Logic Emulation with Virtual Wires

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

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%. Global routing contention mandates the use of expensive crossbar and PC-board technology in a system 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%. Theoretical analysis predicts that virtual wires emulation scales with FPGA size and average routing distance, while traditional emulation does not.

I. INTRODUCTION

FIELD programmable gate array (FPGA)-based logic emulators are capable of emulating complex logic designs at clock speeds four–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 FPGA’s. The netlist partition on each FPGA, configured directly into logic circuitry, is then executed at near hardware speeds.

Fig. 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 1 min. 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 (2-D) meshes. Important target FPGA properties include gate count, pin count, and mapping efficiency. Supporting software is expensive, 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 FPGA’s with custom crossbar 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.

A. 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 interpartition emulated signal is allocated to a pair of pins.
between two FPGA's. 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 (Fig. 2).

When typical circuits are mapped onto available FPGA devices, partitions are predominantly pin limited. That is, all available FPGA gates cannot be utilized due to a 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 V-A) for various gate counts. Fig. 3 shows the resulting curves, plotted on a log–log scale. Partition gate count is scaled by a factor of 2 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 FPGA's: 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 FPGA's.

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 FPGA's 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 interboard 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. 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.

However, by clocking physical wires at the maximum frequency of the FPGA technology, several logical connections can share the same physical resource. Fig. 4 shows an example of six logical wires allocated to six physical wires. Fig. 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 III). 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.

Although this paper focuses on logic emulation, virtual wires can be applied to any multichip system.
BABB et al.: LOGIC EMULATION WITH VIRTUAL WIRES

B. Emulation Software

Software for logic emulation with virtual wires roughly follows the standard emulation tool flow (Fig. 6). The input, a netlist of the logic design to be emulated, is transformed into a multi-FPGA configuration bit stream 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.

Global Placer: Individual circuit partitions must be placed into specific FPGA’s. An ideal placement minimizes system communication, thus requiring fewer 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/O’s 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 bit streams. We used the XACT [40] software for Xilinx FPGA’s.

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 (Fig. 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 II, 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 tristate buses are carefully optimized. The resulting synthesized architectures provide
insight into virtual wires implementation. Section III compares three different architectures for the Xilinx 4000 series.

C. 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 FPGA’s. To demonstrate this advantage, we have constructed FPGA boards composed of 16 mesh-connected FPGA’s and commodity SRAM’s. These boards may themselves be mesh-connected, leading to straightforward software mapping and simplified system scalability. This system (Fig. 7), described in Section IV, 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 V 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.

D. Scalable Technology

Not only can virtual wires be used to compose low-cost systems of gigantic numbers of FPGA’s, but this technology also scales as FPGA sizes increase. To demonstrate this scalability, Section VI 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.

E. Overview

The rest of this paper is organized as follows. Section II describes the virtual wires scheduling and routing algorithms.

II. 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 2 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.

A. Phase-Based Operating Principles

The **emulation clock** period is the clock period of the logic design being emulated. To facilitate multiplexing, we break this period into a number of **microcycles** determined by a free-running **μCLK** (Fig. 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 interchip communication speed, allowing high-speed communication cycles to coexist with a long-latency emulation clock period.

A **μEnable** 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 FPGA’s. 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.

B. Definition of Dependence and Depth

Two timing analysis computations, dependence and depth, aid in virtual wires scheduling. Both dependence and depth apply to interpartition 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 combinatorially 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 of the inputs of that primitive. Similarly, no outputs are dependents of any of the inputs for sequential primitives.

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

Fig. 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:

- $\text{Depend}[\text{In}_1] = \{\text{Out}_1\}$
- $\text{Depend}[\text{In}_2] = \{\text{Out}_1, \text{Out}_2\}$
- $\text{Depend}[\text{In}_3] = \{\text{Out}_1, \text{Out}_2\}$.

Likewise, the ancestors are as follows:

- $D^{-1}[\text{Out}_1] = \{\text{In}_1, \text{In}_2, \text{In}_3\}$
- $D^{-1}[\text{Out}_2] = \{\text{In}_2, \text{In}_3\}$
- $D^{-1}[\text{Out}_3] = \emptyset$ (REG is a storage element).

The depth calculations use the dependence relationships. The depth of interpartition 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 $i$:

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

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

- depth 0: $W_4, W_6$
- depth 1: $W_2, W_3, W_5$
- depth 2: $W_1$.

Our phase assignment algorithm uses depth to prioritize routing of critical paths. During scheduling, although both $W_3$ and $W_2$ have no ancestors, $W_1$ has a greater depth and will be given priority.

C. 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 FPGA’s. 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 main loop of the phase assignment algorithm, Fig. 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 FPGA’s, any design can be scheduled. This is easier than hard-wired routing problems, in which various rip-up and retry 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, $\text{Done}[i]$, is set to false to mark each wire $i$ as unscheduled. The algorithm then initializes a $\text{DependCount}[i]$ array from the dependence information of each wire. When this counter reaches zero, 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 shift group, 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 there are eight cycles per phase and the distance is 3, a total of five wires can be routed in the same shift group. The additional wires are also prioritized by depth. For each routed wire $j$ in the shift group, $\text{Done}[j]$ is set, and the set of wires $k$ in $\text{Depend}[j]$ is iterated to decrement $\text{DependCount}[k]$. If $\text{DependCount}[k] = 0$, 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 V, it turns out that eight communication cycles match the eight-way tristate busing of the Xilinx architecture.

**D. Route Algorithm**

The goal of the routing algorithm (Fig. 12) is to find a shortest available path, in terms of FPGA’s, between the source and destination FPGA of a set of interpartition wires. The algorithm keeps track of the reserved and available physical connections between FPGA’s 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, $\text{Reserve}[i, j]$, is initialized to the number of physical connections between FPGA’s $i$ and $j$ in the emulator topology. Route applies Dijkstra’s shortest path algorithm [34] to channel availability, $\text{Avail}[i, j] = (\text{Reserve}[i, j] \neq 0)$, to determine the

---

**Procedure PhaseAssign**

- call Depend ← $\text{CalcDependents}(l)$
- call Depth ← $\text{CalcDepth}(\text{Depend})$
- initialize $\text{WaitCount}$ array to zero
- initialize $\text{Done}$ array to false

for each wire $i \in I$
  
  for each dependent wire $j \in \text{Depend}[i]$
    
    $\text{DependCount}[j] \leftarrow \text{DependCount}[j] + 1$
  
endfor

n ← 0 /*phase counter*/

loop forever
  
  call RouteInit
  
  W ← wires with $\text{Done}[j] = \text{false}$ and $\text{DependCount}[i] = 0$
  
  if W is empty exit loop
  
  n ← n + 1
  
  sort W by Depth[i], greatest depth first
  
  for each wire $i \in W$
    
    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 in
      
      depth order from W with same src and dest as i
      
      for each $j \in L$
        
        delete $j$ from W
        
        $\text{Done}[j] \leftarrow \text{true}$
        
        for each $k \in \text{Depend}[j]$
          
          $\text{DependCount}[k] \leftarrow \text{DependCount}[k] - 1$
        
      endfor
      
      shiftgroup ← { n, path, L }
      
      add shiftgroup to schedule $S$
    
  endif

endloop

save n, c in schedule $S$ /*final phase and cycle count*/

end Procedure

---

Fig. 11. Phase assignment algorithm.
Fig. 12. Routing algorithm.

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

Produce:
- path: list of FGPA's along shortest route from src to dest

Procedure RouteInit
  for each FPGA src ∈ T
    for each FPGA dest ∈ T
      Record[src,dest] ← connections in T from src to dest
      Avail[src,dest] ← (Record[src,dest] ≠ 0)
  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 */
        srcl ← f
      else /* following FGPA's in path */
        destl ← f
        Record[srcl,destl] ← Record[srcl,destl]-1
        Avail[srcl,destl] ← (Record[srcl,destl] ≠ 0)
        srcl ← f
    endif
  endif
  return path
else
  return null path
endif
end Procedure

Fig. 13. Shift group data structure.

shortest path between the source and destination FPGA’s. 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 (Fig. 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.

E. 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:

\[ v = n \times c \]  \hspace{1cm} (2)

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

\[ n \geq L \]

\[ c \geq D \]  \hspace{1cm} (3)

where \( D \) is the maximum distance of any shift group 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:

\[ v \geq \frac{P_c}{P_f} \]  \hspace{1cm} (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:

\[ v \geq \max \left( L \times D , \frac{P_c}{P_f} \right) \]  \hspace{1cm} (5)

where \( L \) is the 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 predominantly by the latency bound.

F. 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 the following.

- 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.

\(^2\)Note that we have ignored pipeline startup overhead associated with each shift group.
• 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.

### III. 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 FPGA’s.

#### A. Synthesis Algorithm

The virtual wires synthesizer flowchart component in Fig. 6 takes the following input:
- emulator topology
- external design I/O pin assignment

Given:

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

Procedure Produce:

- set of virtualized FPGA netlists

Procedure Synthesize

\[
\begin{align*}
N &\leftarrow \text{number of phases in } S \\
C &\leftarrow \text{cycles per phase in } S \\
\text{for each FPGA } f \in T & \quad \text{given } (N,C) \text{ synthesize control logic } \text{vwFsm}[f] \\
\text{endfor} \\
\text{for each phase } P \in S & \quad \text{for each shiftgroup } G \in P \\
R &\leftarrow \text{inter-FPGA routing path for } G \\
L &\leftarrow \text{number of logical wires in shiftgroup } G \\
\text{for each FPGA } f \in R & \quad \text{if } f \text{ is first FPGA in } R \text{ then} \\
\text{logic} &\leftarrow \text{synthesize output shifter of length } L \\
\text{assign each logical output in } G \text{ to logic} \\
\text{assign physical FPGA output pin to logic} \\
\text{else if } f \text{ is an intermediate FPGA in } R \text{ then} & \quad \text{logic} \leftarrow \text{synthesize intermediate hop} \\
\text{assign physical FPGA I/O pins to logic} \\
\text{else */ if } f \text{ is last FPGA in } R */ & \quad \text{logic} \leftarrow \text{synthesize input shifter of length } L \\
\text{assign each logical input in } G \text{ to logic} \\
\text{assign physical FPGA input pin to logic} \\
\text{endif} \\
\text{assign control nets for vwFsm}[f] \text{ to logic} \\
\text{virtualLogic}[f] &\leftarrow \text{virtualLogic}[f] + \text{logic} \\
\text{endif} \\
\text{endfor} \\
\text{for each external I/O } e \in E & \quad \text{assign } e \text{ to its specified FPGA physical pin} \\
\text{endfor} \\
\text{for each FPGA } f \in T & \quad \text{partition}[f] \leftarrow \text{design partition in D placed on } f \\
\text{X}[f] &\leftarrow \text{virtualLogic}[f] + \text{partition}[f] \\
\text{endfor} \\
\text{end Procedure}
\end{align*}
\]

Fig. 16. Synthesis algorithm.
determined solely by the number of phases and microcycles per phases in the schedule. The VW–FSM (Fig. 15) logic takes as input the $\mu$CLK and the $\mu$Enable signals, distributed to each FPGA, and generates the appropriate control signals during each microcycle. As described in Section II, the $\mu$CLK is the free-running pipeline clock, while the $\mu$Enable 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 shift groups in each phase to construct the input, intermediate hop, and output architectures. Each shift-group 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 buses 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 bit stream.

B. Shift Register Architectures

We now compare three shift register architectures synthesizable to Xilinx 4000 FPGA’s.

**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 (Fig. 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 (Fig. 18), forming a shift register pipeline between source and destination. At the end of the pipeline, corresponding input mode shift loops demultiplex 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-pin-count 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 (Fig. 19). Due to pad configuration constraints, these shift registers cannot be parallel-loaded, so they are not usable for output shift groups. However, the synthesizer can place input shift groups and intermediate hop shift registers here. Since input shift groups 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 $\mu$CLK. 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 shift groups is adjusted to divide evenly into the number of $\mu$CLK’s 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 $\mu$CLK rate.
Addressable Shift Register: A further variation is to replace the output shift-group shift registers with tristate multiplexers available in the Xilinx architecture (Fig. 20). The synthesizer creates an additional set of global control signals, labeled cycle enables, to enable each bit of the multiplexer during the appropriate microclock 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 tristate 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].

Comparisons: Table I compares each architecture in implementing the smallest benchmark circuit, Palindrome (see Section V-A), on the 16-FPGA demonstration hardware presented in Section V. Speed is measured in terms of the $\mu$CLK 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

<table>
<thead>
<tr>
<th>Resource</th>
<th>Design Logic (Virtual Overhead)</th>
<th>Total Logic (Virtual Overhead)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fig</td>
<td>Fig</td>
</tr>
<tr>
<td>Packed CLBs (Total)</td>
<td>54 (32%)</td>
<td>79 (33%)</td>
</tr>
<tr>
<td>Lookup Tables</td>
<td>115 (30%)</td>
<td>165 (31%)</td>
</tr>
<tr>
<td>(Combinational)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td>74 (48%)</td>
<td>141 (27%)</td>
</tr>
<tr>
<td>(Sequential)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xilinx PIPs (Routing)</td>
<td>1009 (42%)</td>
<td>1729 (48%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Resource Usage per FPGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu$CLK Speed</td>
<td>33MHz</td>
<td>24MHz</td>
</tr>
<tr>
<td>Emulation Speed</td>
<td>1.2MHz</td>
<td>.89MHz</td>
</tr>
<tr>
<td>Maximum Clock Speed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
resource consumption from total resource consumption. We measured emulation logic resource consumption by compiling unvirtualized partitions onto high-pin-count FPGA’s. CLB refers to the basic Xilinx combinational logic block, which includes both combinational lookup tables and sequential registers. We list the programmable interconnect points (PIP’s) 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 generally has 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].

IV. DEMONSTRATION HARDWARE SYSTEM

Our demonstration hardware building block is a scalable emulation board [36] which is inexpensive to manufacture and easy to build (Fig. 21). One or more boards are interfaced to a host workstation. Each board contains 16 Xilinx XC4005 FPGA’s [40] interconnected in a 2-D nearest neighbor mesh. The board is six layers, uses only through-hole devices, and is 10 in² 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 the 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.

A. FPGA Array

To emphasize the utility of virtual wires for interchip communication, we used no expensive crossbar chips and only low-pin-count FPGA’s (84-pin PLCC’s). These FPGA’s may be clocked at speeds approaching 40 MHz, thus resulting in small interchip delays and high emulation throughput. Fig. 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 (Fig. 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 pin groups are located along each side of the package. This connection scheme is analyzed in detail in [18].

B. 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 2-D mesh (Fig. 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. \( \mu \text{CLK} \) signals are fanned out to local logic at the destination boards using a clock distribution chip. Bidirectional 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 FPGA’s 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 bit stream, although this limit may be overcome with multiple download cables.

### C. Memory and Host Interfaces

Each FPGA in the mesh has 22 dedicated I/O lines which interface to a 64K x 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. The SRAM’s used in the current prototype are 20 ns. 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 8 bit north port of the Xilinx chip in the upper right position of the array.

D. 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 these 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 FPGA’s 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 FPGA’s on a chip-by-chip basis. The controller can be programmed to trigger a readback bit stream from any FPGA in the system, and subsequently transfer the values back to the host workstation for evaluation.

V. 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 3–4 h on a SparcStation 10, with 90% 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.

A. Emulation Benchmarks

Let us introduce relevant features of three benchmark designs for this paper (Table II). 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 PARC 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.

B. Simulation Acceleration

We have collected results from the successful simulation acceleration of Sparcle and Palindrome (Table III). 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 (2.1 kHz) or the bandwidth across the host interface (41 kHz for S-bus or 30 Hz for serial port). Fig. 25 shows the allocation of resources inside each FPGA for Sparcle. While there is a fixed overhead of roughly 12% of the CLB’s for virtual wires, usable CLB’s exceed 45%. Note that due to internal FPGA routing, total utilization approaching 100% is not achievable.

C. 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 III 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 180 kHz.

D. Comparison with Traditional Emulation

Table IV 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 microcycles, 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 FPGA's. Shown beside the hard-wired I/O's 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 multiplying the number of phase and cycles-per-phase by the CLK rate. Note that to achieve these speeds, FPGA’s 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 FPGA’s without virtual wires. However, in our earlier work [7], we did partition a version of Sparcle without memory or external I/O’s onto 100-pin, 5000-gate FPGA’s. We needed at least 31 FPGA’s if they were fully connected, and greater than 100 FPGA’s 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.

VI. 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.

A. 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:

\[ P = KG^B \]

where \( P \) is the number of pins, \( G \) is the number of gates, \( K \) is 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

### Table IV

<table>
<thead>
<tr>
<th>Comparison with Traditional Emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>FPGAs</td>
</tr>
<tr>
<td>Avg. partition Gates</td>
</tr>
<tr>
<td>Avg. partition I/O</td>
</tr>
<tr>
<td>Max. partition I/O</td>
</tr>
<tr>
<td>Emulation Speed (MHz)</td>
</tr>
</tbody>
</table>

FPGA-to-FPGA hops and FPGA-to-crossbar-to-FPGA delay, and 50 ns delay for internal FPGA logic partition paths. Thus, crossbar speed is \( L \times 20 \text{ns} + L \times 50 \text{ns} \), while mesh speed is \( L \times d \times 20 \text{ns} + L \times 50 \text{ns} \). Note that to achieve these speeds, FPGA’s with the required pin counts must be used to maintain the same critical path and route lengths.
pipeline stages. Table V 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 FPGA’s fall on a different curve due to their higher pin-to-gate ratio. For the other FPGA’s, 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 $K$.

### B. Hard-Wire Gate Utilization

For circuits that obey Rent’s rule, we can determine the gate utilization for hard-wires, under pin-limited conditions (Fig. 26). Given pin limitations, the number of FPGA pins $P_f$ dictates the number of circuit partition pins $P_c$ available for the circuit:

$$ P_c = \frac{1}{d} P_f $$

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. We next substitute Rent’s equation for both sides of (7):

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

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

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

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 VI) to simplify our work. Substitution of these newly defined parameters into (9) and dividing by $G_f$ yields the average per-FPGA utilization with hard wires:

$$ U_{hw} = \frac{1}{d_h} K_h G_f^{B_h} $$

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 nonideal 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 the average routing path length increases.

### C. Virtual Wires Gate Utilization (Fig. 27)

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:

$$ G_v = V_0 + V_1 d K_c G_c^{B_c} $$

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 FPGA’s $N$, as reported by Bakoglu [8], is

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

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

$$ G_f = G_c + G_v $$

Substituting (11) and rewriting yields

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

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

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

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 nonideal 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 the average routing path length increases.
Interestingly, (15), and the mathematical solution for $U_{v,w}$ 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 (15), it is apparent that if $B_c<1$, then as $G_j \to \infty$, virtual wire utilization will approach unity, $U_{v,w}$, independent of other parameters. However, for small $G_j$, the gate utilization will be low due to the dominating $B_c$ factor. The following section compares these results to the hard-wired case for realistic design and FPGA parameters.

D. 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 size increases. Fig. 28 shows utilizations for a hypothetical design, characterized by Rent parameters, on both a $4 \times 4$ mesh and a 16-chip crossbar topology. These graphs are on a semilog scale, with the $Y$ axis measuring percent of usable gates and the $X$ axis logarithmically measuring FPGA device size. Table VII 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 FPGA’s Rent parameters.

The slope of both hard-wire curves is $B_h = B_f/B_c - 1$. In Fig. 28, $B_h$ is negative, and thus the hardware curves slope downward. With a positive $B_h$, this curve would slope upward 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 hardwired mesh, the offset is lower than the crossbar by $1/d_h$ in accordance with (10). For this example, the FPGA pin counts decrease with decreasing gate count such that FPGA’s 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 $B_c$ control logic overhead. However, with increasing circuit size, this overhead is rapidly diluted: in both figures, the virtual wires gate utilization approaches 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.

E. Scalability with Routing Distance

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

For the hard-wires, the utilizations for $d = 1$ and $d = 2$ match the same data points as in Fig. 28. As $d$ increases, the utilization drops exponentially at the rate $d_h = d^{K_h}$ for all hard-wired cases. Additionally, note that the higher utilization curves correspond to lower gate count FPGA’s $G_f$ 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.

For the virtual wires curves, the higher gate count FPGA’s have a higher utilization, as demonstrated in Fig. 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 $G_7 = 100K$ curve has a utilization of 33% when $d = 16$. Thus, virtual wires enable emulation to scale to a gigantic number of FPGA’s using simple direct-connected topologies.

VII. RELATED WORK

IBM’s Yorktown Simulation Engine [31] and the earlier Logic Simulation Engine [12], based on concepts of J. Cocke, used reconfiguring digital hardware to accelerate logic simulation. Actual logic emulation was first explored in cellular array research, such as F. Manning’s 1975 thesis [29], even before FPGA’s 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 of large FPGA-based emulators, first developed emulation systems that interconnect FPGA’s in a 2-D 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 [6], [7] in 1993, and the first successful applications were discussed by Tessier [36], Dahl [13], [14], and Hanono [17]. Virtual wires technology has continued to evolve at Virtual Machine Works, Inc. [1], [32], where commercial emulators based on proprietary VirtualWires™ 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 FPID's. 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.

VIII. 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 retimze a synchronous input design to fit an arbitrary number and arrangement of FPGA’s. 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 have found that FPGA utilization, over 45% in our emulation experiments, will asymptotically approach 100% in larger FPGA’s 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 FPGA’s 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.

REFERENCES

Jonathan Babb received the the B.S. degree in electrical engineering from the Georgia Institute of Technology, Atlanta, in 1991, the S.M. degree in electrical engineering from the Massachusetts Institute of Technology (MIT), Cambridge, in 1994 and is currently pursuing the Ph.D. degree as part of the RAW project at MIT.

He was a previously researcher in the Alewife multiprocessor and the Virtual Wires projects at MIT and was a founder of Virtual Machine Works, Inc.

Russell Tessier received the B.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1989, and the S.M. degree in electrical engineering from the Massachusetts Institute of Technology (MIT), Cambridge, in 1992. He is currently pursuing the Ph.D. degree in computer science at the MIT Laboratory for Computer Science.

He is a founder of Virtual Machine Works, a commercial venture based on the concept of Virtual Wires. His research interests include computer architecture and CAD for field-programmable gate arrays.

Matthew Dahl is currently a Senior Staff Software Engineer at IKOS Systems, Inc., Cambridge, MA. He has previously been a member of the technical staff at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, and a Senior Engineer at Digital Equipment Corp.

Silvina Zimi Hanono received the B.S. degree in electrical engineering from Cornell University, Ithaca, NY, in 1992, and the M.S. degree in electrical engineering and computer science from the Laboratory for Computer Science, Massachusetts Institute of Technology (MIT), Cambridge, in 1995. There she developed a hardware debugger for the Virtual Wires Logic Emulation system. She is currently pursuing the Ph.D. degree in the Research Laboratory of Electronics at MIT.

Under the direction of Prof. S. Devadas, she is developing a retargetable code generator for VLIW embedded processors. Her research interests include computer architecture and computer-aided design.

Ms. Hanono is a National Science Foundation Fellow and a member of theEta Kappa Nu and Tau Beta Pi Engineering honor societies.

David M. Hohi received the B.S. degree in electrical and computer engineering from Brigham Young University (BYU), Provo, UT, in 1993.

He is currently a software engineer at Charles Stark Draper Laboratory, Inc., Cambridge, MA. He was a member of the research staff at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, from 1993 to 1994. He was also a software developer at CADAM Inc. for three years while attending BYU. He was recently building a Global Positioning System (GPS) captive simulator, and is currently doing research in multi-project threaded planning and decision aids.

Anant Agarwal received the Ph.D. degree from Stanford University, Stanford, CA, in 1987 and the B.Tech. degree from the Indian Institute of Technology, Madras, India, in 1982.

He is with the Laboratory for Computer Science at the Massachusetts Institute of Technology (MIT), Cambridge, as an Associate Professor of Electrical Engineering and Computer Science. At Stanford, he participated in the MIPS and MIPS-X projects. He led the Alewife multiprocessor and the Virtual Wires projects at MIT. He was a founder of Virtual Machine Works, Inc., which was aimed at productizing the VirtualWires technology for logic emulation. He has recently initiated the RAW project, which is aimed at developing a configurable computer based on a new model of computation.