Logic Emulation with Virtual Wires

Jonathan Babb, Russell Tessier, Matthew Dahl,
Silvina Hanono, David Hoki, and Anant Agarwal

MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

Logic emulation enables designers to functionally verify complex integrated circuits prior to chip fabrication. However, traditional FPGA-based logic emulators have poor inter-chip communication bandwidth, commonly limiting gate utilization to less than 20 percent. Global routing contention mandates the use of expensive crossbar and PC-board technology in a system of otherwise low-cost, commodity parts. Even with crossbar technology, 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). *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. The resulting increase in bandwidth allows effective use of low dimension, direct interconnect. The size of the FPGA array can be decreased as well, resulting in low cost logic emulation.

This paper covers major contributions of the MIT Virtual Wires project. In the context of a complete emulation system, we analyze phase-based static scheduling and routing algorithms, present Virtual Wires synthesis methodologies, and overview an operational prototype with 20K-gate boards. Results, including in-circuit emulation of a SPARC microprocessor, indicate that Virtual Wires eliminate the need for expensive crossbar technology while increasing FPGA utilization beyond 45 percent. Theoretical analysis predicts that Virtual Wires emulation scales with FPGA size and average routing distance, while traditional emulation does not.

1 Introduction

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 software simulators. This performance is achieved by partitioning a logic design, described by a *netlist*, across an interconnected array of FPGAs. The netlist *partition* on each FPGA, configured directly into logic circuitry, is then executed at near hardware speeds.

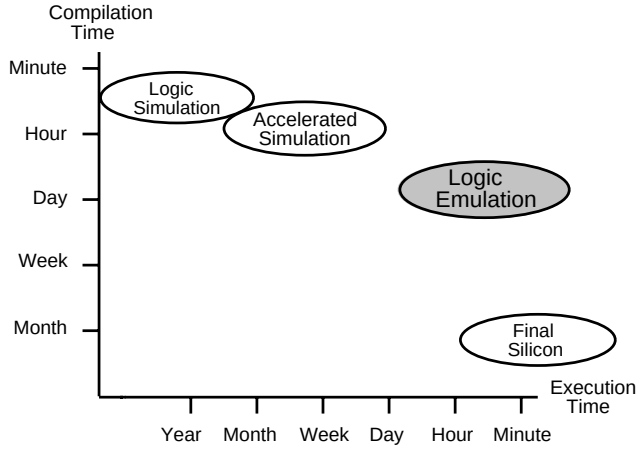

Figure 1: Verification Alternatives

Figure 1 compares logic emulation to other prototyping methods, including simulation and accelerated simulation, as well as to final silicon. The y-axis measures relative time for compiling or constructing a hypothetical design, while the x-axis measures relative time for executing one set of test vectors on this design. As an example, consider final silicon which takes months to construct and runs a set of vectors in less than one minute. The same design and vector set could be compiled for a logic simulator on the order of minutes, but would take years to execute. Logic emulation fills a wide gap between simulation and actual silicon. With both a moderately fast compile time and a fast execution time, emulation offers a compromise between the programmability of software and the fast execution speed of hardware.

Logic emulators are further characterized by interconnection topology, target FPGA, and supporting software. The interconnection topology describes the arrangement of FPGA devices and routing resources. Example interconnects include full crossbars and two-dimensional meshes. Impor-

Not Limited	Gate Limited
− unused FPGA pins − unused FPGA gates	− some unused pins − no unused gates
Pin Limited	**Balanced**
− no unused pins − some unused gates	− no unused pins − no unused gates

Figure 2: Partition Limitation Scenarios

tant target FPGA properties include gate count, pin count, and mapping efficiency. Supporting software is extensive, combining netlist translators, logic optimizers, technology mappers, global and FPGA-specific partitioners, placers, and routers.

Traditional emulators are gate inefficient due to inherent pin limitations in the FPGA devices. To reduce pin limitations, these emulators supplement FPGAs with custom crossbars chips and expensive PC-board and backplane technology, further increasing the per-gate cost of emulation. This paper suggests an alternative solution to pin limitations based on multiplexing of FPGA resources.

1.1 Virtual Wires

In existing emulator architectures, both the logic configuration and the network connectivity remain fixed for the duration of the emulation. Every emulated partition of the input design, one per FPGA, consists of a set of gates and a set of signals communicating to other partitions. Each emulated gate is mapped to one or more FPGA equivalent gates and each inter-partition emulated signal is allocated to a pair of pins between two FPGAs. Thus for a partition to be feasible, the partition gate and pin requirements must be no greater that the available FPGA resources. These constraints yield four possible scenarios (Figure 2).

When typical circuits are mapped onto available FPGA devices, partitions are predominately pin limited. That is, all available FPGA gates cannot be utilized due to lack of pin resources to support them. We demonstrate this resulting *bandwidth gap* with a set of partitionings of the Sparcle and CMMU benchmarks (see Section 5.1) for various gate counts. Figure 3 shows the resulting curves, plotted on a log-log scale. Partition gate count is scaled by a factor of two to get FPGA equivalent gates with an assumed mapping efficiency of 50%. On the same curve we plot the pin and gate capacity of target FPGAs: the Xilinx 3000 and 4000 series [40], the Altera Flex 8000 series [3], and the Atmel 6000 series [5]. For equal average gate counts in the benchmark partitions and FPGA devices, the required average pin counts for partitions are much greater than the available pin capacity of the FPGAs.

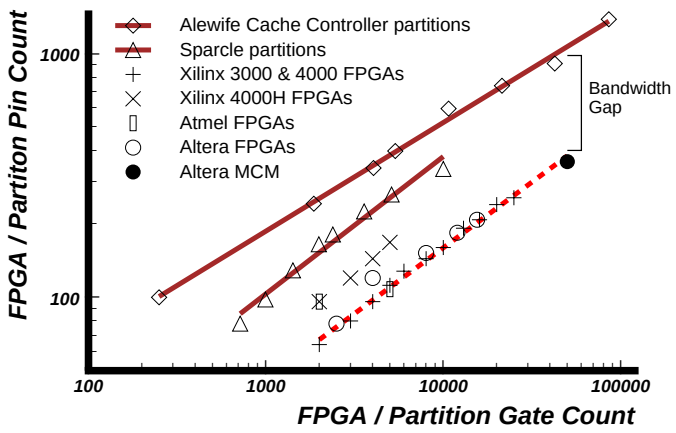

Figure 3: Pin Count as a Function of FPGA Partition Size

Pin limits set a hard upper bound on the maximum usable gate count any FPGA gate count can provide. Low utilization of gate resources increases both the number of FPGAs needed for emulation and the time required to emulate a particular design. This discrepancy will only get worse as technology scales; current trends indicate that available gate counts are increasing faster than available pin counts. Future breakthroughs in area I/O [27] may partially address this problem for FPGA packaging, but will leave open the more difficult issues of inter-board and system-level communication. Additionally, any new technology will be challenged to keep up as minimum feature size decreases faster than required bonding area.

Virtual Wires eliminate the pin limitation problem of previous emulators by intelligently multiplexing each physical wire among multiple logical wires and pipelining these connections at the maximum clocking frequency of the FPGA.[1] 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 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).

Without Virtual Wires, one to one allocation of logical wires to physical wires does not exploit available pin bandwidth because:

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

[1] Although this paper focuses on logic emulation, Virtual Wires can be applied to any multi-chip system.

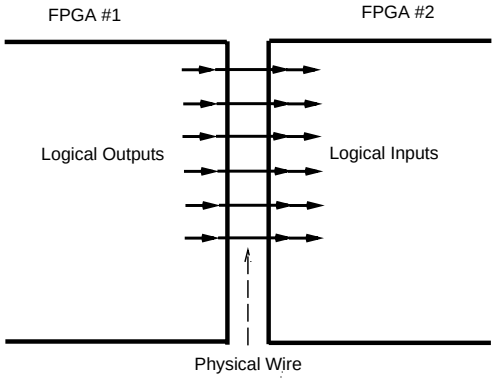

Figure 4: Hard Wire Interconnect

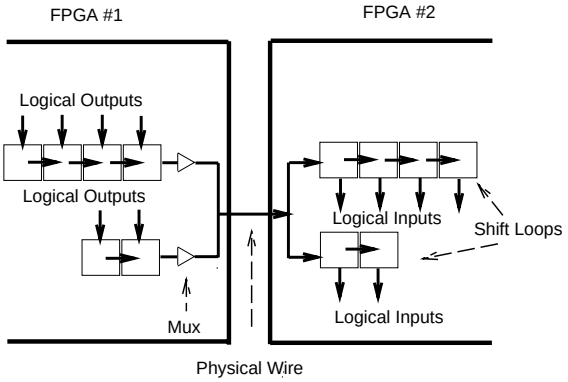

Figure 5: Virtual Wire Interconnect

However, by clocking physical wires at the maximum frequency of the FPGA technology, several logical connections can share the same physical resource. Figure 4 shows an example of six logical wires allocated to six physical wires. Figure 5 shows the same example with the six logical wires sharing a single physical wire. The physical wire is multiplexed between two pipelined *shift loops* (Section 3). Each register in the pipeline carries a single bit of information from one logical output to the corresponding logical input in the neighboring FPGA.

Systems based on Virtual Wires exploit several properties of digital circuits to boost bandwidth from available pins. In a logic design, evaluation flows from system inputs to system outputs. In a synchronous design with no combinational loops, this flow can be represented as a directed acyclic graph. Thus, through analysis of the underlying logic circuit, logical values between circuit partitions only need to be transmitted once. 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 pin utilization.

1.2 Emulation Software

Software for logic emulation with Virtual Wires roughly follows the standard emulation tool flow (Figure 6). The input, a netlist of the logic design to be emulated, is transformed into a multi-FPGA configuration bitstream to be downloaded onto the emulator. Not shown are the technology libraries, target FPGA characteristics, and FPGA interconnect topology needed to make the correct transformations. We next describe the standard steps.

Translator: The input netlist to be emulated is typically generated with a hardware description language or a schematic capture program. The netlist must be syntactically translated into a format readable by the emulation software. Commercial and public domain tools are available for generic source-to-source translation. At MIT we used both Verilog and LSI logic formats.

Tech Mapper: The translated netlist is still specified in terms of the source technology library – for example LSI Logic's LCA100K technology [26]. Before emulation, the netlist must be mapped to a target library of FPGA primitives. Although commercial and public domain tools are also available for mapping, our simple and fast technique is to create a mapping library which describes each source primitive in terms of primitives in the target library. The inefficiency of this mapping can be largely recovered with a following logic optimization pass.

Partitioner: After mapping the netlist to the target technology, the netlist is divided into partitions, each of which can fit into a single target FPGA. Without Virtual Wires, each partition must have both fewer gates and fewer pins than the target device. With Virtual Wires, the total gate count, including the overhead of Virtual Wires multiplexing logic, must be no greater than the target FPGA gate count. In the MIT implementation, we used the InCA Concept Silicon partitioner [19]. This partitioner performs K-way partitioning with min-cut and clustering techniques.

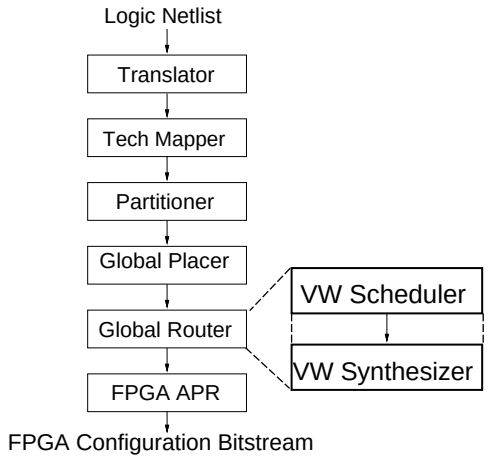

Figure 6: Emulation Software Flow

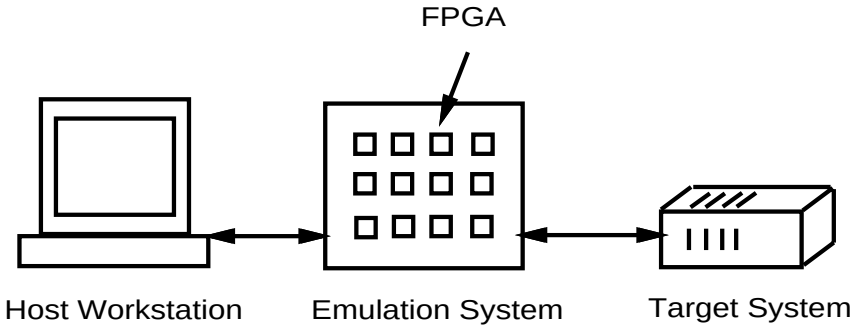

Figure 7: Virtual Wires Emulation System

Global Placer: Individual circuit partitions must be placed into specific FPGAs. An ideal placement minimizes system communication, thus requiring less routing resources. We wrote a simple placer based on simulated annealing [21] to minimize total Manhattan wire length.

Global Router: In traditional emulation, inter-FPGA communication is established with a global routing phase. If crossbars are employed, this phase must also determine the routing configuration for each crossbar as well as pin-assignments of partition I/Os to FPGA pins. For Virtual Wires emulation, there are no direct physical connections between partitions, and this phase is completely replaced with new virtualization software to be described in this paper.

FPGA APR: Once routing is complete, there is one netlist for each FPGA. Each netlist must be processed with FPGA specific automated place-and-route (APR) software to produce configuration bitstreams. We used the XACT [40] software for Xilinx FPGAs.

With Virtual Wires, we replace the global router of traditional software with modules created to specifically support automatic pin multiplexing: the Virtual Wires Scheduler and the Virtual Wires Synthesizer (Figure 6). Together we refer to the transformation performed by these two components as *virtualization*. Although each emulation step is an intriguing aspect of CAD research, this paper focuses on these novel virtualization components, described below.

Virtual Wires Scheduler: The resulting set of netlist partitions mapped to each FPGA, in conjunction with the routing resource constraints of the emulation system, is used to determine an appropriate schedule of logical wires onto physical wires. This schedule establishes a feasible time-space route for every logical wire, while guaranteeing that all multi-FPGA combinational paths are correctly ordered. Schedule optimizations include minimizing the total time needed to execute the circuit, as well as minimizing the Virtual Wires logic overhead. While this scheduling problem is similar to those encountered in high level synthesis, it is complicated by inter-FPGA routing constraints and the need to account for multiplexing overheads. In Section 2, we describe the phase-based scheduling algorithm implemented at MIT.

Virtual Wires Synthesizer: This step implements the chosen routing schedule by synthesizing special multiplexers and registers that are added to the circuit partition in each FPGA. This logic is effectively a pipelined, statically routed network in the FPGA technology itself. For maximum efficiency, the synthesizer takes into account the underlying idiosyncrasies of the target FPGA technology. For example, FPGA pin assignment and allocation of internal tri-state buses are carefully optimized. The resulting synthesized architectures provide insight into Virtual Wires implementation. Section 3 compares three different architectures for the Xilinx 4000 series.

1.3 Low Cost Emulation System

Although virtualization can be used to map input designs to any FPGA-based logic emulator, the process is most valuable when enabling the use of inexpensive, direct interconnect and cheap, low pin count FPGAs. To demonstrate this advantage, we have constructed FPGA boards composed of sixteen mesh-connected FPGAs and commodity SRAMs. These boards may themselves be mesh-connected, leading to straightforward software mapping and simplified system scalability. This system (Figure 7), described in Section 4 has demonstrated the following functionality:

- **In-circuit emulation:** FPGA array mimics one or more components of the target system and is pod-connected to the chip sockets of those missing components.
- **Simulation acceleration:** FPGA array replaces a piece of a simulation model and connects to the software simulation environment by remote calls through the host interface.
- **Hardware subroutines:** FPGA array implements a Verilog version of a subroutine in a C program and connects to the software by remote calls through the host interface [9].

Section 5 describes our results for both in-circuit emulation and simulation acceleration of the Sparcle benchmark on our system, including booting a multiprocessor operating system. We leave the exploration of hardware subroutines to future reports.

1.4 Scalable Technology

Not only can Virtual Wires be used to compose low-cost systems of gigantic numbers of FPGAs, but this technology also scales as FPGA sizes increase. To demonstrate this scalability, Section 6 uses Rent's Rule to derive theoretical models of emulation gate overheads for systems with and without Virtual Wires. This model accounts for the mismatch between circuit communication and FPGA communication in the hard-wired case and includes a topological factor that explains why a mesh topology does not scale without Virtual Wires. With this model we show how the derived Virtual Wire utilization scales with increasing FPGA device size and average routing distance, while hard-wired utilization may not.

1.5 Overview

The rest of this paper is organized as follows: Section 2 describes the Virtual Wires scheduling and routing algorithms. Section 3 then compares three Virtual Wires synthesis architectures. After Section 4 describes our demonstration hardware system, Section 5 then present results for both simulation acceleration and in-circuit emulation on this system. Section 6 analyzes the overhead and scalability of Virtual Wires versus hard-wires. Finally, Section 7 describes related work in the field and Section 8 makes concluding remarks.

2 Scheduling Algorithms

Virtualization replaces the inter-FPGA routing steps of traditional emulation with software that synthesizes a routing network into the netlist partition on each FPGA. This network establishes global routes via statically scheduled bits rather than hard-wired interconnections. The first phase of this approach is a scheduling and routing algorithm. Our phase-based methodology suffices to prove the concept of Virtual Wires scheduling and is within a factor of two of more optimal algorithms presented in recent literature [32]. Before describing the scheduling algorithms, let us first introduce the basic operating principles of Virtual Wires.

2.1 Phase-Based Operating Principles

The *emulation clock* period is the clock period of the logic design being emulated. To facilitate multiplexing we break

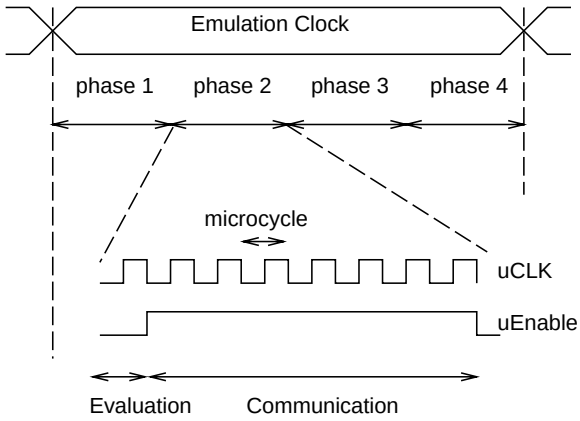

Figure 8: Clocking Framework

this period into a number of *microcycles* determined by a free-running μ*CLK* (Figure 8). In this scheme, a microcycle is the shortest distinguishable unit of time. All routing is scheduled in discrete microcycle increments. These microcycles are grouped into sequential *phases* to support combinational paths that extend across multiple chips. The advantage of this approach is a decoupling of logic execution speed from inter-chip communication speed, allowing high-speed communication cycles to co-exist with a long-latency emulation clock period.

A μ*Enable* signal divides each phase into an evaluation time span and a communication time span. Within a phase, a given number of microcycles are dedicated to the evaluation of the FPGA logic, followed by a set of cycles to communicate the results to other partitions in destination FPGAs. Evaluation takes place at the beginning of a phase, with logical inputs being propagated through each circuit partition to determine logical outputs for that phase. Not all inputs are available at the beginning of each phase, and not all outputs are produced. For inputs which are available, all logic is evaluated and subsequent outputs are produced. Each input and output transmission will be assigned to a single phase such that signal precedences are observed. At the end of the phase, the produced outputs are communicated to other circuit partitions at the microcycle clock rate. All necessary phases must be executed by the end of the emulation clock period.

For simplicity, we limited our approach to synchronous logic with a single global emulation clock. Any asynchronous signals cannot be statically-routed and therefore must be hard-wired to dedicated FPGA pins. Virtual Wires can be extended to multiple clocks [32] and gated clocks, as well as certain types of asynchronous logic, such as multiple asynchronous clock domains.

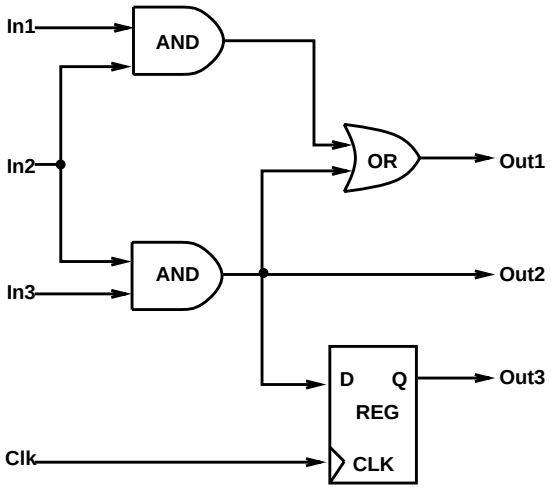

Figure 9: Dependence Calculation Example

2.2 Definition of Dependence and Depth

Two timing analysis computations, dependence and depth, aid in Virtual Wires scheduling. Both dependence and depth apply to inter-partition wires.

To analyze input to output *Dependence*, we scan the logic in each partition to determine the set of outputs to which a combinational path exists from each input. An output is said to be a dependent (or a child) of an input if a change in that input can combinationally change the output. The dependence relationships between inputs and outputs for a given partition are derived recursively from those of its constituent logic elements. In determining dependence, we assume that all outputs of a combinational library primitive are dependents of all the inputs of that primitive. Similarly, no outputs are dependents of any of the inputs for sequential primitives.

Let $Depend[i]$ denote the set of outputs of a given partition that are dependents of an input of the same partition connected to an inter-partition wire i. Similarly, let $D^{-1}[i]$ represent the set of inputs that are ancestors to an output driving an inter-partition wire i. By our definition, inputs to storage elements and external outputs will have no dependents: $Depend[i] = \emptyset$, and outputs of storage elements as well as external inputs will have no ancestors: $D^{-1}[i] = \emptyset$.

Figure 9 shows an example circuit partition containing four interconnected primitive logic elements with three inputs (not including the clock) and three outputs. The dependence relationships for this partition are as follows:

- $Depend[In1] = \{Out1\}$,
- $Depend[In2] = \{Out1, Out2\}$,
- $Depend[In3] = \{Out1, Out2\}$.

Likewise, the ancestors are as follows:

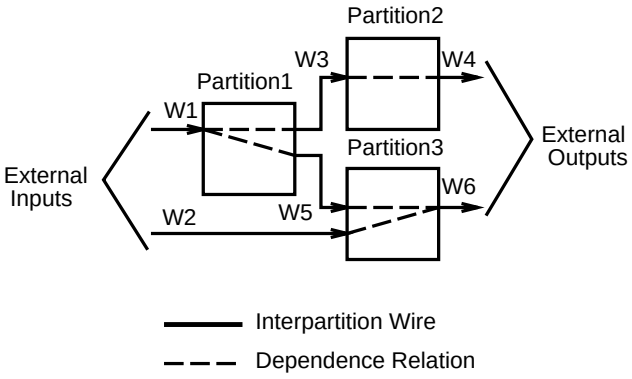

Figure 10: Depth Calculation Example

- $D^{-1}[Out1] = \{In1, In2, In3\}$,
- $D^{-1}[Out2] = \{In2, In3\}$,
- $D^{-1}[Out3] = \emptyset$ (REG is a storage element).

The *depth* calculations use the dependence relationships. The depth of inter-partition wire i is the largest number of partitions in a forward combinational path starting at that wire. Depth is computed recursively from the wire dependence sets such that for each wire i:

$$Depth[i] = \begin{cases} 0 & if Depend[i] = \emptyset \\ 1 + \max\limits_{j \in Depend[i]} Depth[j] & otherwise \end{cases} \quad (1)$$

Figure 10 shows an example design with three partitions and six inter-partition wires. The dashed-lines denote input-output dependence relationships. In this example, wires are at the following depths:

- depth 0: W4, W6
- depth 1: W2,W3,W5
- depth 2: W1

Our phase assignment algorithm uses depth to prioritize routing of critical paths. During scheduling, although both W1 and W2 have no ancestors, W1 has a greater depth and will be given priority.

2.3 Phase Assignment Algorithm

The goal of the phase assignment algorithm is to determine an appropriate schedule of logic wires between design partitions onto physical wires between FPGAs. The resulting schedule must establish a feasible time-space route for every logical wire, while observing FPGA routing resources constraints and guaranteeing that all multi-FPGA combinational paths are correctly ordered.

The core scheduling algorithm consists of a shortest path router inside a greedy phase assignment loop. Within the

```
Given:
    I : set of inter-partition design wires
    ComCycles : communication cycles per phase
    c : total micro-cycles per phase

Produce:
    S : output schedule which assigns each wire i ∈ I
        to a shiftgroup in a particular phase

Procedure PhaseAssign
    call Depend ← CalcDependents(I)
    call Depth ← CalcDepth(Depend)
    initialize Done array to false

    for each wire i ∈ I
        for each dependent wire j ∈ Depend[i]
            DependCount[j] ← DependCount[j] + 1
        endfor
    endfor

    n ← 0 /*phase counter*/
    loop forever
        call RouteInit
        W ← wires with Done[i]=false and DependCount[i]=0
        if W is empty exit loop
        n ← n + 1
        sort W by Depth[i], greatest depth first
        for each wire i ∈ W
            src ← FPGA partition where source of i is placed
            dest ← FPGA partition where dest of i is placed
            path ← Route(src,dest)
            if path exists then
                distance = length(path) - 1
                maxSignals ← ComCycles - distance
                L ← i, and up to maxSignals additional wires
                    from W that have the same src and dest as i
                for each j ∈ L
                    delete j from W
                    Done[j] ← true
                    for each k ∈ Depend[j]
                        DependCount[k] ← DependCount[k] - 1
                    endfor
                endfor
                shiftgroup ← { n, path, L }
                add shiftgroup to schedule S
            endif
        endfor
    endloop

    save n, c in schedule S /*final phase and cycle count*/
end Procedure
```

Figure 11: Phase Assignment Algorithm

main loop of the phase assignment algorithm, Figure 11, as many wires as possible are scheduled and routed. Once no more wires are available to schedule, the algorithm advances to the next phase. All unscheduled wires are thus pushed to the following phase when either their ancestors have not been scheduled, or there is no remaining routing path available in that phase. The phases are processed sequentially and no attempt is made to go back and optimize previously scheduled phases. Given enough phases and at least one potential path between all pairs of FPGAs, any design can be scheduled. This is easier than hard-wired routing problems, in which various rip-up and re-try strategies may be needed to find a feasible route.

The algorithm starts by first calculating the dependence and depth arrays for all wires as described in the previous section. An additional array, $Done[i]$, is set to false to mark each wire i as unscheduled. The algorithm then initializes a $DependCount[i]$ array from the dependence information of each wire. When this counter reaches zero, as the algorithm progresses, wire i will be ready to schedule. The algorithm proceeds by assigning wires to phases until all wires have been scheduled. Advancement to following phases occurs when no wires can be scheduled in the current phase. Within each phase, ready wires with the greatest depth are iterated first, guaranteeing that critical paths are given priority. The routing algorithm is successively called to route as many ready wires as possible.

Once a successful route is established from a source to destination FPGA, as many as $ComCycles - distance$ additional ready wires between the same source and destination are formed into a shiftgroup, where $ComCycles$ is the number of communication cycles in a phase and $distance$ is the number of FPGA crossings in the routing path. For example, if their are eight cycles per phase and the distance is three, a total of five wires can be routed in the same shiftgroup. The additional wires are also prioritized by depth. For each routed wire j in the shiftgroup, $Done[j]$ is set, and the set of wires $k \in Depend[j]$ are iterated to decrement $DependCount[k]$. If $DependCount[k] = 0$, wire k can be scheduled in a following phase. Any ready signals that are not successfully routed in a phase are automatically delayed to following phases. As long as the delayed signals are not on the critical path, the total number of phases will not be affected.

The $ComCycles$ parameter specifies the number of microcycles to spend in communication during each phase. This number must be greater than the routing diameter of the topology to guarantee that all signals can route. For the results in Section 5, it turns out that eight communication cycles match the eight-way tri-state bussing of the Xilinx architecture.

```
Given:
    T : emulator topology
    src : source FPGA in T
    dest : destination FPGA in T

Produce:
    path : list of FPGAs along shortest route from src to dest

Procedure RouteInit
    for each FPGA src ∈ T
        for each FPGA dest ∈ T
            Reserve[src,dest] ← connections in T from src to dest
            Avail[src,dest] ← (Reserve[src,dest] ≠ 0)
        endfor
    endfor
end Procedure

Procedure Route(src, dest)
    path ← ShortestPath(src,dest,Avail) /* Dijkstra's algorithm */
    if path exists then
        for each FPGA f ∈ path
            if f = src then /* first FPGA in path */
                src1 ← f
            else /* following FPGAs in path */
                dest1 ← f
                Reserve[src1,dest1] ← Reserve[src1,dest1]-1
                Avail[src1,dest1] ← (Reserve[src1,dest1] ≠ 0)
                src1 ← f
            endif
        endfor
        return path
    else
        return null path
    endif
end Procedure
```

Figure 12: Routing Algorithm

```
Shiftgroup {
    phase number
    source FPGA, 2nd FPGA, 3rd FPGA, ..., dest. FPGA
    logical wire 1, logical wire 2, ..., logical wire N
}
```

Figure 13: Shiftgroup Data Structure

2.4 Route Algorithm

The goal of the routing algorithm (Figure 12) is to find a shortest available path, in terms of FPGAs, between the source and destination FPGA of a set of inter-partition wires. The algorithm keeps track of the reserved and available physical connections between FPGAs in the emulator topology and is repeatedly called from the inner loop of the phase assignment algorithm. Route uses shortest path analysis with a cost function based on channel availability. Shortest path routing minimizes both the number of microcycles needed per phase and intermediate hop logic overhead.

Before the beginning of each phase, a reservation matrix, $Reserve[i, j]$, is initialized to the number of physical connections between FPGAs i and j in the emulator topology. Route applies Dijkstra's shortest path algorithm [34] to channel availability, $Avail[i, j] = (Reserve[i, j] \neq 0)$, to determine the shortest path between the source and destination FPGAs. If the shortest path exists, then the reservation matrix is updated by subtracting one from each element along that path and route returns with this path; else, route returns unsuccessfully.

After each successful route, PhaseAssign forms a new shiftgroup data structure (Figure 13). This data structure includes the phase number, FPGA path, and set of logical wires in that group. This information is written to the schedule file, to be passed to the synthesis phase of virtualization.

2.5 Execution Speed Analysis

Before proceeding, let us compute the execution speed of Virtual Wires emulation. Based on our phased operating principles, the emulation clock cycle time will be determined by the total number of microcycles needed:

$$v = n \times c, \tag{2}$$

where n is the number of phases and c the number of cycles per phase as previously defined. If c is the same across all phases, then we can immediately recognize that:

$$n \geq L,\ c \geq D, \tag{3}$$

where D is the maximum distance of any shiftgroup route (in the worst-case D is the network diameter of the FPGA topology), and L is the length of the critical path in the design netlist, equivalent to the maximum depth. That is, there must be enough cycles in each phase to route a signal across the diameter of the network as well as enough total phases to schedule the longest combinational path between circuit partitions.

Additionally, we recognize that the total number of microcycles is also constrained by the maximum multiplexing performed at each FPGA:

$$v \geq P_C/P_f, \tag{4}$$

where P_C is the maximum circuit communication requirement, including partition pins and any additional pins required for through hops, and P_f is the pin count of each FPGA[2]. Combining these two observations and assuming that the number of microcycles per phase is constant across all phases, we get the following best-case speed result:

Best-case microcycles: The cumulative microcycle count for all phases within a scheduled emulation clock period is bounded below by the following equation:

$$v \geq \max(\overbrace{L \times D}^{latency\ bound}, \overbrace{P_C/P_f}^{bandwidth\ bound}). \quad (5)$$

where L is critical path length, D is network diameter, P_C is the maximum circuit partition pin count including through hops, and P_f is the FPGA pin count.

In our practical experience, design emulation speed is determined predominately by the latency bound.

2.6 Improvements

We proposed the preceding algorithms to demonstrate the feasibility of Virtual Wires and for ease of implementation of the synthesis structures described in the following section. These algorithms can be improved by scheduling at the granularity of a single microcycle and eliminating the phase barriers altogether. The advantages of such improvements [32] include:

- possible initiation of computation and subsequent routing as early as one microcycle after a signal arrives at a destination rather than waiting for the following phase,
- potential overlapping of computation with communication in different parts of the system rather than execution in exclusive time spans,
- support of different propagation delays for individual wires rather than observing a worst-case delay for all computation in a phase,
- flexible scheduling of wires at the microcycle granularity rather than scheduling of dedicating pipeline paths per phase. This scheduling also eliminates costly pipeline filling overhead at the beginning and end of each phase.

We continue by describing the synthesis architectures designed at MIT. These architectures implement the virtualized routing network produced by the phase assignment and routing procedures.

[2]Note that we have ignored pipeline startup overhead associated with each shiftgroup.

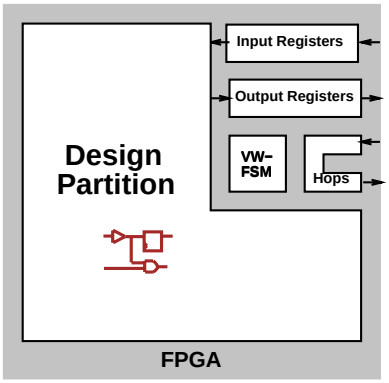

Figure 14: Virtualized FPGA Composition

3 Synthesis Architectures

Although it would be possible to design an FPGA with multiplexed pins, we implemented Virtual Wires without custom hardware support. That is, the virtualization process synthesizes the required multiplexing components directly into the FPGA netlists, to be downloaded with the original design partitions. Thus, any existing FPGA-based logic emulation system can take advantage of Virtual Wires. After discussing the synthesis algorithms, this section proceeds to describe three of many possible synthesis architectures based on shift registers in Xilinx FPGAs.

3.1 Synthesis Algorithm

The Virtual Wires Synthesizer flowchart component in Figure 6, takes the following input:

- emulator topology,
- external design I/O pin assignment,
- routing schedule,
- design partition netlists,

and produces:

- one virtualized netlist for each FPGA.

As shown in Figure 14, these new netlists contain the original design partition along with all necessary Virtual Wires communication logic. Synthesized logic includes input and output shift registers, through hops, and the VW-FSM finite state machine control logic.

After reading in the appropriate inputs from previous compilation stages, the Synthesizer algorithm (Figure 16) proceeds to synthesize the VW-FSM control logic for each FPGA. For the most part, this logic is identical for each

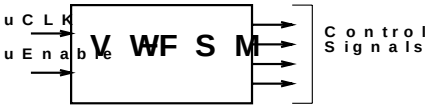

Figure 15: VW-FSM Control Logic

Given:
T : emulator topology
E : external I/O pin assignment
S : routing schedule
D : set of design partition netlists

Produce:
X : set of virtualized FPGA netlists

Procedure Synthesize
N ← number of phases in S
C ← cycles per phase in S
for each FPGA f ∈ T
given (N,C) synthesize control logic vwFsm[f]
virtualLogic[f] ← vwFsm[f]
endfor

for each phase P ∈ S
for each shiftgroup G ∈ P
R ← inter-FPGA routing path for G
L ← number of logical wires ∈ shiftgroup G
for each FPGA f ∈ R:
if f is first FPGA in R **then**
logic ← synthesize output shifter of length L
assign each logical output in G to logic
assign physical FPGA output pin to logic
else if f is an intermediate FPGA in R **then**
logic ← synthesize intermediate hop
assign physical FPGA I/O pins to logic
else /* if f is last FPGA in R */
logic ← synthesize input shifter of length L
assign each logical input in G to logic
assign physical FPGA input pin to logic
endif
assign control nets for vwFsm[f] to logic
virtualLogic[f] ← virtualLogic[f] + logic
endfor
endfor
endfor
for each external I/O e ∈ E
assign e to its specified FPGA physical pin
endfor

for each FPGA f ∈ T
partition[f] ← design partition in D placed on f
X[f] ← virtualLogic[f] + partition[f]
endfor
end Procedure

Figure 16: Synthesis Algorithm

FPGA, determined solely by the number of phases and microcycles per phases in the schedule. The VW-FSM (Figure 15) logic takes as input the μCLK and the μEnable signals, distributed to each FPGA, and generates the appropriate control signals during each microcycle. As described in Section 2, the μCLK is the free running pipeline clock, while the μEnable clock is synchronized to the emulation clock and determines when to start the communication sequence for each emulation phase. The output control signals are responsible for strobing logical wires into the appropriate registers and controlling multiplexer selection.

The algorithm iterates through the shiftgroups in each phase to construct the input, intermediate hop, and output architectures. Each shiftgroup data structure contains the logical wires assigned to that group as well as the group's phase and FPGA path. As the architectures are synthesized, they are connected between the partition logical wires and FPGA physical wires as well as to the appropriate control signals. Not shown in the algorithm, the Synthesizer also makes low level implementation decisions at this time to optimize the use of limited FPGA resources, including tristate busses and combinational logic blocks. In addition, we have added a simple pin permutation algorithm which minimizes the use of on-chip routing resources for hops.

The Synthesizer lastly assigns any external connections to corresponding periphery FPGA pins. Some of these pins connect to external interface hardware for communication with a logic simulator or other control programs. Additional pins provide global clocks and sequencing signals. The remaining pins may be connected to external pods to support in-circuit emulation.

The accumulated logic synthesized for each FPGA is then merged with the original design partition for that FPGA and a final virtualized netlist file is output in FPGA format (XNF for Xilinx). These files are input to the FPGA-specific place-and-route stage which creates the emulator bitstream.

3.2 Shift Register Architectures

We now compare three shift register architectures synthesizable to Xilinx 4000 FPGAs.

Full Shift Register

The full shift register architecture was originally proposed as a proof-of-concept Virtual Wires implementation [7]. This architecture consists of identical input and output *shift loops* (Figure 17). In output mode, shift loops load emulated signal states at the beginning of each phase and shift these states out serially onto a routed physical connection at the microcycle rate. For connections requiring multiple hops, a one-bit shift register is placed in each intermediate FPGA (Figure 18), forming a shift register pipeline between source and destination. At the end of the pipeline, corresponding

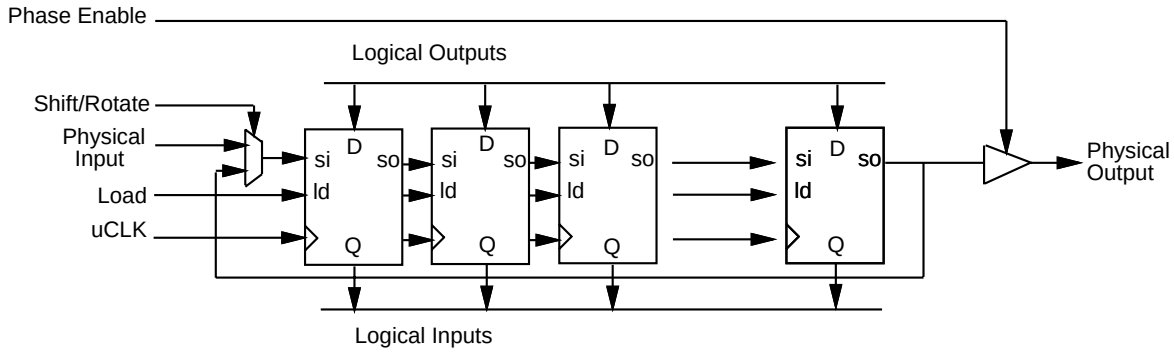

Figure 17: Full Shift Register Architecture

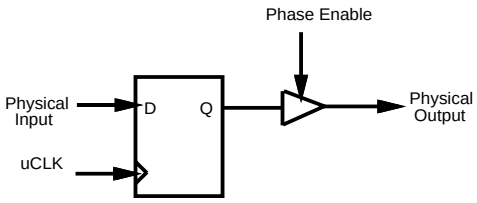

Figure 18: Intermediate Hop Architecture

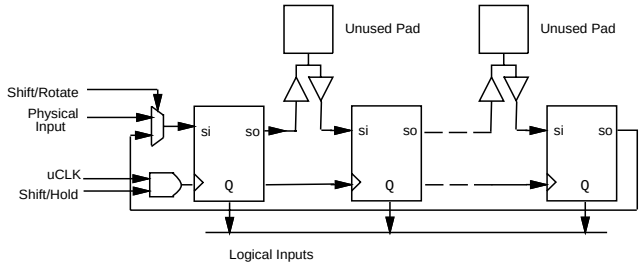

Figure 19: Gated Shift Registers Using Pad Registers

input mode shift loops de-multiplex and latch the emulated signals and drive them into the emulated logic. Note that the input shift loops must store their state so that all emulated logic inputs are available for subsequent evaluation. Output logic, however, can be reused for multiple groups of emulation signals in different phases. To support per-phase routing, each inter-FPGA I/O pad is preceded by a multiplexer that selects the appropriate shift loop output during its active phase. Pads are bidirectional with the pad driver enable signal asserted during phases in which that pad is an output. To minimize associated pad logic, the Synthesizer groups inputs and outputs separately when possible.

Gated Shift Register

To reduce the Virtual Wires consumption of core FPGA resources, the Synthesizer can utilize architecture-specific FPGA features. In low-cost, low-pincount FPGA parts, many of the I/O pads are not connected to pins, and the Synthesizer can concatenate their registers to form Virtual Wires shift registers (Figure 19). Due to pad configuration constraints, these shift registers cannot be parallel-loaded, so they are not usable for output shiftgroups. However, the Synthesizer can place input shiftgroups and intermediate hop shift registers here. Since input shiftgroups must hold the emulated signal state after receiving it, and these I/O registers do not have clock enables, the Synthesizer generates and distributes a gated μCLK. During the portions of the Virtual Wires cycle in which the emulated logic is being evaluated, this clock is frozen. In addition, the length of the input shiftgroups is adjusted to divide evenly into the number of μCLKs between evaluation periods so that the state in these registers can recirculate without change. In the 84-pin PLCC Xilinx 4005, this approach recovered 102 input and hop shift register bits. However, clock gating and the slower timing of the I/O pad registers reduced the achievable μCLK rate.

Addressable Shift Register

A further variation is to replace the output shiftgroup shift registers with tristate multiplexers available in the Xilinx architecture (Figure 20). The Synthesizer creates an additional set of global control signals, labeled cycle enables, to enable each bit of the multiplexer during the appropriate micro-clock tick of each Virtual Wires phase. The Synthesizer also replaces the input shift registers with sets of individual register bits whose clock enables are controlled by the phase signal as before, but whose clocks are successive cycle enables. This architecture considerably reduces the cost of the Virtual Wires shift loops in terms of logic resources, but the additional control signals and the use of the tri-state multiplexers add routing overhead. This overhead is reduced somewhat by placing many of the additional signals on global clock nets. Also, strategic use of the I/O pad registers for pipelining recovers speed. Finally, this architecture can support the more flexible Virtual Wires scheduling methods described in [32].

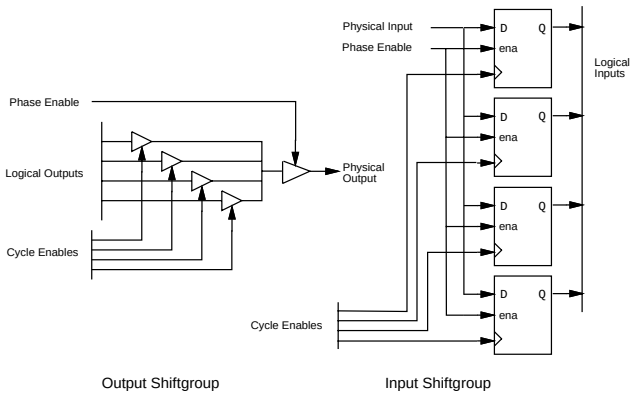

Figure 20: Addressable Shift Loop Architecture

Resource	*Design Logic*	*Total Logic (Virtual Overhead)*		
		Fig 17	Fig 19	Fig 20
Packed CLBs (Total)	54	79 (32%)	80 (33%)	65 (17%)
Lookup Tables (Combinational)	115	165 (30%)	167 (31%)	131 (12%)
Registers (Sequential)	74	141 (48%)	102 (27%)	115 (36%)
Xilinx PIPs (Routing)	1009	1729 (42%)	1948 (48%)	1612 (37%)
Average Resource Usage per FPGA				
μCLK Speed		33MHz	24MHz	25MHz
Emulation Speed		1.2MHz	.89MHz	.93MHz
Maximum Clock Speed				

Table 1: Architectural Comparison

Comparisons

Table 1 compares each architecture in implementing the smallest benchmark circuit, Palindrome (see Section 5.1), on the 16-FPGA demonstration hardware presented in Section 5. Speed is measured in terms of the μCLK speed. We calculated overhead as a percentage of consumed resources taken up by Virtual Wires. This Virtual Wire resource consumption is computed by subtracting the emulation logic resource consumption from total resource consumption. We measured emulation logic resource consumption by compiling un-virtualized partitions onto high pin count FPGAs. CLB refers to the basic Xilinx Combinational Logic Block, which includes both combinational lookup tables and sequential registers. We list the Programmable Interconnect Points (PIPs) as reported by the Xilinx router. Note that the reported numbers are for hardware emulation and do not include any additional speed and resource overheads that may be attributed to simulation acceleration.

The full shift register implementation is relatively fast but has significant overhead. The gated shift register architecture using the I/O pad registers is somewhat slower due to the reduced speed of these registers. This version does use fewer of the core registers, but routing overhead is higher because of the greater wiring distances covered between the pad registers and the core logic. Finally, the addressable scheme, has generally lower logic and routing overhead while maintaining moderate speed. The results in the remainder of this paper are based on the basic full shift register scheme, although we believe the addressable scheme to be the best of the three schemes because it can support more sophisticated scheduling algorithms as described in [32].

4 Demonstration Hardware System

Our demonstration hardware building block is a scalable emulation board [36] which is inexpensive to manufacture and easy to build (Figure 21). One or more boards are interfaced to a host workstation. Each board contains sixteen Xilinx XC4005 FPGAs [40] interconnected in a two-dimensional nearest-neighbor mesh. The board is six layers, uses only through-hole devices, and is ten inches square in size. System size is scaled by attaching additional boards on any of the four sides of the current system boards, without the need for crossbars or esoteric backplane technology. On board SRAM supports emulation of large design memories. At the present time we use a SparcStation 10 as the host interface although the emulation board may be reconfigured to interface to virtually any host computer. Any emulation board can communicate with this host workstation through either a serial or S-bus communication port. These interfaces are used to both observe in-circuit emulation status and to provide circuit inputs and outputs for simulation acceleration.

The following sections further detail the important features of the demonstration system.

4.1 FPGA Array

To emphasize the utility of Virtual Wires for inter-chip communication, we used no expensive crossbar chips and only low pin-count FPGAs (84-pin PLCCs). These FPGAs may be clocked at speeds approaching 40MHz thus resulting in small inter-chip delays and high emulation throughput. Figure 22 shows the board schematic. Each FPGA communicates with its four nearest neighbors (logically North, South, East, and West) through eight bidirectional I/O signals. To minimize multi-FPGA routing resources, these I/O signals are distributed along the chip package in an alternating pattern (Figure 23). Thus I/O port signals are physically allocated so that adjacent I/O pins are assigned to signals from differing ports. With this permutation a signal passing through the chip need only be routed the length of several pins rather than across the body of the entire chip. Two

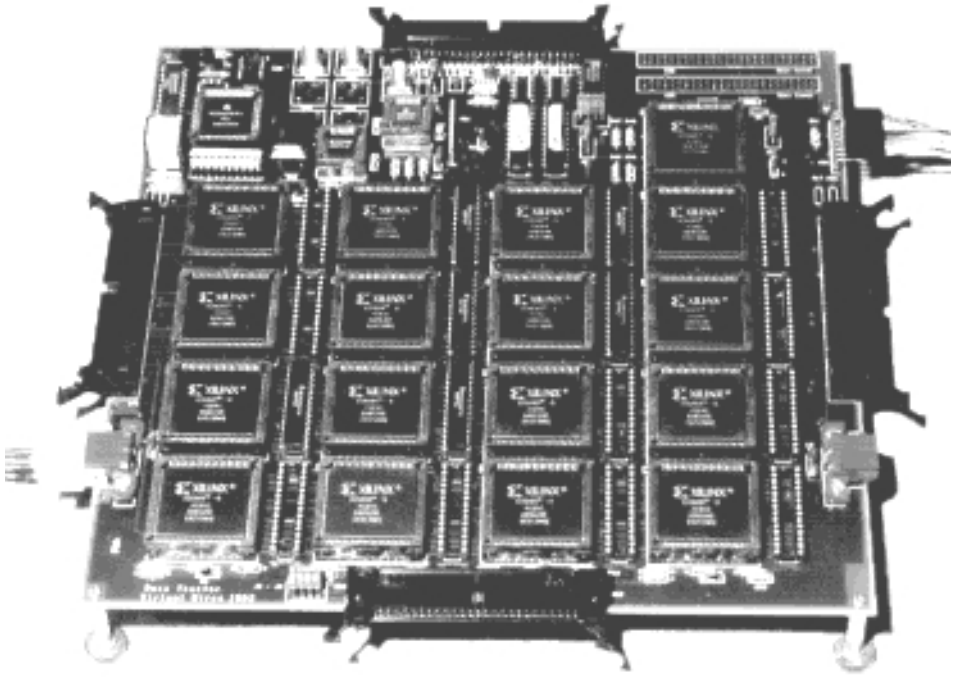

Figure 21: Virtual Wires Emulation Board

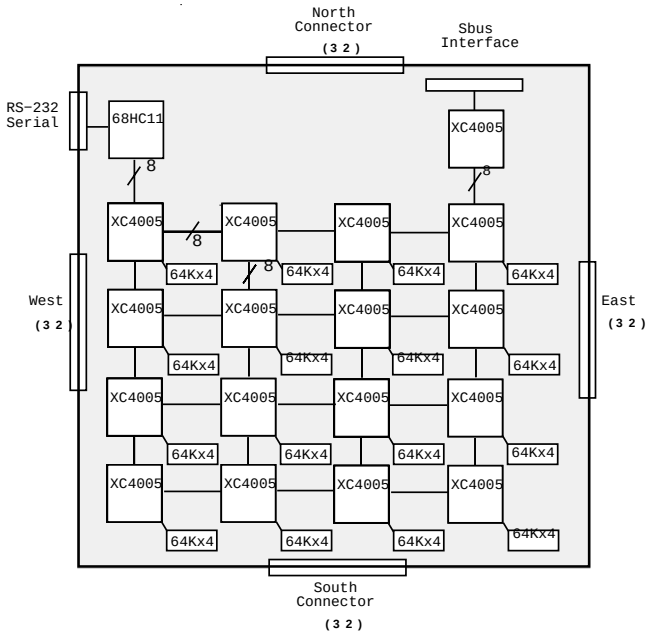

Figure 22: Emulation Board Schematic

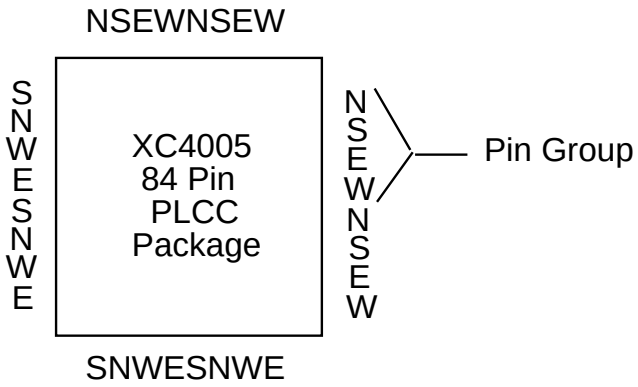

Figure 23: FPGA Pin Permutation

pin groups are located along each side of the package. This connection scheme is analyzed in detail in [18].

4.2 Scaling to Multiple-Board Systems

Each Virtual Wires prototype board can function as either a stand-alone system or as part of a larger group of boards. Multiple-board systems are constructed by connecting individual boards together to form a two-dimensional mesh (Figure 24). A clock driver chip and remote leads for clock cables allow one board to serve as a single global source for the other boards. μCLK signals are fanned out to lo-

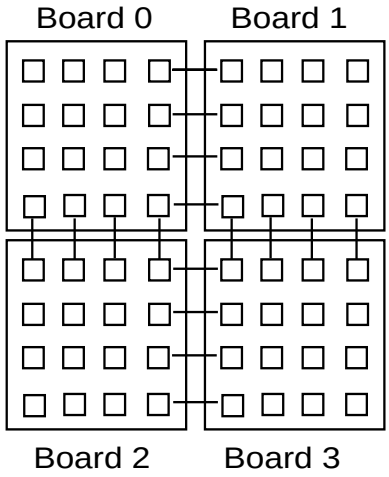

Figure 24: Multi-Board Emulation System

cal logic at the destination boards using a clock distribution chip. Bi-directional FPGA I/O signals along the periphery of each board extend across connectors in each of the four directions. FPGA configuration information is transferred in a serial chain to all FPGAs in the system starting at the board connected to the download cable in the upper rightmost corner of the mesh. System size is currently constrained to a total of ten boards (160 chips) by the Xilinx-imposed limit on the configuration bitstream, although this limit may be overcome with multiple download cables.

4.3 Memory and Host Interfaces

Each FPGA in the mesh has twenty-two dedicated I/O lines which interface to a 64K$\times$4 SRAM chip. These chips can be used to emulate sections of on-chip memory and are populated as needed. Virtual Wires software is used to multiplex address, data, and control signals for the SRAM so that a number of individual memory accesses to the same SRAM chip may take place during each emulation clock cycle. Twenty nanosecond SRAMs are used in the current prototype. SRAM interface signals have been allocated to dedicated FPGA pins to reduce capacitive loading on inter-FPGA signal lines and to simplify system software.

A low bandwidth serial interface via an embedded microcontroller provides access to the array for data transfer, configuration, and FPGA state readback. Data signals from the controller interface directly to the North port of the FPGA in the upper left corner of the array. The embedded controller has the capability to download configuration information to the array thus eliminating the need for an additional download cable from Xilinx.

To provide a higher bandwidth interface to the host, a seventeenth Xilinx XC4005 chip serves as an intermediary between an Sbus interface card located in the host SparcStation and the Xilinx array. This chip is capable of transferring words of data between the Sbus card [16] and the eight bit North port of the Xilinx chip in the upper right position of the array.

4.4 Application

The prototype system allows the logical behavior of one circuit component to be *emulated* while the rest of the system is *simulated*. It contains a simulation interface to both the LSI Logic LSIM and Cadence Verilog simulators. At a given simulated clock edge, software drivers transfer data representing inputs to the host workstation which subsequently forwards this data to the emulation system via the serial or S-bus host interface. Once output results are generated, the drivers return them for display or further simulation.

As an alternative to simulation acceleration, a target system may be interfaced directly to the emulator with a prototyping pod. This pod plugs into the chip socket in the target system. After FPGA configuration, the emulation system exchanges data with the target system at each emulation clock while performing internal evaluation at FPGA device speeds.

In both modes of operation, simulation and emulation, the usability of the system is enhanced by our InnerView Hardware Debugger [17]. This tool consists of host software, embedded controller software, and FPGA circuitry which extracts the emulation state from all FPGAs and coordinates this state with the internal register names of the design under emulation. This tool takes advantage of the serial interface's capability to perform readback from FPGAs on a chip by chip basis. The controller can be programmed to trigger a readback bitstream from any FPGA in the system, and subsequently transfer the values back to the host workstation for evaluation.

5 Results

We have successfully compiled designs up to 18K gates onto the demonstration system. In conjunction with the scheduling and synthesis algorithms described in this paper, we used the Synopsys Design Compiler [33] for translation and mapping, the InCA Concept Silicon partitioner [19] for partitioning, our simulated-annealing-based placer, and the standard Xilinx-provided tools for FPGA-specific place and route. Compile time is roughly 3 to 4 hours on a SparcStation 10, with 90 percent of this time consumed by the vendor specific FPGA compile. This compilation can thus be accelerated by processing independent FPGA compiles in parallel. The following sections describe our benchmarks and report simulation and emulation results.

Statistic	Palindrome	Sparcle	CMMU
LSI Gate Count	14,241	17,252	85,721
Element Count	4,623	4,802	37,871
Element Complexity	3.1	3.6	2.3
Memory Bit Count	0	4,352	na
Net Count	4,626	5,094	na

Table 2: Statistics for Benchmarks Designs

Results	Palindrome Simulation	Sparcle Simulation	Sparcle Emulation
FPGAs	16	20	24
Avg. partition Gates	890	868	714
Avg. partition I/O	45	126	119
Max. partition I/O	45	437	206
Emulation Speed (MHz)	1.00	0.12	0.18

Table 3: Simulation Acceleration and Emulation Results

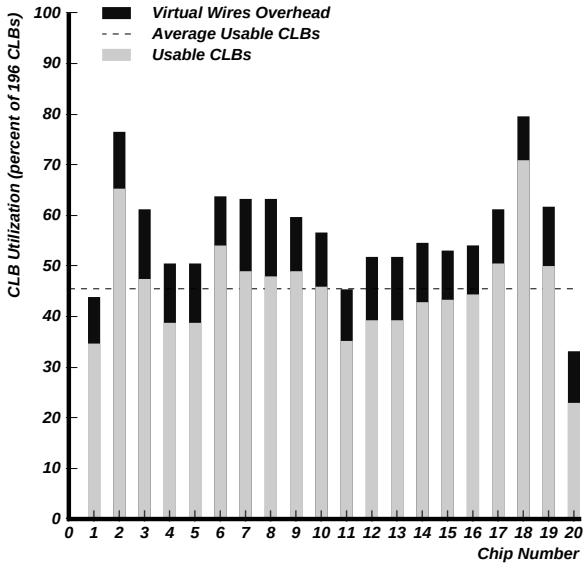

Figure 25: FPGA Resource Allocation for Sparcle

5.1 Emulation Benchmarks

Let us introduce relevant features of three benchmark designs for this paper (Table 2). The first design, Palindrome, is a simple 15K gate systolic array used for debugging the system and calibrating the various Virtual Wires architectures. The remaining two designs are actual chips from the MIT Alewife Multiprocessor. Sparcle [2], is an 18K gate SPARC processor with some modifications to enhance its usefulness in a multiprocessor. The Cache Controller and Memory Management Unit (CMMU) [22] is a complex 86K gate controller. For each design, our statistics include the LSI Logic LCA100K [26] gate count, the number of logic elements, the element complexity (gate count / element count), the number of on-chip memory bits, and the total number of nets connecting elements. Note that for CMMU measurements the memory elements are not included.

5.2 Simulation Acceleration

We have collected results from the successful simulation acceleration of Sparcle and Palindrome (Table 3). For simulation acceleration, speed refers to the evaluation rate of the emulation hardware rather than the actual simulation rate. The latter rate is currently limited by the speed of the simulator interface (2.1KHz) or the bandwidth across the host interface (41KHz for S-Bus or 30Hz for serial port). Figure 25 shows the allocation of resources inside each FPGA for Sparcle. While there is a fixed overhead of roughly 12 percent of the CLBs for Virtual Wires, usable CLBs exceed 45 percent. Note that due to internal FPGA routing, total utilization approaching 100 percent is not achievable.

5.3 In-Circuit Emulation

We used the emulation system in place of a Sparcle chip in a testbed board developed for the Alewife multiprocessor project. The emulator plugs directly into the Sparcle chip PGA socket using a commercially-built interface pod attached to the emulator board edge connectors. The emulator synchronizes automatically to the Sparcle system clock and control signals. Therefore, no modifications to the target system are needed other than slowing the system clock. The last column in Table 3 shows in-circuit emulation results for Sparcle. Four additional FPGA chips are needed to support the pod interface for simulation acceleration. The emulated Sparcle has executed system test programs successfully, including booting the Alewife operating system at 180KHz.

5.4 Comparison with Traditional Emulation

Table 4 contrasts the required FPGA pin counts for Sparcle emulation on a hard-wired crossbar and mesh configurations with the actual Virtual Wires board pin counts. Note that the Virtual Wires pin count is not a fixed constraint like the hard-wired pin counts. By increasing the total number of microcyles, we can lower the Virtual Wires pin count to as low as two pins per FPGA. Hard-wired mesh pins were estimated by multiplying the required crossbar pins by the average route length. This estimate is actually an underestimate because some wires connect to multiple FPGAs. Shown beside the hard-wired I/Os is also the *Pin Multiplication Factor* for each case. The PMF is simply the increase in pins needed if Virtual Wires are not employed. The table also compares emulation speed with estimated speeds for the hard-wired case. Virtual Wires speed is computed by

Results	Sparcle Simulation	Sparcle Emulation
FPGAs	20	24
Critical Path Length, L	10 partitions	11 partitions
Average Route Length, d	2.44 FPGAs	2.34 FPGAs
Maximum Route Length, D	7 FPGAs	6 FPGAs
Avg. VW-mesh I/O	25	29
Avg. HW-cross I/O (PMF)	126 (5.0)	119 (4.1)
Est. HW-mesh I/O (PMF)	294 (11.8)	293 (10.1)
VW-mesh μCLK Speed	12.0 MHz	18.0 MHz
VW-mesh phases $\times$ cycles	10×10	11×9
VW-cross phases $\times$ cycles	10×2	11×2
VW-mesh Emulation Speed	0.12 MHz	0.18 MHz
Est. VW-cross Emulation Speed	0.60 MHz	0.82 MHz
Est. HW-mesh Speed (ideal)	0.94 MHz	1.01 MHz
Est. HW-cross Speed (ideal)	1.30 MHz	1.43 MHz

Table 4: Comparison with Traditional Emulation

multiplying the number of phase and cycles-per-phase by the μCLK rate. Because our system does not have crossbars or FPGAs of the required pin count, we estimate potential hard-wired speeds. The estimate is based on the critical path length, L, and average route distance d, in our circuit partitions. It also assumes 20ns delays for FPGA-to-FPGA hops and FPGA-to-crossbar-to-FPGA delay, and 50ns delay for internal FPGA logic partition paths. Thus crossbar speed is $L \times 20ns + L \times 50ns$ while mesh speed is $L \times d \times 20ns + L \times 50ns$. Note that to achieve these speeds, FPGAs with the required pin counts must be used to maintain the same critical path and route lengths.

It is beyond our partitioning capability to map Sparcle onto 32-pin mesh connected FPGAs without Virtual Wires. However, in our earlier work [7] we did partition a version of Sparcle without memory or external I/Os onto 100-pin, 5000-gate FPGAs. We needed at least 31 FPGAs if they were fully connected, and greater than 100 FPGAs if they were connected in a torus. The FPGA explosion is correspondingly worse for the high communication A-1000 benchmark.

As a final comparison, note that reported results for application of the TIERS Virtual Wires routing algorithm [32] to Sparcle claim microcycle counts as low as 40 for a mesh, and 16 for other direct-connected topologies. These results support potential Sparcle emulation in excess of 1 MHz.

6 Analysis

In this section we derive theoretical gate utilization for logic emulation with and without Virtual Wires and show that emulation with Virtual Wires scales with increasing FPGA device size.

Parameter	IBM	FPGAs	4000H	Sparcle	CMMU
K	2.5	0.84	0.93	1.3	8.4
B	0.6	0.57	0.60	0.62	0.45

Table 5: Rent's Rule Parameters

6.1 Rent's Rule

We begin by reviewing an empirical observation made in 1960 by E.F. Rent of IBM. Rent prepared two internal memoranda containing the log plots of pins versus gates for portions of the IBM series 1400 computers [23]. The basic result is the following equation:

$$\textbf{Rent's Rule}: \quad P = KG^B \tag{6}$$

where P is the number of pins, G is the number of gates, K is a Rent's constant, and B is Rent's exponent. As with most rules, it has limitations. Rent's Rule can be used to measure the communication parameters of a given implementation technology as well as the parameters of a circuit. For a circuit, both its architecture and organization greatly affect the parameters. For example, pipelining a processor increases communication requirements due to dependencies between pipeline stages. Table 5 shows the original reported IBM constants as well as those we have measured for the FPGA technology and the Sparcle and CMMU benchmarks. Note that the 4000H FPGAs fall on a different curve due to their higher pin to gate ratio. For the other FPGAs, a B of 0.5 roughly corresponds to the area versus perimeter for the FPGA die. The lower B, the more locality there is within the circuit. Thus, the CMMU has more locality than Sparcle, although it has more total communication requirement, K.

6.2 Hard-Wires Gate Utilization

For circuits that obey Rent's Rule, we can determine the gate utilization for hard-wires, under pin-limited conditions (Figure 26). Given pin limitations, the number of FPGA pins, P_f, dictates the number of circuit partition pins, P_c, available for the circuit:[3]

$$P_c = \frac{1}{d} P_f, \tag{7}$$

where d is the average distance, in terms of FPGA boundary crossings, for each wire. This factor accounts for pins consumed by intermediate hop routing[4]. We next substitute Rent's equation for both sides of Equation 7:

$$K_c G_c^{B_c} = \frac{1}{d} K_f G_f^{B_f} \tag{8}$$

[3]For this analysis, we work with average partition pin and gate requirements. Pin limitation effects are worse when the circuit is non-uniform.

[4]For simplicity we are ignoring the effect of global nets with multiple destinations.

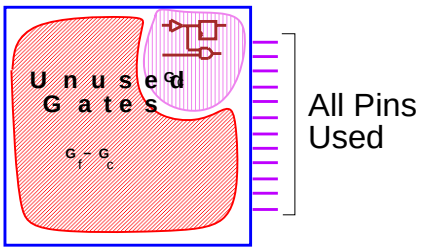

Figure 26: Gate Utilization without Virtual Wires

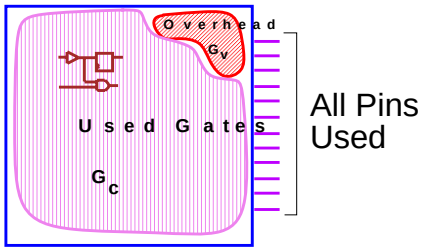

Figure 27: Gate Utilization with Virtual Wires

$$d_h = d^{1/B_c}$$
$$K_h = \left(\frac{K_f}{K_c}\right)^{1/B_c}$$
$$B_h = \frac{B_f}{B_c} - 1$$

Table 6: Hard-Wired Utilization Parameters

Solving for G_c yields the predicted number of mapped gates available to each circuit partition:

$$G_c = \left(\frac{1}{d}\frac{K_f}{K_c}{G_f}^{B_f}\right)^{1/B_c} \quad (9)$$

In this analysis, *mapped gates* refers to the gate count in the circuit's native technology, not the much higher count, FPGA-equivalent gates, claimed by FPGA vendors.

We next define new parameters, d_h, K_h, and B_h, Table 6, to simplify our work. Substitution of these newly defined parameters into Equation 9 and dividing by G_f yields the average per-FPGA utilization with hard-wires:

$$U_{hw} = \frac{1}{d_h}K_h {G_f}^{B_h}. \quad (10)$$

Note that if we combine the $1/d_h$ and K_h terms, this equation is very similar to the original Rent equation. Here we leave these terms separate to provide insight into the factors which affect overall hard-wired utilization. Each parameter is significant as follows: B_h shows how utilization will scale with FPGA device size. If B_h is negative, utilization will decline with increasing size, with a slope of B_h on a log scale. On the other hand, utilization is directly proportional to K_h for a fixed device size, and inversely proportional to d_h. For a crossbar interconnect with $d_h = 1$, K_h determines the offset of the utilization curve. For non-ideal interconnects and with B_c in the range of 0.5, the topological factor of d_h translates to a roughly quadratic decrease in hard-wired utilization as average routing path length increases.

6.3 Virtual Wires Gate Utilization

Let us model per-FPGA Virtual Wires costs as $V_0 + V_1 \times dP_c$, where V_0 is the per FPGA cost associated with the control circuitry and V_1 is the cost associated with each logic I/O. The total number of circuit pins is P_c and d is the same distance factor used here to amortize the cost of intermediate hops for each Virtual Wire into the overall cost equation.

For mapping circuits which obey the Rent equation, we can substitute $P_c = K_c G_c^{B_c}$ to get the average Virtual Wires cost:

$$G_v = V_0 + V_1 d K_c {G_c}^{B_c}. \quad (11)$$

Furthermore, $d = 1$ for a crossbar, and a derived upper limit to the average wire length for a mesh as a function of the total number of FPGAs, N, as reported by Bakoglu [8], is:

$$d = \frac{2}{9}\left(7\frac{N^{B_c-0.5}-1}{4^{B_c-0.5}-1} - \frac{1-N^{B_c-1.5}}{1-4^{B_c-1.5}}\right)\frac{1-4^{B_c-1}}{1-N^{B_c-1}}, \quad (12)$$

Thus given Rent's parameters for a given design, the average Virtual Wires overhead can be expressed strictly in terms of circuit and FPGA gate counts. We can then relate the FPGA gate count to the circuit gate count and virtual wires overhead as:

$$G_f = G_c + G_v. \quad (13)$$

Substituting Equation 11 and rewriting yields:

$$V_1 d K_c {G_c}^{B_c} + G_c - (G_f - V_0) = 0. \quad (14)$$

Solving Equation 14 for G_c yields the optimal partition size for a particular FPGA device size and circuit Rent parameters. We can further rewrite equation 14 in terms of utilization $U_{vw} = G_c/G_f$ to get:

$$V_1 d K_c {G_f}^{B_c} {U_{vw}}^{B_c} + (U_{vw} - 1)G_f + V_0 = 0. \quad (15)$$

Interestingly, Equation 15, and the mathematical solution for U_{vw}, are not a function of P_f, the FPGA pin counts, or the FPGA Rent parameters. While lower pin counts may increase the pin multiplexing factor, and therefore emulation latency, utilization will not be affected. Finally, from Equation 15 it is apparent that if $B_c < 1$, then as $G_f \to \infty$ Virtual Wire utilization will approach unity, $U_{vw} \to 1$, independent of other parameters. However, for small G_f the gate utilization will be low due to the dominating V_0 factor. The following section compares these results to the hard-wired case for realistic design and FPGA parameters.

6.4 Scalability with FPGA size

Using the results from the previous sections, we first compare achievable FPGA device utilization for both Virtual Wires

and hard-wires as FPGA sizes increases. Figure 28 shows utilizations for a hypothetical design, characterized by Rent parameters, on both a 4×4 mesh and a 16-chip crossbar topology. These graphs are on a semi-log scale, with the Y-axis measuring percent of usable gates and the X-axis logarithmically measuring FPGA device size. Table 7 shows the assumed parameters for the design, the technology, and the Virtual Wires overhead. These assumptions roughly fall into the vicinity of our experimental measurements. Note that since the FPGA device size is increasing, with a constant number of devices, the circuit size is increasing as well. Also, the FPGA pin count is increasing with the gate count in accordance with the FPGAs Rent parameters.

The slope of both hard-wire curves is $B_h = B_f/B_c - 1$. In Figure 28, B_h is negative and thus the hardware curves slope downwards. With a positive B_h, this curve would slope upwards instead. As a general rule, this equation suggests that FPGA vendors should attempt to track $B_f = B_c$ for hard-wired FPGA systems. The log-intercept is similarly K_h as earlier defined for the hard-wires crossbar. For the hard-wired mesh, the offset is lower than the crossbar by $1/d_h$ in accordance with Equation 10. For this example, the FPGA pin counts decrease with decreasing gate count such that FPGAs can never be fully utilized in the hard-wired case.

For Virtual Wires, the utilization is low for small gate count, partly due to the constant factor of V_0 control logic overhead. However, with increasing circuit size, this overhead is rapidly diluted: in both figures the Virtual Wires gate utilization approachs 100% with increasing FPGA size. For a mesh topology, the utilization increases more slowly, however it too asymptotically approaches unity. As a caveat to this comparison, note that for most FPGA devices, utilization will saturate at less than 100% due to internal routing restrictions.

6.5 Scalability with Routing Distance

As a final comparison, we consider the utilization effects of increasing the number of FPGAs, and therefore the average routing distance. To isolate topological effects, we show utilization as a function of the average routing distance (Figure 29) for FPGAs of size 1K, 10K, and 100K mapped gates. As before, circuit size is assumed to increase as the FPGA array size increases. The same parameters as in Table 7 are used with the exception of d, which is now a variable.

For the hard-wires curves, the utilization for $d = 1$ and $d = 2$ match the same data points as in Figure 28. As d increases, the utilization drops exponentially at the rate $d_h = d^{1/0.6}$ for all hard-wired cases. Additionally, note that the higher utilization curves correspond to lower gate count FPGAs, G_f, since B_h is negative. Here we see that crossbars, with $d = 1$, are essential for emulation without Virtual Wires. After only a few average FPGA hops of through routing, hard-wired utilization is nearly zero.

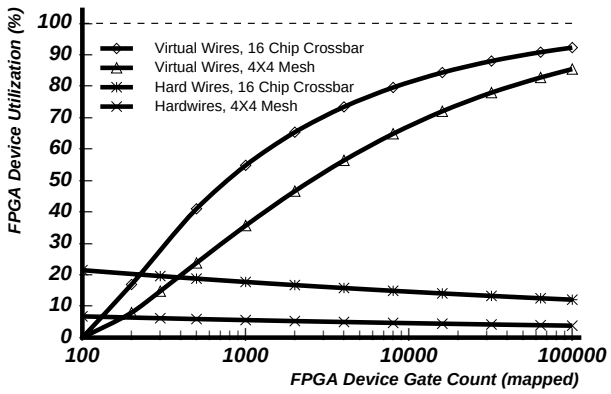

Figure 28: Scalability with FPGA Device Size

Parameter	Value
B_c	0.60
B_f	0.55
K_c	2.0
K_f	1.0
$d_{crossbar}$	1 hop
d_{mesh}	2 hops
V_0	100 mapped gates
V_1	4 mapped gates

Table 7: Parameters for Scalability Comparison

For the Virtual Wires curves, the higher gate count FPGAs have a higher utilization, as demonstrated in Figure 28. All curves asymptotically approach zero utilization as distance increases. However, for the larger gate count curves, a respectable utilization is maintained even for large values of d. For example, the $G_f = 100K$ curve has a utilization of 33 percent when $d = 16$. Thus Virtual Wires enable emulation to scale to a gigantic number of FPGAs using simple direct-connected topologies.

7 Related Work

IBM's *Yorktown Simulation Engine* [31] and the earlier *Logic Simulation Engine* [12], based on concepts of John Cocke, used reconfiguring digital hardware to accelerate logic simulation. Actual logic emulation was first explored in cellular array research, such as Frank Manning's 1975 thesis[29], even before FPGAs existed. His work explicitly shows how an "embedded machine" in programmable logic cells could be used in place of an actual machine. Since this work, FPGA-based logic emulation systems have been developed for design complexity ranging from several thousand to several million gates. Quickturn Design Systems, the pioneer

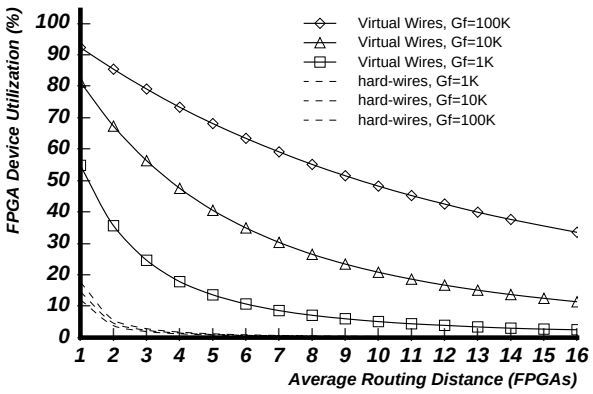

Figure 29: Scalability with Average Routing Distance

of large FPGA-based emulators, first developed emulation systems that interconnect FPGAs in a two-dimensional mesh and later in a partial crossbar topology [38] . Their largest systems use a hierarchical approach to interconnection [37]. Thorough reviews of contemporary emulation systems are provided by Hauck [18] and Owen [30].

Multiplexing to overcome pin limitations was first proposed by Babb [7] [6] in 1993 and the first successful applications discussed by Tessier [36], Dahl [14] [13] and Hanono [17]. Virtual Wires technology has continued to evolve at Virtual Machine Works, Inc. [32] [1], where commercial emulators based on proprietary VirtualWires(TM) technology are now being produced.

Since the original application of multiplexing to FPGA systems, others have proposed several similar approaches. In [24], multiplexing is extended to field-programmable interconnect devices called dynamic FPIDs. In [25], the resources inside the FPGA are multiplexed to reduce internal routing requirements. Recent time-multiplexed FPGA-like architectures include VEGA [20], Pegasus [28], DPGA [35], and Dharma [11]. Other related uses of static routing techniques include FPGA-based systolic arrays, such as Splash2 [4], and the very large simulation subsystem (VLSS) [39], a massively parallel simulation engine which uses time-division multiplexing to stagger logic evaluation. Finally, Virtual Wires are similar to virtual channels [15], which decouple resource allocation in dynamically-routed networks, and to virtual circuits [10] found in a connection-oriented network.

8 Conclusions

We have illustrated the benefits of logic emulation with Virtual Wires as a verification alternative. Previous pin limitations encountered when mapping designs onto multi-FPGA systems are now overcome. We have described correct-by-construction virtualization software, including both phase-based scheduling and synthesis algorithms, which can automatically re-time a synchronous input design to fit an arbitrary number and arrangement of FPGAs. We have demonstrated the success of Virtual Wires emulation on a low-cost system without crossbars, esoteric backplanes, or large pin-count FPGA devices. Finally, we have analyzed the overheads associated with Virtual Wires and found that FPGA utilization, over 45% in our emulation experiments, will asymptotically approach 100% in larger FPGAs, if not limited by internal FPGA resource constraints.

Although this paper has focused on logic emulation, virtualization is a generic tool that may be applied to other multi-FPGA systems, enabling a collection of FPGAs to be treated as a single, gigantic FPGA. In the field of FPGA computing, more tools like Virtual Wires are needed to efficiently utilize increasing amounts of available FPGA gate capacity. A future direction is to speed up and potentially multiplex the place and route inside the FPGA. However, the current greatest deficiency for computing is in software compilation techniques to quickly map applications from a higher level language, such as C or Behavioral Verilog, into FPGA instructions.

9 Acknowledgments

The research reported in this paper was funded by ARPA contract # N00014-91-J-1698 and NSF grant # MIP-9012773.

References

[1] A. Agarwal. VirtualWires: A technology for massive multi-FPGA systems. Technical report, Virtual Machine Works, Inc., December 1994. http://www.ikos.com/products/virtualwires.ps.

[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. *IEEE Micro*, 13(3):48–61, June 1993.

[3] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95124. *Flex 8000 Handbook*, May 1994.

[4] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–324, June 1992.

[5] Atmel Corporation. *Atmel Configurable Logic Design and Application Book*, 1994.

[6] J. Babb. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulation. Master's thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, February 1994. Also as MIT/LCS TR-586, November 1993.

[7] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators. In

Proceedings IEEE Workshop on FPGA-based Custom Computing Machines, pages 142–151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, January 1993.

[8] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.

[9] T. Bauer. The Design of an Efficient Hardware Subroutine Protocol for FPGAs. Master's thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, May 1994.

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

[11] N. Bhat. *Novel Techniques for High Performance Field Programmable Logic Devices*. PhD thesis, University of California, Berkeley, Electronic Research Laboratory, November 1993.

[12] J. Cocke and R. E. Miller. Configurable computer system. Technical Report 9, IBM Technical Disclosure Bulletin, Feb. 1973.

[13] M. Dahl. An implementation of the Virtual Wires interconnect scheme. Master's thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, February 1994.

[14] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki, and A. Agarwal. Emulation of a SPARC microprocessor with the MIT Virtual Wires Emulation System. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 14–22, Napa, CA, April 1994. IEEE.

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

[16] A. Dehon and S. Perentz. Transit Note No. 67: Transit Sbus Interface. Technical report, Artificial Intelligence Laboratory, MIT, June 1992.

[17] S. Hanono. Innerview hardware debugger: A logic analysis tool for the Virtual Wires Emulation System. Master's thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, February 1995.

[18] S. Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Department of Computer Science and Engineering, June 1995.

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

[20] D. Jones and D. Lewis. A time-multiplexed FPGA architecture for logic emulation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, May 1995.

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

[22] 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.

[23] B. Landman and R. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C-20(12), Dec. 1971.

[24] J. Li and C.-K. Cheng. Routability improvement using dynamic interconnect architecture. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, CA, Apr. 1995.

[25] C.-C. Lin, D. Chang, Y.-L. Wu, and M. Marek-Sadowska. Time-multiplexed routing resources for FPGA design. In *Proceedings, 1996 ACM International Workshop on Field-Programmable Gate Arrays*, Monterey, CA, Feb. 1996.

[26] LSI Logic Corporation. *0.7-Micron Array-Based Products Databook*, April 1990.

[27] V. MaheshWari, J. Darnauer, J. Ramireza, and W. W.-M. Dai. Design of FPGAs with area I/O for field programmable MCM. In *FPGA '95*, Monterey, California, Feb. 1995.

[28] L. Maliniak. Hardware Emulation Draws Speed from Innovative 3D Parallel Processing Based on Custom ICs. *Electronic Design*, May 1994.

[29] F. P. Manning. *Automatic Test, Configuration, and Repair of Cellular Arrays*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1975.

[30] H. Owen, U. Khan, and J. Hughes. FPGA-based ASIC hardware emulator architectures. In *Proc. International Workshop on Field Programmable Logic and Applications*, Oxford, UK, Sept. 1993.

[31] G. F. Pfister. The Yorktown Simulation Engine: Introduction. In *Proc. 19th Design Automation Conference*, pages 51–54. IEEE Computer Society Press, 1982.

[32] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb. TIERS: Topology independent pipelined routing and scheduling for VirtualWire compilation. In *1995 ACM International Workshop on Field-Programmable Gate Arrays*, Berkeley, CA, February 1995. ACM.

[33] Synopsys, Inc. *Command Reference Manual, Version 3.0*, Dec 1992.

[34] R. R. T. Cormen, C. Leiserson. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1992.

[35] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A first generation DPGA implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, May 1995.

[36] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal. The Virtual Wires Emulation System: A gate-efficient ASIC prototyping environment. In *1994 ACM International Workshop on Field-Programmable Gate Arrays*, Berkeley, CA, February 1994. ACM.

[37] J. Varghese, M. Butts, and J. Batcheller. An efficient logic emulation system. *IEEE Transactions on VLSI Systems*, 1(2), June 1993.

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

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

[40] XILINX, Inc., 2100 Logic Drive, San Jose, California, 95214. *The Programmable Gate Array Data Book, The XC4000 Data Book*, Aug. 1992.