Incremental Compilation for Logic Emulation

Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, Ma. 01003.
tessier@ecs.umass.edu

Abstract

Over the past decade, the steady growth rate of FPGA device capacities has enabled the development of multi-FPGA prototyping environments capable of implementing millions of logic gates. While software support for translating new user designs from gate and RTL-level netlists to FPGA bitstreams has improved steadily, little work has been done in developing techniques to support the translation of *incremental* design changes at the netlist level to a set of replacement bitstreams for a small number of FPGAs in a multi-FPGA system. As system sizes and design compilation times increase, the need to support rapid, incremental compilation grows progressively important.

In this paper we describe and analyze a set of incremental compilation steps, including incremental design partitioning and incremental inter-FPGA routing, for two specific classes of multi-FPGA emulation systems. These classes are defined by the techniques that emulation software systems use to determine inter-FPGA communication. In *hard-wired* emulation systems each logic signal traversing an FPGA boundary is dedicated to a physical inter-FPGA wire and this assignment remains static during the execution of the prototyped design. In contrast, for *virtual-wired* systems, inter-FPGA wires are multiplexed over time during design execution to support the communication of multiple logical signals using the same physical resources.

Through experimentation we find that while incremental compilation techniques can be applied both to hard-wired and virtual-wired systems, a lack of available FPGA pin resources frequently limits their applicability in the hard-wired case. In contrast, it is shown that incremental techniques can be seemlessly integrated into virtual-wired systems resulting in a valid implementation of a modified design and requiring the need to re-place and re-route only a small fraction of FPGAs in the target system.

1 Introduction

Even though contemporary FPGA device capacities have reached the one million gate threshold, many logic emulation and prototyping tasks still require significantly more logic resources than can fit in a single FPGA device. As a result hardware systems containing tens and even hundreds of discrete FPGA devices have been developed and are currently being used for verification and computing. For systems of this complexity software-based automated translation of user designs to FPGA-based hardware is a must.

Existing software systems for multi-FPGA emulation equipment typically contain a number of automated steps to translate a gate-level or RTL netlist to the FPGA-based hardware. Included in these steps is a partitioner to separate the user design into pieces that will fit in each device, a placement step to select the appropriate device to hold each design partition, a router to interconnect inter-partition wires using board wiring resources, and FPGA place and route software to perform individual device layout. By far the most computationally expensive task of this design flow is FPGA place and route which can require several hours *per device* given tight timing constraints and capacity limitations. Initial compilation of a prototype design may require many hours at the FPGA place and route stage, even if multi-device compiles are performed in parallel across a network of workstations.

The contemporary design flow of many verification and reconfigurable computing tasks typically involves several iterations of user design modification during the development of an ASIC or reconfigurable computing application. Frequently, this change can be isolated to a single or small number of RTL components that is substantially smaller than the overall design but encompasses more than a single FPGA partition. The need in software to allow changes to the design in this small set of devices is crucial to avoid the need to re-compile all FPGAs in the system from scratch leading to long delays from design change to physical implementation.

To support design changes software for multi-FPGA systems must be able to support incremental emulation system tasks such as design partitioning to reapportion the new logic of the user design netlist onto affected devices after modified logic and nets have been removed and incremental routing to construct feasible paths for new inter-FPGA connections that have been added to the system. In this paper we apply incremental partitioning and routing to two classes of multi-FPGA software systems, those that dedicate individual logical wires to inter-FPGA routing resources and those that support the multiplexing and pipelining of multiple logical signals onto inter-FPGA resources during the execution of the prototyped design. It is shown that the latter case simplifies incremental compilation and allows incremental design changes to be made much more easily than the former at the cost of modest reduction in emulation system performance. To exhibit the direct benefits of this new incremental compilation approach, incremental partitioning and routing have been integrated in the Virtual Wires Emulation System [2] [11] and applied to several benchmark designs.

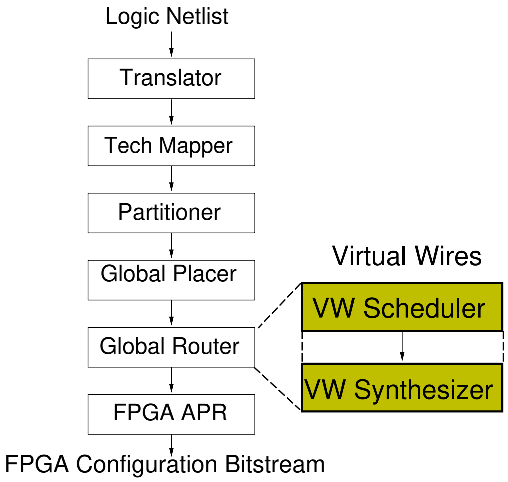

Figure 1: Emulation software flow

The rest of the paper is structured as follows. Section 2 discusses previous work in FPGA-based logic emulation hardware and software and the basic system design flow. In Section 3 software requirements to support incremental compilation are reviewed. Our incremental approach to partitioning is described in Section 4 and incremental routing is addressed in Section 5. Section 6 describes experimental results that have been derived from our system. Finally, in Section 7, we summarize this work and make concluding remarks.

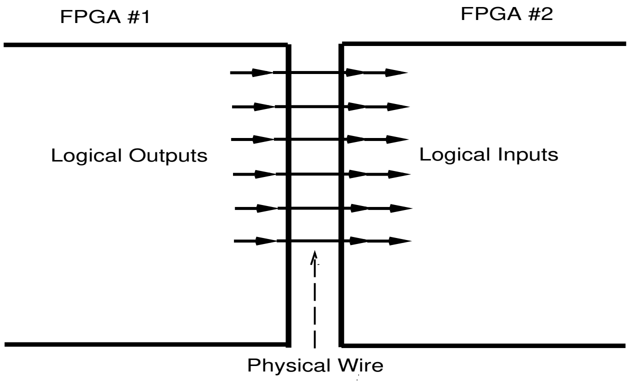

Figure 2: Hard-wire interconnect

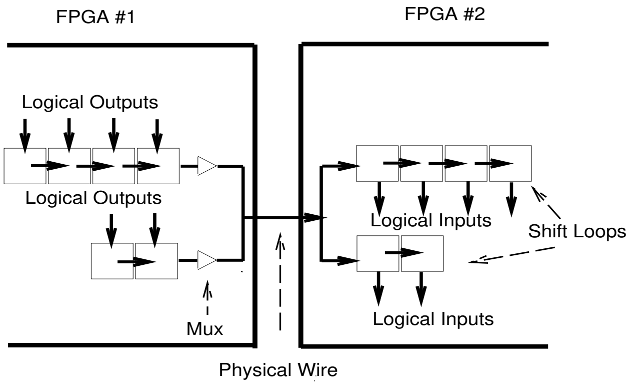

Figure 3: Virtual wire interconnect

2 Background

A sizable number of multi-FPGA logic emulation systems have been developed that support complex designs containing millions of logic gates. In recent years increased system capacity has been achieved primarily through the exponential growth in FPGA device capacity. A brief summary of several large multi-FPGA emulation systems is presented below. More thorough reviews of contemporary emulation and prototyping systems are provided in [5].

Quickturn Design Systems [12] first developed emulation systems that interconnect FPGAs in a 2-D mesh and later in a partial crossbar topology. Current Quickturn FPGA-based systems use a hierarchical approach to interconnection that employ field-programmable interconnect devices (FPIDs) [12] to enhance board-level routability. Recent multi-FPGA emulation systems from Ikos Systems [6] have returned to the 2-D mesh model by developing emulation hardware that contains no FPIDs and supports primarily near-neighbor inter-FPGA communication. Multi-FPGA systems created by both Quickturn and Ikos have a capacity of several hundred FPGAs distributed across several PCBs.

Contemporary compilation software for multi-FPGA systems has evolved significantly as component FPGA device capacities and system sizes have increased. A typical multi-FPGA system software flow for converting a structural or RTL netlist to a physical realization appears in Figure 1. Translation and technology mapping converts the original design netlist into the target library of system FPGAs. Logic partitioning takes the input netlist and divides it into pieces, each of which will fit in a target FPGA. Global placement algorithms assign individual circuit partitions to specific FPGAs while attempting to minimize system communication. While the previous steps are similar across several types of emulation systems, significant differences appear in the inter-FPGA global routing process.

In traditional emulation, inter-FPGA communication is established with a global routing phase. In this phase, each inter-FPGA signal is routed from a source FPGA to one or more destinations using board-level wiring. If crossbars are employed, this phase must also determine the routing configuration for each crossbar in addition to signal FPGA pin assignments. Iterative path determination algorithms, such as maze routing, search through available routing paths in an effort to determine connections that minimize routing congestion and cost leading to minimized delay and resource utilization. As seen in Figure 2, effectively, logic signals are *hard-wired* to a specific set of physical inter-FPGA routing resources.

Virtual-wired routing systems add a time dimension to routing to create paths between FPGAs in both space *and* time. In addition to physically routing individual paths between FPGAs (e.g. one path in Figure 3) as in the hard-wired case, the virtual-wired global router *schedules* inter-FPGA communication by assigning multiple logical signals to the same physical wire and pipelining the communication path at the clock rate of the FPGA. This schedule establishes a feasible time-space route for every logical wire, while guaranteeing that all FPGA combinational dependencies are correctly ordered. Following scheduling, communi-

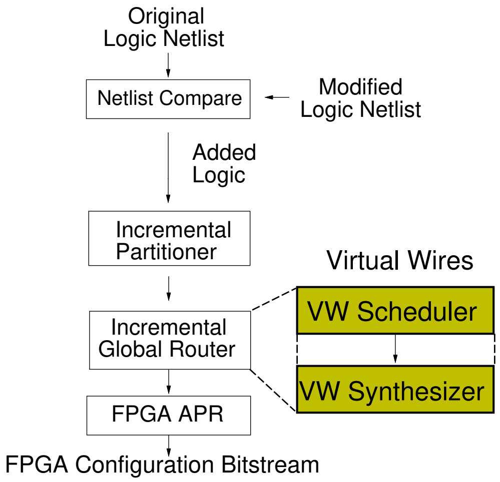

Figure 4: Incremental compilation flow

```
Assign unchanged nodes to previously assigned bins.
Randomly assign new design nodes to balance bins.
While cutsize is reduced.
Unlock new design nodes.
    Add locked dummy nodes to represent external connects.
    While valid moves exist
        Find unlocked node that most improves cutsize.
        Move whichever node most improves cutsize.
        Lock moved node.
        Update affected nets and node.
    endwhile
    Backtrack to point with minimum cutsize.
endwhile
```

Figure 5: Modified KLFM Algorithm

cation hardware, like the shift loops pictured in Figure 3, are synthesized from FPGA logic to allow for pipelined communication.

A number of academic and commercial software systems have been implemented that support virtual-wired routing between FPGAs. The first implementation of pin multiplexing for FPGA-based emulation systems was developed for the Virtual Wires project [2]. This work was later commercialized by Ikos Systems and implemented in emulation systems containing hundreds of FPGAs. Other implementations of virtual-wired routing to support FPGA-based logic emulation have also been reported in [8], which augmented the multiplexing of FPGA pins with the multiplexing of FPIDs, and [9] and [10] which developed enhanced versions of virtual wires scheduling algorithms.

3 Incremental Compilation Software Flow

A desirable goal for incremental compilation is to physically implement required logic design changes through modification of a minimum number of FPGA devices. By limiting the number of FPGAs affected, the number of new FPGA place and route compiles are reduced and incremental turnaround time is significantly improved.

Typically, when design changes are added to a user design, some existing logic is replaced with new logic. In order to successfully complete incremental compilation, the size and interconnect structure of the changed piece of logic must fit within the available resources of the system FPGAs that previously contained the removed logic. In our new design flow the incremental compilation system attempts to reallocate previously used resources amongst the added design logic by re-partitioning added logic across modified FPGAs and incrementally routing new design nets across routing resources no longer needed by removed logic.

A combined incremental design flow for both virtual-wired and hard-wired logic emulation systems is shown in Figure 4 with virtual wires specific steps shaded. The initial flow step extracts the modified portion of the user design and determines the minimum number of FPGAs that must be modified. Changed portions of the user design include both logic and nets that have been added to the design and those that have been removed. Any FPGA in contact with changed logic and nets must be re-compiled.

Incremental partitioning divides logic and nets that have been added to the design across devices that contain design logic that has changed. As is described in the next section, new logic is partitioned across the target FPGA set to minimize interconnect between the devices using a modified K-way partitioner based on mincut techniques. It will be shown in Section 6 that for hard-wired systems, a lack of external device pins makes these incremental changes difficult since the bandwidth between affected devices following incremental partitioning is typically higher than that created by the original partitioning and the additional bandwidth frequently overflows available pin counts on FPGA packages.

Following incremental partitioning, incremental routing is performed to create a path for the new design signals connecting the FPGAs that have had contents modified. Since other FPGAs surrounding the modified FPGAs have not changed, incremental routing needs to be performed using board-level routing resources that have not previously been consumed by still-existing design routes. For hard-wired systems each new inter-FPGA wire that has been created by incremental partitioning is assigned to dedicated routing resources. For virtual-wired systems both inter-FPGA path creation and incremental scheduling is needed to form a communication pipeline.

4 Incremental Partitioning

Incremental partitioning of added design logic onto modified FPGAs follows directly from the basic Kernighan-Lin, Fiduccia-Mattheyses (KLFM) [7] [4] bipartitioning algorithm. To promote design quality. this algorithm has been supplemented with several optimizations to take unchanged logic and connections to unchanged, fixed-placement FPGAs into account. An illustration of these optimizations can be seen in the example circuit shown in Figure 6 that contains flip flops A, D, and E that remain from an original design and logic components B and C that have been added to the design and now must be assigned to an FPGA. In the figure, a net from unchanged FPGA 1 connects to added inverter B and two nets from unchanged FPGA 4

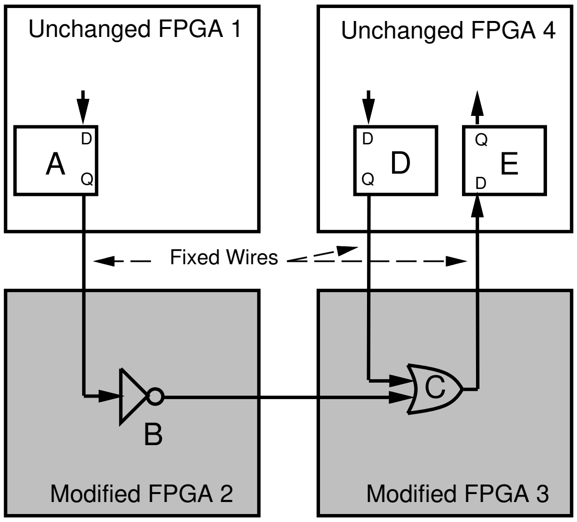

Figure 6: Incremental partitioning

connect to added OR gate C. These connections are preexisting from the original design and have been previously fixed to inter-FPGA routing resources. Simply bipartitioning logic targeted to the two modified FPGAs into balanced partitions without taking these fixed connections into account might lead to B and C being assigned to FPGAs 3 and 2, respectively, rather than to the shown, more optimal configuration. To alleviate this possibility during the partitioning process, extra *anchor nodes* are added to each partition, to reflect fixed external connections to partition logic. These nodes guide partitioning to the placement of logic in more desirable physical bins. Figure 5, modified from [5], contains several partitioning steps in italics which reflect this change as well as steps to ensure unchanged logic is re-assigned to the same bin to which it was initially allocated by pre-modified design partitioning. Anchor nodes are removed after partitioning and are not physically implemented. Note that this approach is effectly the same as terminal propagation [3] used in ASIC layout.

5 Incremental Routing Techniques

Following partitioning, added inter-FPGA signal connections must be made between changed FPGAs to allow the new circuit to be finalized. For the hard-wired case these connections are created by applying the board-level maze router and treating existing, unchanged inter-FPGA routes as pre-existing obstacles.

For virtual-wired systems, incremental routing is accomplished by *incrementally scheduling* the new inter-FPGA signals in pipeline slots that were previously used by removed inter-FPGA connections. If insufficient pipeline cycles from the implementation of the original design are available for communication, additional pipeline cycles between the FPGA partitions can be added to accommodate increased bandwidth needs. To illustrate this technique, a modification to phase-based virtual wires scheduling, previously described in [2], is outlined. In the example timeline, shown in Figure 7, the emulation clock period is the clock period of the logic being emulated. This clock is broken down into a series of emulation phases, subcycles in

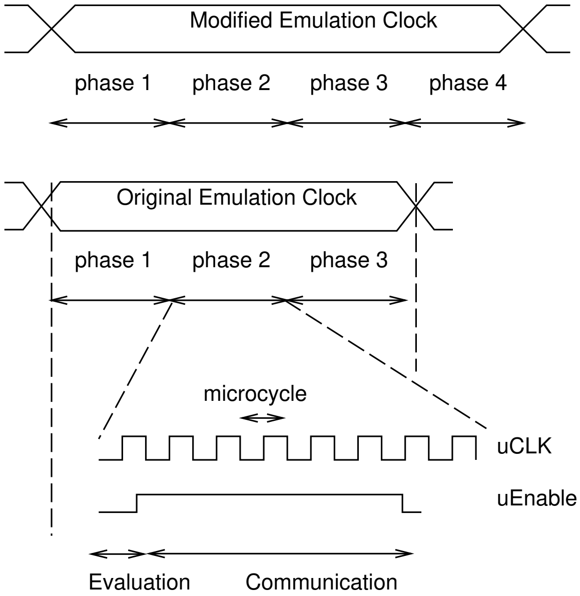

Figure 7: Clocking Framework

which partial evaluations of combinational design paths are performed. In this example, the original design uses three phases to evaluate a combinational path that has been partitioned and placed across three FPGA devices.

Each phase is divided into two parts: an evaluation portion and a communication portion. After combinational evaluation has completed, multiple cycles of microclock, *uClk*, are used to pipeline results between neighboring partitions. Complete evaluation of an emulation clock cycle is completed after the last phase has completed. The number of phases per emulation clock must be sufficient to allow computation in all FPGAs along all combinational paths in the system, effectively retiming communication to suit bandwidth needs.

Incremental scheduling attempts to re-fill inter-FPGA pipelined communication slots in each phase that were previously used by removed inter-FPGA signals with new inter-FPGA signals that have been added. If insufficient bandwidth in terms of existing pipeline slots exist, *additional phases* can be added at the end of the original schedule to allow the new signals to communicate, while keeping previously scheduled signals intact. The emulation clock can then be readjusted to reflect the overall emulation clock period as shown in Figure 7, where the number of phases has been increased from 3 to 4. The addition of an extra phase comes at the cost of a reduced emulation clock rate since the emulation clock must be extended to accommodate the additional signal delay.

6 Results

The incremental compilation system outlined in previous sections has been implemented and tested. To evaluate the limitations of incremental compilation for hard-wired systems we applied the system to two designs from the RAW Reconfigurable Computing Benchmark Suite that have been previously applied to FPGA-based logic emulation hardware [1], *ssp16*, a hardware implementation of a shortest

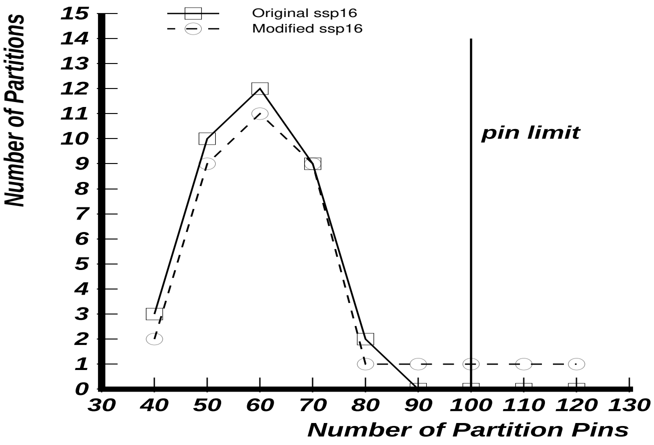

Figure 8: Partition pin counts - Design ssp16

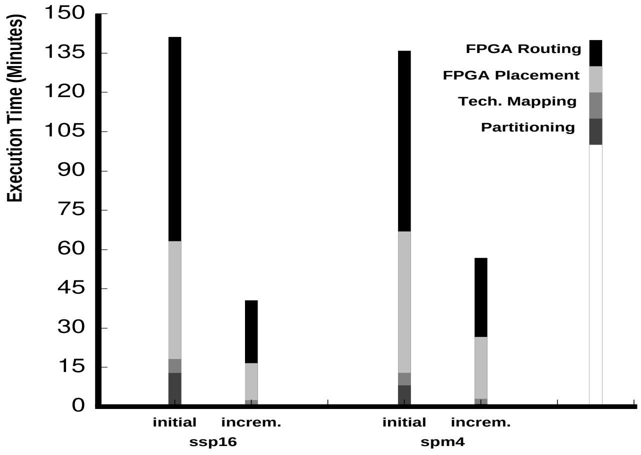

Figure 10: Compilation time - initial and incremental

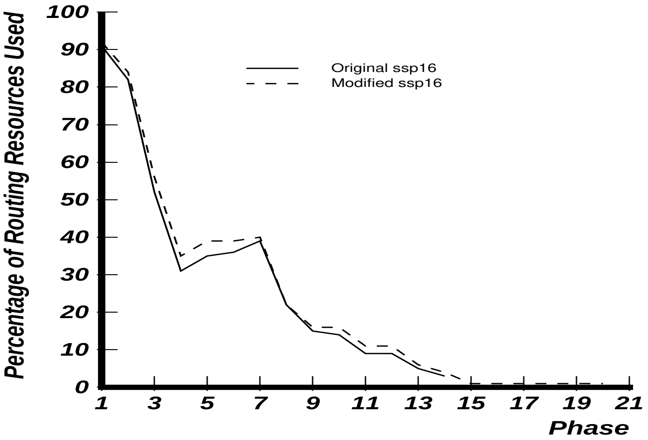

Figure 9: Per phase communication utilization - ssp16

path solver, and *spm4*, a hardware implementation of multiplicative shortest path. Each of these modular RTL benchmarks were easily modified to test the flexibility of the incremental compilation approach.

It has been determined previously [2] that pin limitations on FPGA packages greatly restrict hard-wired emulation systems. To demonstrate the limitations of incremental compilation for hard-wired systems we partitioned the two benchmark designs into logic partitions that would fit within devices containing 100 I/O pins and 5000 logic gates. Given the hard constraint of 100 pins, it was necessary to reduce the number of gates that were assigned to each partition for the hard-wired case to about 500-600 gates to meet both pin *and* logic constraints. After determining the minimum number of devices needed to implement the original design for the hard-wired case, an incremental design change was made to each design by replacing the functionality and interconnection of an RTL **node** component containing about 400 gates. The added logic was then re-apportioned across the partitions that previously contained the removed logic. For both designs tested, the pin counts on partitions that were assigned logic during incremental partitioning *exceeded* the 100 pin limitation indicating that hard-wired implementations of the modified designs would necessarily have to be re-created from scratch. The resulting required pin counts of partitions both before and after incremental partitioning for design *ssp*16 is shown in Figure 8. It can be seen that following incremental partitioning the pin count requirements of the design for the four FPGA devices affected by the design logic change exceeded those available in the emulation system hardware. Similar experiments on the *spm*4 benchmark resulted in the same conclusion.

While the sample set for above experiments was small and it is possible to envision modifications to other designs that do not significantly change inter-FPGA bandwidth, our belief is that, in general, incremental partitioning will not reveal the same minimal cut size between partitions as from-scratch partitioning due to variances between initial and ECO internal interconnects. This belief and the results described above for hard-wired systems motivates the following evaluation of incremental compilation for virtual-wired systems.

After evaluating incremental compilation for hard-wired systems, incremental compilation was applied to the same benchmarks targetted to the Virtual Wires Emulation System [11] containing a 4×4 array of 84 pin XC4010e-2 devices interconnected in a near-neighbor 2-D mesh and operating at a *uClk* rate of 30 MHz. In each case the original benchmark netlist was divided into 15 partitions of approximately 5000 gates by the K-way mincut partitioner, assigned to individual FPGAs through placement using simulated annealing, and routed using the phase-based routing technique described in Section 5. After successful initial implementation, the same design changes that were applied to the hard-wired case were made to the netlists and the design flow of Figure 4 was followed. Following the design change, the modified portion of each design was re-partitioned across the minimum number of devices needed to support the change (5 for *ssp*16, 7 for *spm*4), incremental scheduled routing was performed, and FPGA place and route was performed on the modified devices.

Table 1 shows that for both designs the design changes could be made successfully with minimal loss of system performance. In the case of *ssp*16, 6 additional communication phases totaling nine *uClks* were required to allow new inter-FPGA signals to be communicated. For design *spm*4,

	ssp16		spm4	
	original	incremental	original	incremental
FPGAs compiled	16	5	16	7
Critical Path Length	14 partitions	20 partitions	11 partitions	11 partitions
Average Route Length	1.95 FPGAs	1.88 FPGAs	2.08 FPGAs	2.07 FPGAs
Average Virtual-wire I/O	24	24	24	24
Average Hard-wire I/O	126	139	93	93
Virtual-wire phases	14	20	11	11
Virtual-wire *uClk*s	71	80	66	66
Virtual-wire Emulation Speed	0.42 MHz	0.38 MHz	0.45 MHz	0.45 MHz
Est. Hard-wire Emulation Speed (ideal)	0.99 MHz	0.70 MHz	1.21 MHz	1.21 MHz

Table 1: Emulation clock speed comparison

all communication of new signals could be overlapped with existing communication eliminating the need for additional communication phases and *uClks*. Since all computation is completed by the end of the original emulation clock cycle, no performance penalty is paid for the incremental *spm*4 design change and only a subset (7 out of 16) FPGAs needed to be re-compiled. Figure 9 illustrates that simply having additional inter-FPGA bandwidth in each phase may not be sufficient. Even though communication phases show limited routing resource utilization, additional phases are still required to complete communication due to combinational dependencies across partitions for design *ssp*16. Since hard-wired versions of the circuits were not physically implemented, it was necessary to estimate their performance using the method outlined in [2] using an internal FPGA clock rate of 30 MHz. and FPGA-to-FPGA delay of 20ns. These values are included in Table 1 for comparison.

Figure 10 illustrates the amount of compile time needed for each component of the compilation process. Tasks excluded from the figure, such as global placement and global routing, each required less than one minute to complete for all cases. Incremental partitioning was also very fast, completing in about one minute for each design. In total, incremental compilation for *ssp*16 and *spm*4 required 28% and 41% of the time of initial compile, respectively. All compilations were performed on a 166 MHz SparcStation 20.

7 Conclusions and Future Work

In this paper incremental design compilation for multi-FPGA logic emulation systems has been evaluated through the use of new algorithms for incremental partitioning and routing. It has been shown that while pin limitations restrict the capability of hard-wired systems to support incremental compilation, virtual-wired systems can overcome these limitations by scheduling changed signals during unused communication cycles or by adding additional communication cycles to the end of existing evaluation. For future work we will evaluate implementing additional incremental approaches using different scheduling approaches to further overlap computation and communication.

References

[1] J. Babb, M. Frank, V. Lee, E. Waingold, and R. Barua. The RAW Benchmark Suite: Computation structures for general purpose computing. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1997.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6), June 1997.

[3] A. E. Dunlop and B. W. Kernighan. A Procedure for Placement of Standard Cell VLSI Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-4(1), Jan. 1985.

[4] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improved network partitions. In *Proceedings: Design Automation Conference*, pages 241–247, 1982.

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

[6] Ikos Systems, Inc. *VirtuaLogic Emulation System Manual*, Feb. 1997.

[7] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning of Electrical Circuits. *Bell Systems Technical Journal*, 49(2), Feb. 1970.

[8] J. Li and C.-K. Cheng. Routability Improvement Using Dynamic Interconnect Architecture. *IEEE Transactions on VLSI Systems*, 6(3), Sept. 1998.

[9] 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.

[10] H.-P. Su and Y.-L. Lin. A phase assignment method for virtual-wire-based hardware emulation. *IEEE Transactions on Computer Aided Design*, 16(7), July 1997.

[11] 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.

[12] J. Varghese, M. Butts, and J. Batcheller. An Efficient Logic Emulation System. *IEEE Transactions on VLSI Systems*, 1(2), June 1993.