implementations, but also due to the types of users these systems hope to support. For reconfigurable computing, a reconfigurable system is used as a replacement for a standard computer, providing a high-performance implementation medium. Accordingly, these systems must support input formats familiar to software programmers, including C and other standard software languages. As we discussed, there are several systems for converting standard software algorithms into FPGA-based implementations. However, most of these

systems are quite restricted, limiting the constructs allowed in the input. Also, for systems which include both reconfigurable and standard microprocessor components, tools are needed to determine which parts of an algorithm should be mapped into hardware, and which should remain software; Currently, such techniques have not been completely developed.

Run-time reconfigurable systems face even greater challenges. Allowing the configuration of the FPGAs in a system to change over time, even to the level of changing subregions of the FPGA while the rest of the system continues operating, creates many new issues for software tools to address. While there have been some tools developed for simulating these systems, methods for detecting erroneous configurations, automatically scheduling reconfiguration, and placing logic to minimize the amount of reconfiguration necessary, all still need to be developed.

In general, there are significant opportunities for future innovating in software support for reconfigurable systems. Techniques must be developed for performing partitioning and global routing in arbitrary topologies. Current tools tend to be focused on specific topologies (primarily crossbars and partial crossbars), limiting the types of systems that can be supported. Mapping tools for structured netlists may also be able to greatly improve reconfigurable systems; Current tools treat the input circuit as random logic, missing opportunities for optimization based on the structure of regular datapaths, hierarchical circuits, and other non-random components.

Perhaps one of the most unique considerations of reconfigurable systems, as opposed to standard hardware implementations, is the need for high-performance mapping tools. Normal VLSI implementations take months to develop the circuit design, and weeks for the fabrication. Thus, mapping times of a few days are acceptable. For reconfigurable systems, systems which can be programmed to implement a desired behavior in a matter of microseconds, mapping times of days limit the usefulness of these systems. For logic emulation systems, which are used in a testing-debug-retesting methodology, mapping times of a day or more are common, and thus limit the turnaround time for design changes. For custom computing systems, software programmers accustomed to compile times of minutes to tens of minutes may be unwilling to accept mapping times an order of magnitude longer. Thus, along with the development of algorithms to support tasks not covered by current software, development of faster mapping times will be a primary concern for future reconfigurable system compilers. Techniques such as incremental update, where small changes in an input circuit cause only minor recompilation of the mapping, can greatly speed up mapping tools. Also, tools which can trade quality for compile time may be useful for this domain, allowing the circuit developer the choice of how long to wait for their implementation. Of course, new FPGA architectures that admit better compile times and are better suited for reconfigurable computing must also be developed.

The move towards automating the mapping process must always strike a balance between ease of design and the quality degradation of the results. In a system like a logic emulator it is clear that a fully automatic mapping system is the only reasonable method for using such a system. However, for custom-computing devices there will always be room for hand-mapping of applications, just as in general-purpose software development some critical components will always be written in assembly language. The hope in the FPGA-based computing community is to develop mapping tools that achieve a level of quality approaching that of hand-mapping, and transition many FPGA-based systems to a partially or wholly automatic mapping system. How successful such efforts are depends largely on the quality of the results generated. However, without such an automatic mapping solution for at least some designs, the benefits of FPGA-based systems will be denied to a large user base that is unwilling to learn how to perform the complete hand-mapping of applications to this substrate.

In summary, the challenges of reconfigurable systems are unlike those of other systems. Their restricted interconnect, demand for fast compile times, and scheduling concerns of run-time reconfiguration provide optimization goals different from standard hardware implementations, while their fine-grained parallelism and low level programmability are much different from the implementation realities of standard processors. While there has been significant work on software tools specifically optimized to reconfigurable systems, there are many remaining issues in the development of a complete hardware and software reconfigurable system.

References

[Abouzeid93] P. Abouzeid, B. Babba, M. C. de Paulet, G. Saucier, “Input-Driven Partitioning Methods and Application to Synthesis on Table-Lookup-Based FPGA's”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, July 1993.

[Adé94] M. Adé, R. Lauwereins, M. Engels, J. A. Peperstraete, “Communication Primitives on FPGAs in a Heterogeneous Multi-processor Emulation Environment”, in W. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 170-181, 1994.

[Agarwal91] A. Agarwal, “Limits on Interconnection Network Performance”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 398-412, October, 1991.

[Agarwal94] L. Agarwal, M. Wazlowski, S. Ghosh, “An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs”, *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 101-110, 1994.

[Agarwal95] A. Agarwal, "VirtualWires: A Technology for Massive Multi-FPGA Systems", http://www.ikos.com/products/virtualwires.ps, 1995.

[Alpert95] C. J. Alpert, A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration: the VLSI Journal*, Vol. 19, No. 1-2, pp. 1-81, 1995.

[Aptix93] Aptix Corporation, *Data Book*, San Jose, CA, February 1993.

[Arnold94] J. M. Arnold, D. A. Buell, “VHDL Programming on Splash 2”, in W. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 182-191, 1994.

[Athanas93] P. M. Athanas, H. F. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis”, *IEEE Computer*, Vol. 26, No. 3, pp. 11-18, March, 1993.

[Babb93] J. Babb, R. Tessier, A. Agarwal, “Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators”, *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 142-151, 1993.

[Bertin94] P. Bertin, H. Touati, “PAM Programming Environments: Practice and Experience”, *Proceedings of the Second IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 133-138, April 1994.

[Brayton85] R. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic, MA, 1985.

[Brown92a] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, *Field-Programmable Gate Arrays*, Boston, Mass: Kluwer Academic Publishers, 1992.

[Brown92b] S. Brown, J. Rose, Z. G. Vranesic, “A Detailed Router for Field-Programmable Gate Arrays”, *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 5, pp. 620-628, May 1992.

[Brown96] G. Brown, A. Wenban, “A Software Development System for FPGA-Based Data Acquisition Systems”, *Proceedings of the Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[Butts91] M. R. Butts, J. A. Batcheller, *Method of Using Electronically Reconfigurable Logic Circuits*, United States Patent 5,036,473, July 30, 1991.

[Cai91] Y. Cai, D. F. Wong, “Optimal Channel Pin Assignment”, *IEEE Transaction on Computer-Aided Design*, Vol. 10, No. 11, pp. 1413-1424, Nov. 1991.

[Chan93] P. K. Chan, M. D. F. Schlag, “Architectural Tradeoffs in Field-Programmable-Device-Based Computing Systems”, *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 152-161, 1993.

[Chan95] P. K. Chan, M. D. F. Schlag, J. Y. Zien, “Spectral-Based Multi-Way FPGA Partitioning”, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 133-139, 1995.

[Chang96] S.-C. Chang, M. Marek-Sadowska, T.-T. Hwang, “Technology Mapping for TLU FPGA's Based on Decomposition of Binary Decision Diagrams”, *IEEE Transactions on Computer-Aided Design of Integrated Ciruciuts and Systems*, Vol. 15, No. 10, October 1996.

[Chen95] C.-S. Chen, Y.-W. Tsay, T.-T. Hwang, A. C.H. Wu, Y.-L. Lin, “Combining Technology Mapping and Placement for Delay-Minimization in FPGA Designs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 9, September 1995.

[Chou94] N.-C. Chou, L.-T. Liu, C.-K. Cheng, W.-J. Dai, R. Lindelof, "Circuit Partitioning for Huge Logic Emulation Systems", *Proceedings of the 31st Design Automation Conference*, pp. 244-249, 1994.

[Clark96] D.A. Clark, B.L. Hutchings, "The DISC Programming Environment", *Proceedings of the Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[Cong91] J. Cong, "Pin Assignment with Global Routing for General Cell Designs", *IEEE Transaction on Computer-Aided Design*, Vol. 10, No. 11, pp. 1401-1412, Nov. 1991.

[Cong92] J. Cong, Y. Ding, "An Optimal Technology Mapping Algorithm For Delay Optimization In Lookup-Table Based FPGA Design", *International Conference on Computer-Aided Design*, 1992.

[Cong93] J. Cong, Y. Ding, "On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping", Design Automation Conference, pp. 213-218, 1993.

[DeHon96] A. DeHon, "Entropy, Couting, and Programmable Interconnect", *International Symposium on Field Programmable Gate Arrays*, pp. 73-79, 1996.

[Dossis94] M. F. Dossis, J. M. Noras, G. J. Porter, "Custom Co-processor Compilation", in W. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 202-212, 1994.

[Fang96] W.-J. Fang, A. C.-H. Wu, "A Hierarchucal Functional Structuring and Partitioning Approach for Multiple-FPGA Implementations", *International Conference on Computer-Aided Design*, pp. 638-643, 1996.

[Farrahi94] A. H. Farrahi, M. Sarrafzadeh, "Complexity of the Look-up Table Minimization Problem for FPGA Technology Mapping", *IEEE Transactions on Computer Aided Design*, Vol. 13, No. 11, pp. 1319-1332, November 1994.

[Francis90] R. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays", *Design Automation Conference*, pp. 613-619, 1990.

[Francis91a] R. Francis, J. Rose, Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs", *Design Automation Conference*, pp. 227-233, 1991.

[Francis91b] R. Francis, J. Rose, Z. Vranesic, "Technology Mapping for Lookup Table-Based FPGAs for Performance", *International Conference on Computer-Aided Design*, pp. 568-571, 1991.

[Frankle92] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing", *Design Automation Conference*, pp. 539-542, 1992.

[Galloway95] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[Gokhale93] M. Gokhale, R. Minnich, "FPGA Programming in a Data Parallel C", *Proceedings of the First IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 94-102, April 1993.

[Gokhale95] M. Gokhale, A. Marks, "Automatic Synthesis of Parallel Programs Targetted to Dynamically Reconfigurable Logic Arrays", in W. Moore, W. Luk, Eds., *Field-Programmable Logic and Applications*, Berlin, Germany: Springer, pp. 399-408, 1995.

[Guccione93] S. A. Guccione, M. J. Gonzalez, "A Data-Parallel Programming Model for Reconfigurable Architectures", *Proceedings of the First IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 79-87, 1993.

[Hauck94] S. Hauck, G. Borriello, "Pin Assignment for Multi-FPGA Systems (Extended Abstract)", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 11-13, April, 1994. The complete paper appears as University of Washington, Dept. of CSE Technical Report #94-04-01, 1994.

[Hauck95a] S. Hauck, G. Borriello, "An Evaluation of Bipartitioning Techniques", *Chapel Hill Conference on Advanced Research in VLSI*, pp. 383-402, March, 1995.

[Hauck95b] S. Hauck, G. Borriello, "Logic Partition Orderings for Multi-FPGA Systems", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 32-38, February, 1995.

[Hauck98] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-639, April 1998.

[Huang95a] J. Huang, J. Jou, and W. Shen, "Compatible Class Encoding in Roth-Karp Decomposition for Two-Output LUT Architectures", *International Conference on Computer-Aided Design*, pp. 359-363, 1995.

[Huang95b] D. J.-H. Huang, A. B. Kahng, "Multi-Way System Partitioning into a Single Type or Multiple Types of FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 140-145, 1995.

[Hwang94] T.-T. Hwang, R. M. Owens, M. J. Irwin, K. H. Wang, "Logic Synthesis for Field-Programmable Gate Arrays", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 10, October 1994.

[I-Cube94] I-Cube, Inc., "The FPID Family Data Sheet", Santa Clara, CA, February 1994.

[Iseli95] C. Iseli, E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[Isshiki95] T. Isshiki, W. W.-M. Dai, "High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 167-173, 1995.

[Kadi94] M. Slimane-Kadi, D. Brasen, G. Saucier, "A Fast-FPGA Prototyping System That Uses Inexpensive High-Performance FPIC", *ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Karchmer94] D. Karchmer, J. Rose, "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems", *Proceedings of ICCAD*, pp. 20-26, 1994.

[Kernighan70] 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.

[Kim96] C. Kim, H.Shin, "A Performance-Driven Logic Emulation Systems: FPGA Network Design and Performance-Driven Partitioning", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 5, May 1996.

[Kuznar93] R. Kuznar, F. Brglez, K. Kozminski, "Cost Minimization of Partitions into Multiple Devices", *Proceedings of the 30th Design Automation Conference*, pp. 315-320, 1993.

[Kuznar94] R. Kuznar, F. Brglez, B. Zajc, "Multi-way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Cost and Interconnect", *Design Automation Conference*, pp. 238-243, 1994.

[Kuznar95] R. Kuznar, F. Brglez, "PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of Large FPGA Netlists", International Conference on Computer-Aided Design, pp. 644-649, 1995.

[Kwait96] K. Kwait, W. Debany, "Reconfigurable Logic Modeling", *Integrated Systems Design*, Vol. 8, Issue 90, pp. 18-28, December 1996.

[Legl96] C. Legl, B. Wurth, K. Eckl, "An Implicit Algorithm for Support Minimization During Functional Decomposition", *European Design and Test Conference*, 1996.

[Luk94] W. Luk, D. Ferguson, I. Page, "Structured Hardware Compilation of Parallel Programs", in W. Moore, W. Luk, Eds., *More FPGAs*, Oxford, England: Abingdon EE&CS Books, pp. 213-224, 1994.

[Luk95] W. Luk, "A Declarative Approach to Incremental Custom Computing", *Proceedings of the Third IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 164-172, April 1995.

[Lysaght96] p. Lysaght, J. Stockwood, "A Simulation Tool fpr Dynamically Reconfigurable Field Programmable Gate Arrays", *IEEE Transactions on VLSI Systems*, Vol. 4, No. 3, pp. 381-390, September 1996.

[Mak95a] W.-K. Mak, D. F. Wong, "Board-Level Multi-Terminal Net Routing for FPGA-based Logic Emulation", *International Conference on Computer-Aided Design*, pp. 339-344, 1995.

[Mak95b] W.-K. Mak, D. F. Wong, "On Optimal Board-Level Routing for FPGA-based Logic Emulation", *Design Automation Conference*, 1995.

[Mathur95] A. Mathur, C. K. Chen, C. L. Liu, "Applications of Slack Neighborhood Graphs to Timing Driven Optimization Problems in FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 118-124, 1995.

[McFarland90] M. C. McFarland, A. C. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, Vol. 78, N0. 2, pp. 301-318, February 1990.

[Murgai90] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Synthesis Algorithms for Programmable Gate Arrays", *Design Automation Conference*, pp. 620-625, 1990.

[Murgai91a] R. Murgai, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli., “Improved Logic Synthesis Algorithms for Table Look Up Architectures”, Design Automation Conference, pp. 564-567, 1991.

[Murgai91b] R. Murgai, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli, “Performance Directed Synthesis for Table Look Up Programmable Gate Arrays”, *International Conference on Computer-Aided Design*, pp. 572-575, 1991.

[Murgai94] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Optimum Functional Decomposition Using Encoding”, *Design Automation Conference*, pp. 408-414, 1994.

[Page91] I. Page, W. Luk, “Compiling Occam into FPGAs”, in W. Moore, W. Luk, Eds., *FPGAs*, Abingdon, England: Abingdon EE&CS Books, pp. 271-283, 1991.

[Peterson96] J.B. Peterson, R.B. O'Connor, P.M. Athanas, “Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures”, *Proceedings of the Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[Pförtner92] T. Pförtner, S. Kiefl, R. Dachauer, “Embedded Pin Assignment for Top Down System Design”, *European Design Automation Conference*, pp. 209-214, 1992.

[Pottier96] B. Pottier, J. Llopis, “Revisiting Smalltalk-80 blocks: A logic generator for FPGAs”, *Proceedings of the Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[Raman94] S. Raman, C. L. Liu, L. G. Jones, “A Delay Driven FPGA Placement Algorithm”, *EURO-DAC*, pp. 277-82, 1994.

[Razdan94a] R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.

[Razdan94b] R. Razdan, M. D. Smith, “A High-Performance Microarchitecture with Hardware-Programmable Functional Units”, *International Symposium on Microarchitecture*, pp. 172-180, 1994.

[Roy-Neogi95] K. Roy-Neogi, C. Sechen, “Multiple FPGA Partitioning with Performance Optimization”, *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 146-152, 1995.

[Roy94] K. Roy, *A Timing-Driven Multi-Way Partitioning System for Integrated Circuits and Multi-Chip Systems*, Ph.D. Thesis, University of Washington, Dept. of EE, 1994.

[Sample94] S. J. Sample, M. R. D'Amour, T. S. Payne, *Reconfigurable Hardware Emulation System*, United States Patent 5,329,470, July 12, 1994.

[Sawada95] H. Sawada, T. Suyama, A. Nagoya, “Logic Synthesis for Lookup Table Based FPGAs Using Functional Decomposition and Support Minimization”, *International Conference on Computer-Aided Design*, pp. 353-358, 1995.

[Selvidge95] C. Selvidge, A. Agarwal, M. Dahl, J. Babb, “TIERS: Topology IndependEnt Pipelined Routing and Scheduling for VirtualWire™ Compilation”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 25-31, 1995.

[Shahookar91] K. Shahookar, P. Mazumder, “VLSI Cell Placement Techniques”, *ACM Computing Surveys*, Vol. 23, No. 2, pp. 145-220, June 1991.

[Shen95] W. Shen, J. Huang, S. Chao, “Lambda Set Selection in Roth-Karp Decomposition for LUT-Based FPGA Technology Mapping”, *Design Automation Conference*, pp. 65-69, 1995.

[Singh95] S. Singh, “Architectural Descriptions for FPGA Circuits”, *Proceedings of the Third IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 145-154, April 1995.

[Snider95] G. Snider, P. Kuekes, W. B. Culbertson, R. J. Carter, A. S. Berger, R. Amerson, “The Teramac Configurable Compute Engine”, in W. Moore, W. Luk, Eds., *Field-Programmable Logic and Applications*, Berlin, Germany: Springer, pp. 44-53, 1995.

[Stanion95] T. Stanion, C.~Sechen, “A Method for Finding Good Ashenhurst Decomposition and its Application to FPGA Synthesis”, *Design Automation Conference*, pp. 60-64, 1995.

[Thomae91] 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.

[Togawa94] N. Togawa, M. Sato, T. Ohtsuki, “A Simultaneous Placement and Global Routing Algorithm for Symmetric FPGAs”, *ACM/SIGDA International Workshop on Field Programmable Gate Arrays*, 1994.

[Van den Bout92] D. E. Van den Bout, J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo, D. Hallman, "Anyboard: An FPGA-Based, Reconfigurable System", *IEEE Design & Test of Computers*, pp. 21-30, September, 1992.

[Van den Bout93] D. E. Van den Bout, "The Anyboard: Programming and Enhancements", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 68-77, 1993.

[Venkateswaran94] R. Venkateswaran, P. Mazumder, "A Survey of DA Techniques for PLD and FPGA Based Systems", *Integration, the VLSI Journal*, Vol. 17, No. 3, pp. 191-240, 1994.

[Vijayan90] G. Vijayan, "Partitioning Logic on Graph Structures to Minimize Routing Cost", *IEEE Trans. on CAD*, Vol. 9, No. 12, pp. 1326-1334, December 1990.

[Vincentelli93] A. Sangiovanni-Vincentelli, A. El Gamal, J. Rose, "Synthesis Methods for Field Programmable Gate Arrays", *Proceedings of the IEEE*, Vol. 81, No. 7, pp. 1057-1083, 1993.

[Vuillemin96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI Systems*, Vol. 4, No. 1, pp. 56-69, March, 1996.

[Wazlowski93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh, "PRISM-II Compiler and Architecture", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9-16, 1993.

[Weinmann94] U. Weinmann, "FPGA Partitioning under Timing Constraints", in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford: Abingdon EE&CS Books, pp. 120-128, 1994.

[Wo94] D. Wo, K. Forward, "Compiling to the Gate Level for a Reconfigurable Co-processor", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 147-154, 1994.

[Woo93] N.-S. Woo, J. Kim, "An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementations", *Design Automation Conference*, pp. 202-207, 1993.

[Wurth95] B. Wurth, K. Eckl, K. Antreich, "Functional Multiple-output Decomposition: Theory and an Implicit Algorithm", *Design Automation Conference*, pp. 54-59, 1995.

[Yamada94] K. Yamada, H. Nakada, A. Tsutsui, N. Ohta, "High-Speed Emulation of Communication Circuits on a Multiple-FPGA System", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Yamauchi96] T. Yamauchi, S. Nakaya, N. Kajihara, "SOP: A Reconfigurable Massively Parallel System and Its Control-Data-Flow based Compiling Method", *Proceedings of the Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[Yao88] X. Yao, M. Yamada, C. L. Liu, "A New Approach to the Pin Assignment Problem", *Design Automation Conference*, pp. 566-572, 1988.