Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators

Jonathan Babb, Russell Tessier, and Anant Agarwal
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

Existing FPGA-based logic emulators suffer from limited inter-chip communication bandwidth, resulting in low gate utilization (10 to 20 percent). This resource imbalance increases the number of chips needed to emulate a particular logic design and thereby decreases emulation speed, since signals must cross more chip boundaries. Current emulators only use a fraction of potential communication bandwidth because they dedicate each FPGA pin (physical wire) to a single emulated signal (logical wire). These logical wires are not active simultaneously and are only switched at emulation clock speeds.

Virtual wires overcome pin limitations by intelligently multiplexing each physical wire among multiple logical wires and pipelining these connections at the maximum clocking frequency of the FPGA. A virtual wire represents a connection from a logical output on one FPGA to a logical input on another FPGA. Virtual wires not only increase usable bandwidth, but also relax the absolute limits imposed on gate utilization. The resulting improvement in bandwidth reduces the need for global interconnect, allowing effective use of low dimension inter-chip connections (such as nearest-neighbor). Nearest-neighbor topologies, coupled with the ability of virtual wires to overlap communication with computation, can even *improve* emulation speeds. We present the concept of virtual wires and describe our first implementation, a "softwire" compiler which utilizes static routing and relies on minimal hardware support. Results from compiling netlists for the 18K gate Sparcle microprocessor and the 86K gate Alewife Communications and Cache Controller indicate that virtual wires can increase FPGA gate utilization beyond 80 percent without a significant slowdown in emulation speed.

Keywords: FPGA, logic emulation, prototyping, reconfigurable architectures, static routing, virtual wires.

1 Introduction

1.1 FPGA-based Logic Emulation

Field Programmable Gate Array (FPGA) based logic emulators are capable of emulating complex logic designs at clock speeds four to six orders of magnitude faster than even an accelerated software simulator. This performance is achieved by partitioning a logic design, described by a *netlist*, across an interconnected array of FPGAs (Figure 1). This array is connected to a host workstation which is capable of downloading design configurations, and is directly wired into the target system for the logic design. The netlist *partition* on each FPGA (termed FPGA partition throughout this paper), configured directly into logic circuitry, can then be executed at hardware speeds.

Once configured, an FPGA-based emulator is a heterogeneous network of special purpose processors, each FPGA processor specifically designed to cooperatively execute its embedded circuit partition. As parallel processors, these emulators are characterized by their interconnection topology (network), target FPGA (processor), and supporting software (compiler). The interconnection topology describes the arrangement of FPGA devices and routing resources (i.e. full crossbar, two dimension mesh, etc.). Important target FPGA properties include gate count (computational resources), pin count (communication resources), and mapping efficiency. Supporting software is extensive, combining netlist translators, logic optimizers, technology mappers, global and FPGA-specific partitioners, placers, and routers.

This paper presents a compilation technique to overcome device pin limitations using *virtual wires*. This method can be applied to any topology and FPGA device, although some benefit substantially more than others.

1.2 Pin Limitations

In existing architectures, both the logic configuration and the network connectivity remain fixed for the duration of the emulation. Each emulated gate is mapped to one FPGA equivalent gate and each emulated signal is allocated to one

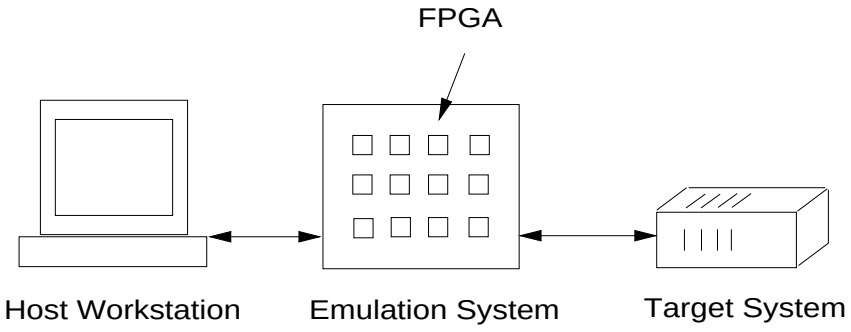

Figure 1: Typical Logic Emulation System

FPGA pin. Thus for a partition to be feasible, the partition gate and pin requirements must be no greater that the available FPGA resources. This constraint yields the following possible scenarios for each FPGA partition:

1. Gate limited: no unused gates, but some unused pins.
2. Pin limited: no unused pins, but some unused gates.
3. Not limited: unused FPGA pins and gates.
4. Balanced: no unused pins or gates.

For mapping typical circuits onto available FPGA devices, partitions are predominately pin limited; all available gates can not be utilized due to lack of pin resources to support them. For example Figure 9 in Section 4 shows that for equal gate counts in the FPGA partitions and FPGA devices, the required pin counts for FPGA partition sizes of our sample designs are much greater than the available FPGA device pin counts. Low utilization of gate resources increases both the number of FPGAs needed for emulation and the time required to emulate a particular design. Pin limits set a hard upper boundary on the maximum usable gate count any FPGA gate size can provide. This discrepancy will only get worse as technology scales; trends (and geometry) indicate that available gate counts are increasing faster than available pin counts.

1.3 Virtual Wires

To overcome pin limitations in FPGA-based logic emulators,[1] we propose the use of virtual wires. A virtual wire represents a simple connection between a logical output on one FPGA and a logical input on another FPGA. Established via a pipelined, statically routed [12] communication network, these virtual wires increase available off-chip communication bandwidth by multiplexing the use of FPGA pin resources (physical wires) among multiple emulation signals (logical wires).

[1] Although this paper focuses on logic emulators, virtual-wire technology can be employed in any system comprising multiple interconnected FPGAs.

Virtual wires effectively relax pin limitations. While low pin counts may decrease emulation speed, there is no longer a hard pin constraint which must be enforced. Emulation speed can potentially be increased if there is a large enough reduction in system size. We demonstrate that the gate overhead of using virtual wires is low, comprising gates which could not have been utilized anyway in the purely hardwired implementation. Furthermore, the flexibility of virtual wires allows the emulation architecture to be balanced for each logic design application.

Our results from compiling two complex designs, the 18K gate Sparcle microprocessor [2] and the 86K gate Alewife Communications and Cache Controller [11] (A-1000) show that the use of virtual wires can decrease FPGA chip count by a factor of 3 for Sparcle and 10 for the A-1000, assuming a crossbar interconnect. With virtual wires, a two dimensional torus interconnect can be used for only a small increase in chip count (17 percent for the A-1000 and 0 percent for Sparcle). Without virtual wires, the cost of a replacing the full crossbar with a torus interconnect is over 300 percent for Sparcle, and practically impossible for the A-1000. Emulation speeds are comparable with the no virtual wires case, ranging from 2 to 8 MHZ for Sparcle and 1 to 3 MHZ for the A-1000. Neither design was bandwidth limited, but rather constrained by its critical path. With virtual wires, use of a lower dimension network reduces emulation speed proportional to the network diameter; a factor of 2 for Sparcle and 6 for the A-1000 on a two dimensional torus.

1.4 Background

FPGA-based logic emulation systems have been developed for design complexity ranging from several thousand to several million gates. Typically, the software for these systems is considered the most complex component and comprises a major portion of system costs. Quickturn Inc. [14] [13] has developed emulation systems which interconnect FPGAs in a two-dimensional mesh and, more recently, in a partial crossbar topology. The Quickturn Enterprise system uses a hierarchical approach to interconnection. The Virtual ASIC system by InCA [9] uses a combination of nearest neigh-

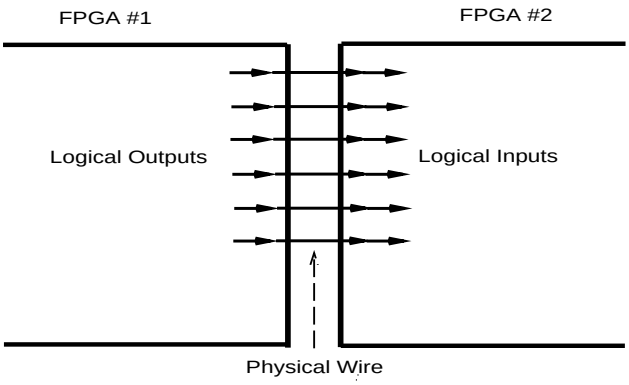

Figure 2: Hard Wire Interconnect

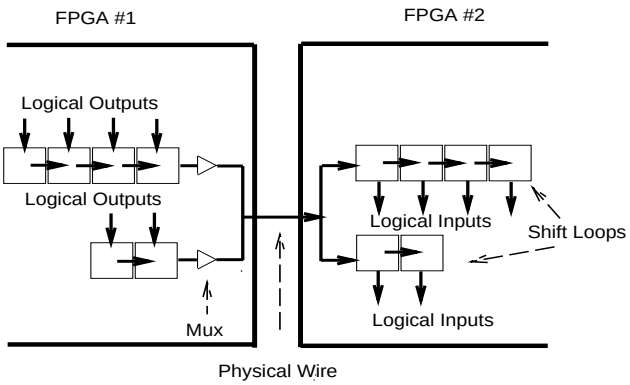

Figure 3: Virtual Wire Interconnect

bor and crossbar interconnect. Like Quickturn's systems, Virtual ASIC logic partitions are hardwired to FPGAs following partition placement. AnyBoard, developed at North Carolina State University, [6] is targetted for logic designs of a few thousand gates.

Statically routed networks can be used whenever communication can be predetermined. *Static* refers to the fact that all data movement can be determined and optimized at compile-time. This mechanism has been used in scheduling real-time communication in a multiprocessor environment [12]. Other related uses of static routing include FPGA-based systolic arrays, such as Splash [7], and in the very large simulation subsystem (VLSS) [15], a massively parallel simulation engine which uses time-division multiplexing to stagger logic evaluation.

Virtual wires are similar to virtual channels [5], which decouple resource allocation in dynamically-routed networks, and to virtual circuits [3] found in a connection-oriented network.

1.5 Overview

The rest of this paper is organized as follows: Section 2 describes the basic ideas behind virtual wires. Section 3 outlines the key components of our initial system, including softwire compiler and hardware support. In Sections 4 we analyze experimental results for compiling two current designs to various interconnect topologies and FPGA device sizes. Finally, Section 5 summarizes our research and outlines directions for future research.

2 Virtual Wires

One to one allocation of emulation signals (logical wires) to FPGA pins (physical wires) does not exploit available off chip bandwidth because:

- emulation clock frequencies are one or two orders of magnitude lower than the potential clocking frequency of the FPGA technology.
- all logical wires are not active simultaneously.

By pipelining and multiplexing physical wires, we can create virtual wires to increase usable bandwidth. By clocking physical wires at the maximum frequency of the FPGA technology, several logical connections can share the same physical resource. Figure 2 shows an example of six logical wires allocated to six physical wires. Figure 3 shows the same example with the six logical wires sharing a single physical wire. The physical wire is multiplexed between two pipelined *shift loops* (see section 3.3.1).

Systems based on virtual wires exploit several properties of digital circuits to boost bandwidth using available pins. In a logic design, evaluation flows from system inputs to system outputs. In a synchronous design with no combinatorial loops, this flow can be represented as a directed acyclic graph. Thus, through intelligent dependency analysis of the underlying logic circuit, logical values between FPGA partitions only need to be transmitted once (see section 3.2.3). Furthermore, since circuit communication is inherently static, communication patterns will repeat in a predictable fashion. By exploiting this predictability, communications can be scheduled to increase the utilization of pin bandwidth.

In our first implementation, we support virtual wires with a "softwire" compiler. This compiler analyzes logic signal dependencies and statically schedules and routes FPGA communication. These results are then used to construct (in the FPGA technology) a statically routed network. This hardware consists of a sequencer and shift loops. The sequencer is a distributed finite state machine. It establishes virtual connections between FPGAs by strobing logical wires into special shift registers, the shift loops. Shift loops are then alternately connected to physical wires according to a predetermined schedule.

While this paper focuses on logic emulation, we believe that the technique of virtual wires is also applicable to other areas of reconfigurable logic.

2.1 Limitations and Assumptions

The use of virtual wires is limited to synchronous logic. Any asynchronous signals must still be "hardwired" to dedicated FPGA pins. This limitation is imposed by the inability to statically determine dependencies in asynchronous loops. Furthermore, we assume that each *combinational loop* (such as a flip-flop) in a synchronous design is completely contained in a single FPGA partition. For simplicity, this paper assumes that the emulated logic uses a single global clock.

3 System Overview

This section describes an implementation of virtual wires in the context of a complete emulation software system, independent of target FPGA device and interconnect topology. While this paper focuses primarily on software, the ultimate goal of this research is a low-cost, reconfigurable emulation system.

3.1 The Emulation Clocking Framework

The various clocks used in the virtual-wire system define a framework for system-level design with virtual wires. Let us first describe this framework based on multiple clocks (see Figure 4).

The *emulation clock* period is the clock period of the logic design being emulated. We break this clock into evaluation *phases*. We use multiple phases to evaluate the multiple FPGA partitions across which the combinational logic between flip-flops in the emulated design may be split. In other words, evaluation within each FPGA partition, followed by the communication of results to other FPGA partitions is accomplished within a phase.

A phase is divided into two parts: an evaluation portion and a communication portion. Evaluation takes place at the beginning of a phase, and logical outputs of each FPGA partition are determined by the logical inputs in the input shift loops. At the end of the phase, outputs are then sent to other FPGA partitions with the pipelined shift loops and intermediate hop stages (see section 3.3). These pipelines are clocked with a *pipeline clock* (Figure 4) at the maximum frequency of the FPGA. After all phases within an emulation clock period are complete, the emulation clock is ticked.

In contrast, hardwired systems dedicate a physical pin to a distinct wire in the circuit and let the evaluation "flow" through multiple partitions within the emulation clock period until the entire system settles. Phases in virtual wire systems allow a physical pin that is unused during some portion of the emulation clock period to be gainfully employed by other signals.

3.2 Softwire Compiler

The input to the softwire compiler consists of a netlist of the logic design to be emulated, target FPGA device characteristics, and FPGA interconnect topology. The compiler then produces a configuration bitstream which can be downloaded onto the emulator. Figure 5 outlines the compilation steps involved. Briefly, these steps include translation and mapping of the netlist to the target FPGA technology, partitioning the netlist, placing the partitions into an interconnect topology, routing the inter-node communication paths, and finally FPGA-specific automated placement and routing.

3.2.1 Translation and Mapping

The input netlist to be emulated is usually generated with a hardware description language or schematic capture program. This netlist must be translated and mapped to a library of FPGA macros. It is important to perform this operation before partitioning so that partition gate counts accurately reflect the characteristics of the target FPGAs. We can also use logic optimization tools at this point to optimize the netlist for the target architecture (considering the system as one large FPGA).

3.2.2 Partitioning

After mapping the netlist to the target architecture, it must be partitioned into logic blocks which can fit into the target FPGA. With only hardwires, each partition must have both fewer gates and fewer pins than the target device. With virtual wires, the total gate count (logic gates and virtual wiring overhead) must be no greater than the target FPGA gate count. In our current implementation we use the Concept Silicon partitioner by InCA [9]. This partitioner performs K-way partitioning with min-cut and clustering techniques to minimize partition pin counts.

3.2.3 Dependency Analysis

Since a combinatorial signal may pass through several FPGA partitions during an emulated clock cycle, all signals will not be ready to schedule at the same time. In our current implementation, we solve this problem by only scheduling a partition output once all the inputs it depends upon are scheduled. An output depends on an input if a change in that input can change the output. To determine input to output dependencies, we analyze the logic netlist, backtracing from partition outputs to determine which partition inputs they depend upon. In backtracing, we assume all outputs depend

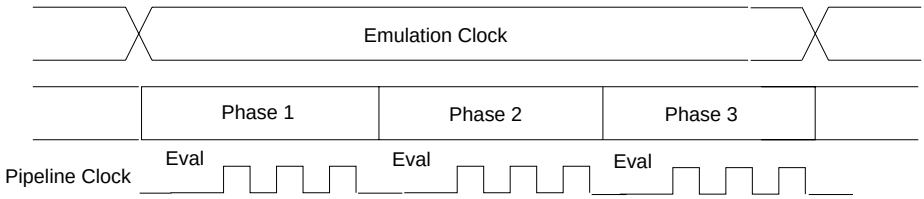

Figure 4: Emulation Phase Clocking Scheme

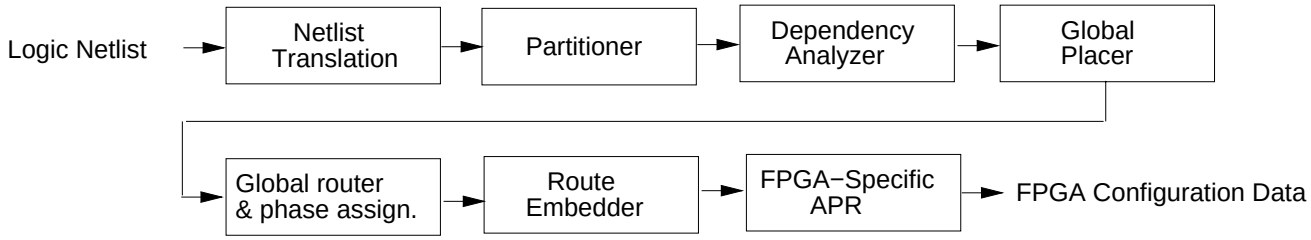

Figure 5: Softwire Tool Flowchart

on all inputs for gate library parts, and no outputs depend on any inputs for latch (or register) library parts. If there are no combinatorial loops which cross partition boundaries, this analysis produces a directed acyclic graph, the signal flow graph (SFG), to be used by the global router.

3.2.4 Global Placement

Following logic partitioning, individual FPGA partitions must be placed into specific FPGAs. An ideal placement minimizes system communication, thus requiring fewer virtual wire cycles to transfer information. We first make a random placement followed by cost-reducing swaps, and then further optimize with simulated annealing [10].

3.2.5 Global Routing and Phase Assignment

The input to the global routing and phase assignment module is a set of FPGA partitions that have been assigned to FPGA devices, and a graph describing the dependency relationships between inputs and outputs. Phase assignment and global routing schedules each logical wire to a phase and assigns a pipeline time slot on a physical pin. Thus, the assignment corresponds to one cycle of the pipeline clock (i.e., a specific register) in a specific phase (i.e., a specific shift register loop) on a physical wire between a pair of FPGAs. For simplicity, all wires in a given shift loop are assigned to a single phase.

Phase assignment uses the following methodology. Before the assignment, the criticality of each logical wire is determined based on the signal flow graph produced by dependency analysis. In each phase, the router first determines the schedulable wires. A wire is schedulable if all wires it depends upon have been scheduled in previous phases. The router then uses shortest path analysis with a cost function based on pin utilization to route as many schedulable signals as possible, routing the most critical signals first. Any schedulable signals which can not be routed are delayed to the next phase.

3.2.6 Embedding and Vendor Specific APR

Once routing is completed, appropriately-sized shift loops and associated logic are added to each partition to complete the internal FPGA hardware description. At this point there is one netlist for each FPGA. These netlists are then processed with a vendor-specific FPGA place and route software to produce configuration bitstreams.

3.3 Hardware Support

Technically, there is no required hardware support for implementation of virtual wires (unless one considers re-designing an FPGA optimized for virtual wiring). The necessary "hardware" is compiled directly into the configuration for the FPGA device. Thus, any existing FPGA-based logic emulation system can take advantage of virtual wiring. There are many possible ways to implement hardware support for virtual wires. This section describes a simple and efficient implementation. The additional logic to support virtual wires can be composed entirely of *shift loops* and a small amount of phase control logic.

3.3.1 Shift Loops

A shift loop (Figure 6) is a circular, loadable shift register with enabled shift in and shift out ports. Each shift register is capable of performing one or more of the operations of *load*, *store*, *shift*, *drive*, or *rotate*, (Figure 7). In our current

- **Load** — Strobes logical outputs into shift loop.
- **Store** — Drives logical inputs from shift loop.
- **Shift** — Shifts data from a physical input into shift loop.
- **Drive** — Drives a physical output with last bit of shift loop.
- **Rotate** —Rotate bits in shift loop.

Figure 7: Shift Loop Operations

design, for simplicity, all outputs loaded into a shift loop must have the same final destination FPGA. As described in section 3.2.3, a logical output can be strobed once all its corresponding depend inputs have been stored. The purpose of rotation is to preserve inputs which have reached their final destination and to eliminate the need for empty gaps in the pipeline when shift loop lengths do not exactly match phase cycle counts. Note that in this implementation store can not be disabled.

Shift loops can be re-scheduled to perform multiple output operations. However, since the internal latches being emulated will depend on the logical inputs, inputs will need to be stored until the tick of the emulation clock.

3.3.2 Intermediate Hop Pipelining

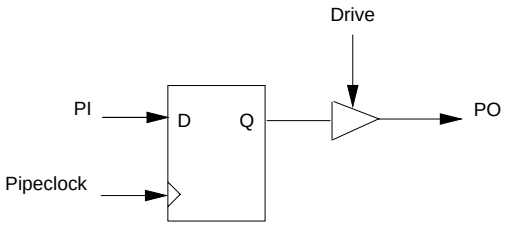

Figure 8: Intermediate Hop Pipeline Stage

For networks where multiple hops are required (i.e. a mesh), one bit shift loops which always shift and sometimes drive are used for intermediate stages (figure 8). These stages are chained together, one per FPGA hop to build a pipeline connecting the output shift loop on the source FPGA with the input shift loop on the destination FPGA.

3.3.3 Phase Control Logic

The phase control logic is the basic run-time kernel in our simple implementation. This kernel is a sequencer which controls the phase enable (denoted *drive* in Figure 6) and strobe lines (denoted *load* in Figure 6), the pipeline clock, and the emulation clock. The phase enable lines are used to enable shift loop to FPGA pin connections. Recall that multiple shift loops (including single-bit shift stages for intermediate hop pipelining) can connect to a single physical pin through tri-state drivers as depicted in Figure 3. The phase strobe lines strobe the shift loops on the correct phases. This logic is generated with a state machine specifically optimized for a given phase specification.

4 Experimental Results

We implemented the system compiler described by developing a dependency analyzer, global placer and global router and using the InCA [9] partitioner. Except for the partitioner, which can take hours to optimize a complex design, running times on a SPARC 2 workstation were usually 1 to 15 minutes for each stage.

In order to evaluate the costs and benefits of virtual wires, we compiled two complex designs, Sparcle and the A-1000. Sparcle is an 18K gate SPARC microprocessor enhanced with multiprocessing features. The Alewife controller and memory management unit (A-1000) [11] is an 86K gate cache controller for the Alewife Multiprocessor [1], a distributed shared memory machine being designed at MIT. For target FPGAs we consider the Xilinx 3000 and 4000 series (including the new 4000H series) [16] [17] and the Concurrent Logic Cli6000 series [4]. This analysis does not include the final FPGA-specific APR stage; we assume a 50 percent APR mapping efficiency for both architectures.

4.1 Virtual Wire Gate Overhead

In the following analysis, we estimate the FPGA gate costs of virtual wires based on the Concurrent Logic CLI6000 series FPGA. We assume the phase control logic is 300 gates (after mapping). Virtual wire overhead can be measured in terms of the number of gates required to implement a single shift register bit, C_s. In the Cli6000, a single-bit shift register takes 1 of 3136 cells in the 5K gate part, which implies that $C_s \approx 3$ mapped gates. For simplicity, we will also assume that each tri-state driver, which forms the multiplexer component, costs C_s.

The cost of virtual wires for an FPGA partition is the sum of three components: (1) the shift register bits required for the inputs (see section 3.3.1), (2) the shift register bits required for the intermediate hops, and (3) the tri-state drivers required to multiplex a given number of shift loops on a single physical pin. The above costs assume that the storage of logical outputs is not counted since they can be overlapped with logical inputs. When routing in a mesh or torus, intermediate hops cost one shift register bit per hop. The degree of multiplexing of a physical wire (or the number of shift loops connected to that physical wire) is the number of

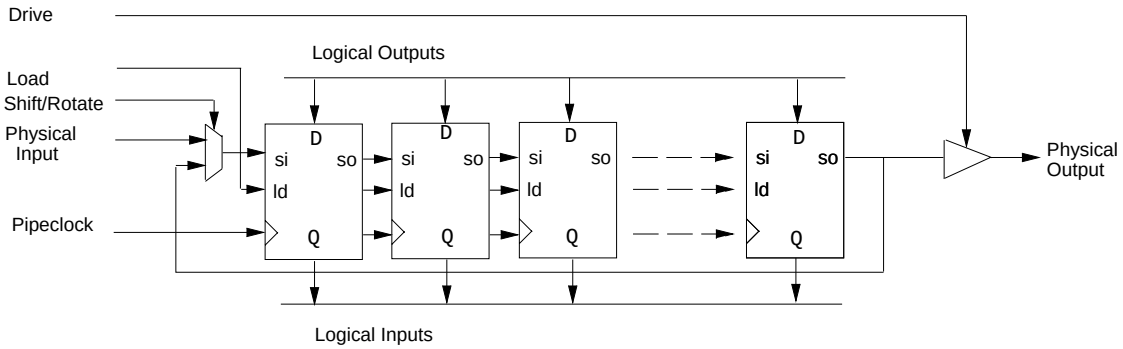

Figure 6: Shift Loop Architecture

FPGA Technology	Sparcle	A-1000
0.50	0.6	0.44

Table 1: Rent's Rule Parameter (slope of log-log curve)

tri-state drivers needed.

The gate overhead is then C_s times the total number of shift register bits. Let V_i denote the number of logical inputs for partition i, M_p denote the number of times a physical wire p is multiplexed, and L_h the number of bit shift registers used for intermediate hops in an FPGA. Gate overhead for partition i is then:

$$Gate_{vw} = C_s \times \left(V_i + L_h + \sum_p M_p \right)$$

4.2 Effect of Pin Limitations

Before compiling the two test designs, we first compared their communication requirements to the available FPGA technologies. For this comparison, we partitioned each design for various gate counts and measured the pin requirements. Figure 9 shows the resulting curves, plotted on a log-log scale (note that partition gate count is scaled to represent a mapping inefficiency of 50%).

Both design curves and the technology curves fit Rent's Rule, a rule of thumb used for estimating communication requirement in random logic. Rent's Rule can be stated as:

$$pins_2/pins_1 = (gates_2/gate_1)^b,$$

where $pins_2$, $gates_2$ refer to a partition, and $pins_1$, $gates_1$ refer to a sub-partition, and b is a constant between 0.4 and 0.7. Table 1 shows the resulting constants. For the technology curve, a constant of 0.5 roughly corresponds to the area versus perimeter for the FPGA die. The lower the constant, the more locality there is within the circuit. Thus, the A-1000 has more locality than Sparcle, although it has more total communication requirement.

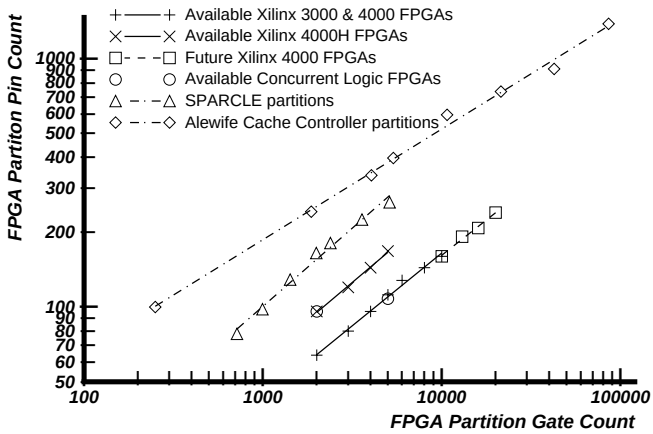

Figure 9: Pin Count as a Function of FPGA Partition Size

As Figure 9 shows, both Sparcle and the A-1000 will be pin-limited for any choice of FPGA size. In hardwired designs with pin-limited partition sizes, usable gate count is determined solely by available pin resources. For example, a 5000 gate FPGA with 100 pins can only utilize 1000 Sparcle gates or 250 A-1000 gates.

Next, we compiled both designs for a two dimensional torus and a full crossbar interconnect of 5000 gate, 100 pin FPGAs, 50 percent mapping efficiency. Table 2 shows the results for both hard wires and virtual wires. Compiling the A-1000 to a torus, hardwires only, was not practical with our partitioning software. The gate utilizations obtained for the hardwired cases agree with reports in the literature [9] [14] on designs of similar complexity.

In order to understand the tradeoffs involved, we plotted both the hard wires pin/gate constraint and the virtual wires pin/gate constraint curve against the partition curves for the two designs (Figure 10). The region enclosed by the axes and the constraint curves represents feasible regions in the design space. The intersection of the partition curves and the wire curves gives the optimal partition and sizes. This graph shows how virtual wires add the flexibility of trading

Design	*Hardwires Only*		*Virtual Wires Only*	
	2-D Torus	Full Crossbar	2-D Torus	Full Crossbar
Sparcle (18K gates)	>100 (<7%)	31 (23%)	9 (80%)	9 (80%)
A-1000 (86K gates)	Not Practical	>400 (<10%)	49 (71%)	42 (83%)

Number of FPGAs (Average Usable Gate Utilization)

Table 2: Number of 5K Gates, 100 Pin FPGAs Required for Logic Emulation

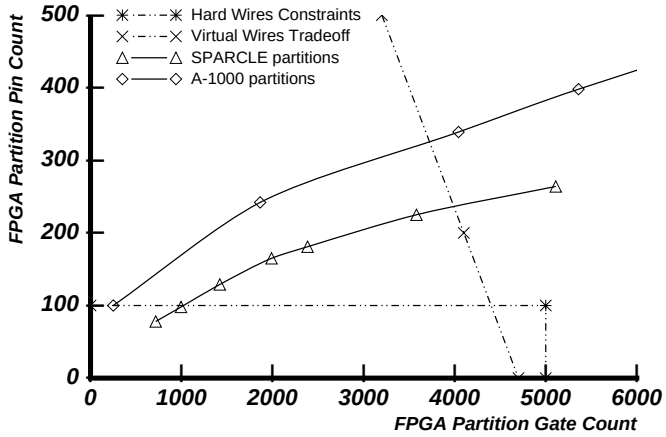

Figure 10: Determination of optimal partition size

gate resources for pin resources.

4.3 Emulation Speed Comparison

Emulation clock cycle time T_E is determined by:

- Communication delay per hop, t_c, which is the time required to transmit a single bit on a wire between a pair of FPGAs.
- Length of longest path in dependency graph, L, in terms of number of FPGA partitions (and hence phases) in an emulation clock cycle.
- Total FPGA gate delay along longest path T_L, which is the sum of the FPGA partition delays in the longest path (not counting communication time).
- Sum of pipeline cycles across all phases, N
- Network diameter, D (D = 1 for a crossbar)
- Average network distance, d (d = 1 for a crossbar)

Delays in a system are related to the number of phases in an emulation clock, and the sum of the number of pipeline clocks within each of the phases. The total number of phases L in an emulation clock is the largest number of partitions through which a combinatorial path passes. The number of pipeline cycles in each phase is directly related to physical wire contention.

If the emulation is *latency dominated*, then the optimal number of phases is L, and the pipeline cycles per phase should no greater than D, giving:

$$N = L \times D$$

The upper bound of D is imposed by the worst case number of intermediate hops.

On the other hand, if the emulation is *bandwidth dominated*, then the total pipeline cycles (summed over all phases) will be at least:

$$N = MAX_i \left(\frac{V_i}{P_i} \right)$$

where V_i and P_i are the number of virtual and physical wires for FPGA partition i. If there are hot spots in the network (not possible with a crossbar), the bandwidth dominated delay will be higher. Emulation speeds for Sparcle and the A-1000 were both latency dominated.

Although we have integrated FPGA specific placing and routing tools into our software system, we can not yet determine the exact computation time per partition. Instead we consider a *computation only* delay component, and a *communication only* delay component. This dichotomy is used to give a lower and upper bound on emulation speed.

Computation only delay: $T_{EP} = T_L + t_c \times N$, where $N = 0$ for the hardwired case. The computation-only bound assumes that communication time between chips is negligible. Even though communication is assumed to be infinitely fast, we add in a component equal to t_c to reflect the extra cost of multiplexing for virtual wires.

Communication only delay : $T_{EC} = t_c \times N$.

Based on CLi6000 specifications, we assumed that $T_L = 250ns$ and $t_c = 20ns$ (based on a 50 MHZ clock). Table 3 shows the resulting emulation speeds for virtual and

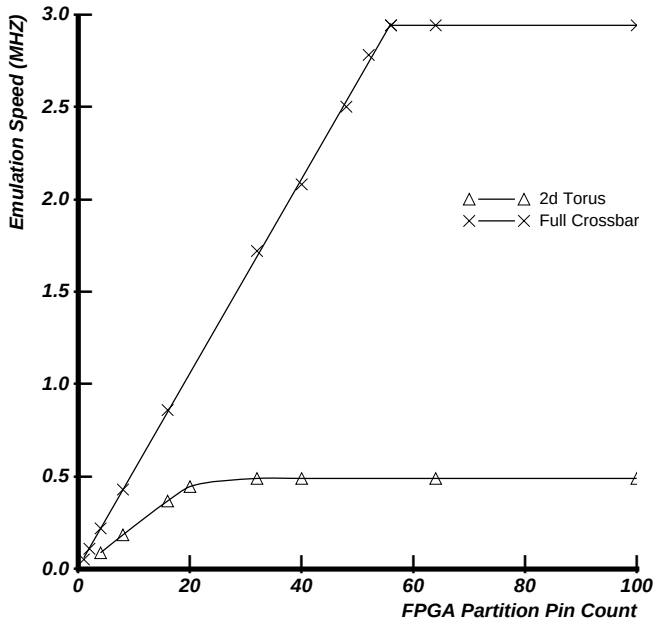

Figure 11: A-1000 Emulation Speed (Communication only Component)

hardwires for the crossbar topology. The emulation clock range given is based on the sum and minimum of the two components (lower and upper bounds). For example, the computation-only delay in Sparcle for hardwires is exactly T_L yielding $T_{EP} = 250ns$. The computation-only delay in Sparcle for virtual wires is $250 + 6 \times 20 = 370ns$. Note that we have made the conservative assumption in the computation dominated case that T_L for virtual wires remains the same as that for hardwires, even though virtual wires yields fewer partitions. When the use of virtual wires allows a design to be partitioned across less FPGAs, L is decreased, decreasing T_{EC}. However, the pipeline stages will increase T_{EP} by t_c per pipeline cycle.

In Table 3, the virtual wire emulation clock was determined solely by the length of the longest path; the communication was limited by latency, not bandwidth. In order to determine what happens when the design becomes bandwidth limited, we varied the pin count and recorded the resulting emulation clock (based on T_{EC}) for both a crossbar and torus topology. Figure 11 shows the results for the A-1000. The knee of the curve is where the latency switches from bandwidth dominated to latency dominated. The torus is slower because it has a larger diameter, D. However, the torus moves out of the latency dominated region sooner because it exploits locality; several short wires can be routed during the time of a single long wire. Note that this analysis assumes that the crossbar can be clocked as fast as the torus; the increase in emulation speed obtained with the crossbar is lower if t_c is adjusted accordingly.

4.4 Combination of Virtual Wires with Hardwiring

With virtual wires, neither design was bandwidth limited, but rather limited by its respective critical paths. As shown in Figure 11, the A-1000 only needs about 20 pins per FPGA to run at the maximum emulation frequency. While this allows the use of lower pin count (and thus cheaper) FPGAs, another option is to trade this surplus bandwidth for speed. This tradeoff is accomplished by *hardwiring* logical wires at both ends of the critical paths. Critical wires can be hardwired until there is no more surplus bandwidth, thus fully utilizing both gate and pin resources. For our designs on the 100 pin FPGAs, hardwiring reduced the longest critical path from 6 to 3 for Sparcle and from 17 to 15 for the A-1000.

5 Conclusions and Future Research

This paper describes the software portion of a project at MIT to produce a scalable, low cost FPGA-based emulation system which maximizes FPGA resource utilization. While this paper has focused on software techniques for improving performance in FPGA-based logic emulation systems, it is also applicable to other types of FPGA-based systems.

Our results show that virtual wires allow maximum utilization of FPGA gate resources at emulation speeds competitive with existing hardwired techniques. This technique is independent of topology. It allows the use of less complex topologies, such as a torus instead of a crossbar, in cases where such a topology was not practical otherwise.

This project has uncovered several possible areas for future research. Using timing and/or locality sensitive partitioning with virtual wires has potential for reducing the required number of routing sub-cycles. Communication bandwidth can be further increased with *pipeline compaction*, a technique for overlapping the start and end of long virtual paths with shorter paths traveling in the same direction. A more robust implementation of virtual wires replaces the global barrier imposed by routing phases with a finer granularity of communication scheduling, possibly overlapping computation and communication as well.

Using the information gained from dependency analysis, we can now predict which portions of the design are active during which parts of the emulation clock cycle. If the FPGA device supports fast partial reconfiguration, this information can be used to implement *virtual logic* via invocation of hardware subroutines [8]. An even more ambitious direction which we are exploring is event-driven emulation - only transmit signals that change, only activate (configure) logic when it is needed.

		Hardwire Only	Virtual Wire Only
Sparcle	Longest Path	9 hops	6 hops
	Computation only delay	250 ns	370 ns
	Communication Only delay	180 ns	120 ns
	Emulation Clock Range	2.3–5.6 MHz	2.0–8.3 MHz
A-1000	Longest Path	27 hops	17 hops
	Computation only delay	250 ns	590 ns
	Communication Only delay	540 ns	340 ns
	Emulation Clock Range	1.3–4.0 MHz	1.1–2.9 MHz

Table 3: Emulation Clock Speed Comparison

References

[1] A. Agarwal et al. The MIT Alewife machine: A large-scale distributed memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Press, 1991.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *To appear in IEEE Micro*, June 1993.

[3] D. Bertsekas and R. Gallagher, editors. *Data Networks*. Prentice Hall, Englewood Cliffs, N.J., 1992.

[4] Concurrent Logic, Inc. *CLi6000 Series Field-Programmable Gate Arrays*, May 1992. Revision 1c.

[5] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), Mar. 1992.

[6] D. V. den Bout, J. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman. Anyboard: An FPGA-based, reconfigurable system. *IEEE Design and Test of Computers*, Sept. 1992.

[7] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1), Jan. 1991.

[8] N. Hastie and R. Cliff. The implementation of hardware subroutines on field programmable gate arrays. In *IEEE Custom Integrated Circuits Conference*, May 1990.

[9] InCA Inc. *Concept Silicon Reference Manual*, Nov. 1992. Version 1.1.

[10] S. Kirkpatrick, C. D. Gellatt, and M. P. Vecchi. Simulated annealing. *Science*, 220, 1983.

[11] J. Kubiatowicz. User's Manual for the A-1000 Communications and Memory Management Unit. ALEWIFE Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[12] H. T. Kung. Systolic communication. In *Proceedings of the International Conference on Systolic Arrays*, San Diego, California, May 1988.

[13] L. Maliniak. Multiplexing enhances hardware emulation. *Electronic Design*, Nov. 1992.

[14] S. Walters. Computer-aided prototyping for ASIC-Based systems. *IEEE Design and Test of Computers*, June 1992.

[15] Y.-C. Wei, C.-K. Cheng, and Z. Wurman. Multiple-level partitioning: An application the very large-scale hardware simulator. *IEEE Journal of Solid-State Circuits*, 26(5), May 1991.

[16] Xilinx, Inc. *The Programmable Gate Array Data Book*, Aug. 1992.

[17] Xilinx, Inc. *The XC4000 Data Book*, Aug. 1992.