Static Scheduling of Multidomain Circuits for Fast Functional Verification

Murali Kudlugi and Russell Tessier

***Abstract*—With the advent of system-on-a-chip design, many application specific integrated circuits (ASICs) now require multiple design clocks that operate asynchronously to each other. This design characteristic presents a significant challenge when these ASIC designs are mapped to parallel verification hardware such as parallel cycle-based simulators and logic emulators. In general, these systems require all computation and communication to be synchronized to a global system clock. As a result, the undefined relationship between design clocks can make it difficult to determine hold times for synchronous storage elements and causality relationships along reconvergent communication paths. This paper presents new scheduling and synchronization techniques to support accurate mapping of designs with multiple asynchronous clocks to parallel verification hardware. Through analysis, it is shown that this approach is scalable to an unlimited number of domains and supports increasingly large design sizes. To prove the effectiveness of the authors' approach, developed algorithms have been integrated into the compilation system for a commercial multi-FPGA logic emulation system. For three designs mapped to a logic emulator using this software environment, modeling fidelity is maintained and performance is enhanced versus previous manual mapping approaches. A theoretical analysis based on Rent's rule validates the scalability of the approach as device sizes increase.**

***Index Terms*—Asynchronous circuits, FPGA-based emulation, functional verification, static scheduling.**

I. Introduction

As APPLICATION specific integrated circuit (ASIC) design sizes grow toward a billion transistors on a chip, the need for fast, effective verification becomes increasingly apparent. Although microprocessor-based simulators are still the dominant means of prefabrication functional verification, parallel verification platforms, such as logic emulators [4], [8], [14], [15], are increasing in importance. The inherent parallelism found in many system-on-a-chip designs necessitates parallel evaluation of functional resources that is difficult to accomplish on accelerated uniprocessor systems. Although specific system implementations vary, most parallel verification systems contain a tightly connected collection of special-purpose logic processors or FPGAs. Due to the distributed nature of these systems, a global system clock is used to coordinate combinational evaluations and to transfer intermediate results throughout the system. Evaluation and communication phases are often delineated by edges of the user clock(s) of the design under test. In general, since multiple system clock cycles are required to perform computation and communication for a single design clock, a fixed relationship must exist between the clocks.

Many contemporary ASIC designs require multiple design clocks that operate asynchronously to each other. While synchronization between a verification system clock and a single design clock can be addressed through linear event ordering, derived relationships for asynchronously occurring events are much more difficult to determine. For parallel verification systems, asynchronous domain limitations occur both during data transport between processors and during data evaluation inside processors. Synchronous data transport often requires that logical signals assigned to the same physical interprocessor wire be driven in the same clock domain. As a result, the transport of a multidomain signal requires that each signal be logically split into constituent single-domain values before interprocessor transport. These single-domain values are then combined at the destination to support multidomain behavior. Causality is an issue in such systems since routing delays can vary across interprocessor paths. System scheduling algorithms must ensure that a regenerated multidomain value is causally consistent with the pretransport value created at the source processor.

When modeling design latches, hold-time constraints can arise if latches are evaluated with gate and data signals which transition on multiple clock domains. For each latch, the validity of the gate must be assured before a data transition is presented, even in the presence of multidomain data and control transitions.

In this paper, we identify a set of scheduling constraints that achieve provable modeling fidelity for designs with multiple asynchronous design clocks. These constraints are integrated into a reverse-ordered computation and communication scheduler to provide causally correct transport of multidomain signals and phase-ordered evaluation of latch data. This automated approach is easier to use than previous manually mapped techniques that isolate multidomain circuits in verification hardware. Additionally, our approach is shown to scale to an unlimited number of asynchronous domains. To validate our approach, new scheduling algorithms have been integrated into the software flow of a commercial FPGA-based logic emulation system from Ikos Systems [8]. When applied to a collection of three large ASIC benchmarks containing multiple asynchronous clock domains, this approach exhibits performance improvement compared to "hard-wired" [4] approaches while maintaining modeling fidelity. A Rent's rule analysis of the approach shows that as FPGA sizes grow, the overhead of our technique is reduced.

Manuscript received March 15, 2002. This paper was recommended by Associate Editor J. H. Kukula.

M. Kudlugi is with the Mentor Emulation Division, Mentor Graphics Corporation, Waltham, MA 02451 USA.

R. Tessier is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA (e-mail: tessier@ecs.umass.edu).

Digital Object Identifier 10.1109/TCAD.2002.804086

0278-0070/02$17.00 © 2002 IEEE

II. Background

A. Related Work

In an effort to enhance functional verification speed, ASIC designers have increasingly turned to parallel verification hardware. Contemporary parallel verification systems include multiprocessor cycle simulators, logic emulators, and rapid prototyping engines. These systems generally contain a collection of processing elements, such as custom logic processors [7] or FPGAs, organized in a fixed topology. Contemporary systems contain up to hundreds of devices packaged on boards in a cardcage. Although prototyping speeds range from a few megaHertz to 20–30 MHz [9], parallel verification systems provide up to five orders of magnitude speedup [1] versus uniprocessor simulation. This speedup has remained roughly constant since increases in ASIC integration due to Moore's law have tracked capacity increases in verification system devices.

Unlike many other forms of parallel processing, the circuit structure of verified designs does not change during execution. As a result, all computation within logic processors and communication between logic processors is predictable at compile time. This compile-time approach has been demonstrated in a number of current and previous emulation systems. Example verification architectures that fit this model include Quickturn CoBalt [14] and Arkos emulators [13] and Ikos VirtuaLogic emulators [8].

Generally, in parallel verification systems both intraprocessor logic evaluation and interprocessor communication is performed in reference to a high-speed system clock. By necessity, the system clock runs at a higher clock rate than the clock(s) of the design under evaluation. The system clock serves as a discrete timebase, providing a reliable mechanism for controlling events at a fine granularity. Generally, it is straightforward to determine a fixed relationship between one emulation clock and a high-speed system clock since multiple cycles of the system clock make up one emulated design clock cycle. It is more difficult to form a fixed relationship between the system clock and multiple design clocks that operate asynchronously to each other since a fixed phase relationship for computation and communication scheduling cannot be easily derived. Previous logic emulation systems have used special compilation and/or manual steps [4], [6] to isolate logic that is evaluated on multiple asynchronous clocks in dedicated system hardware. This approach comes at the expense of performance and mapping flexibility. Not only did these techniques require special system knowledge, but manual intervention can lead to verification errors.

B. Verification Software Flow

Although individual systems may differ in implementation, design mapping similarities exist across many parallel verification systems. A typical system flow for converting a structural or register transfer level (RTL) design to a physical realization appears in Fig. 1. The first step in the mapping process is **design translation**. This step converts the original RTL or structural design to the native technology of the verification system [7]. Typical design translation steps include technology mapping for FPGA-based logic emulation systems and Boolean minimization for parallel simulators that contain sequential logic processors. Following design translation, design **timing analysis** is performed to determine the clock domains or regions of influence of user clocks. After timing analysis, a design is **partitioned** into pieces appropriately sized to meet the physical constraints of each logic processor. These logic partitions are subsequently assigned to specific system processors via **global placement**. Communication between logic processors is determined based on logical **dependency flow** and available interprocessor resources. A **static scheduling** step includes the scheduling of logic evaluation, interprocessor communication, and access to system memory resources. This step is the focus of the algorithm development described in this manuscript. Scheduling is followed by a logic **synthesis** step which creates necessary control and communication logic to implement the static schedule. As a final step, compilation for each logic processor is performed. This step includes FPGA place and route for FPGA-based logic emulators [2] and Boolean instruction scheduling for special-purpose logic processors [13].

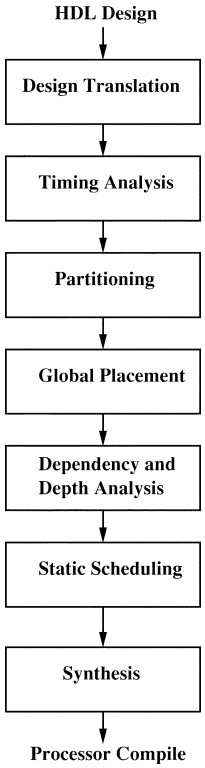


Fig. 1. Verification software compiler flow.

C. The Target System

The target system for this work is an Ikos VirtuaLogic emulation system [8] that contains 384 Xilinx XC4062XL FPGAs. The FPGA interconnect topology of this emulator is primarily a nearest neighbor mesh [9]. The design mapping steps for VirtuaLogic systems follow the basic flow shown in Fig. 1. Compilation steps include design partitioning of logic blocks into FPGAs using K-way mincut, assignment of partitions to specific FPGAs using simulated annealing, and scheduling of

both intra-FPGA logic evaluation and inter-FPGA communication. Communication scheduling is based on Virtual Wires technology, a static list-ordered scheduling technique. The initial step in Virtual Wires scheduling is the determination of all circuit combinational dependencies. Logic can be scheduled for evaluation once all dependent inputs have reached a known value. After intermediate values have been determined, the scheduling approach pipelines multiple logical signals (*virtual wires*) across inter-FPGA wires to overcome FPGA pin limitations [1], [2]. The derived communication schedule determines a feasible space–time route for inter-FPGA connections such that all inter-FPGA dependencies are satisfied.

Both logic evaluation and inter-FPGA signal communication are controlled by a high-speed system clock called a Virtual Clock. This clock serves as a discrete timebase, providing a reliable mechanism for controlling the order of events at a fine granularity. Since many combinational evaluations and signal transfers may occur in a single design clock cycle, the Virtual Clock by necessity runs at a much higher frequency than the design clock. Additional detailed discussion of virtual wires compilation can be found in [2] and [16]. A preliminary approach for scheduling latch evaluation for asynchronous clock domains in logic emulation systems is described in [10]. This previous paper did not consider asynchronous domain inter-FPGA transport. Asynchronous domain memory evaluation for logic emulation is described in [11].

III. Multidomain Problems

There are a number of problems that make multidomain circuits interesting and challenging from a functional modeling point of view. These problems directly affect the scheduling of logic evaluation and inter-FPGA signal communication in parallel verification systems.

A. Timing Closure

Functional Axiom 1 (**Timing Closure**): *Combinational logic plus transmission delay plus setup time between two sequential elements in the same domain takes less than one clock period of the fastest clock attached to either of the sequential elements.*

Consider the circuitry shown in Fig. 2 where two asynchronous clocks CLK1 and CLK2 drive state elements (*FF1, FF3*) and (*FF2, FF4*), respectively. This circuit contains two same domain paths, *FF1.Q–N3–N5–FF3.D* in the domain of CLK1 (domain1) and *FF2.Q–N4–N5–FF4.D* in the domain of CLK2 (domain2). Note that net N5 transitions and is sampled in both clock domains. It is called an MTSD (multitransition and sample domain) net. The correct functional model of this circuit must simultaneously satisfy the timing closure axiom in each constituent domain. This indicates that the data at *FF1.Q* must reach *FF3.D* in exactly one cycle of CLK1 and data at *FF2.Q* must reach *FF4.D* in exactly one cycle of CLK2 irrespective of combinational delays or multidomain net segments in the paths.

B. Transporting Multidomain Values

Functional Axiom 2 (**Causality**): *The occurrence times of combinational logic form a partial order based on causality.*

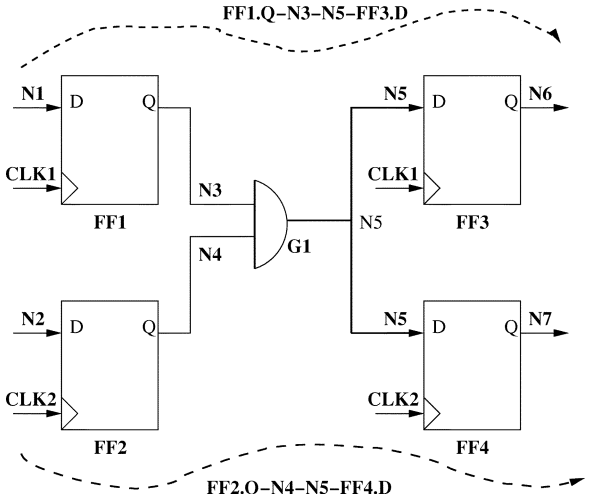


Fig. 2. A multitransition and sample domain (MTSD) example.

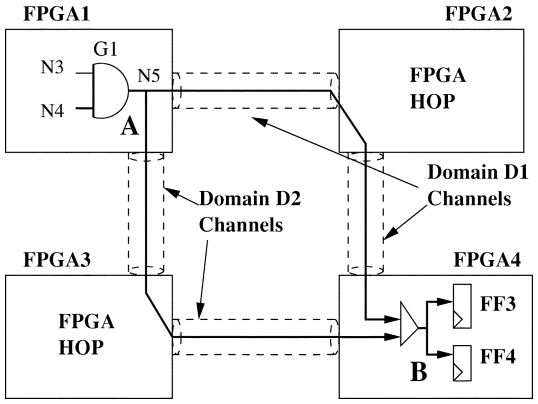


Fig. 3. An example of multitransition and sample domain (MTSD) signal transport.

If part A feeds part B, events on A must have occurred before events on B.

Consider a situation where the circuit in Fig. 2 is partitioned such that the multidomain value N5 must be transported from FPGA1 to FPGA4 as shown in Fig. 3. In a multi-FPGA system, the physical wires that connect FPGAs are grouped into unidirectional channels where each physical wire is capable of carrying multiple signals that belong to the same clock domain, one signal in each Virtual Clock cycle. Pin multiplexing makes it possible to reuse physical wires to support numerous logical wires. To complete signal transport, the communication scheduler determines a path from a source FPGA to a destination FPGA and identifies schedule time slots for the communication to take place. Signal routing may include several intermediate FPGA hops.

A key verification issue involves the transport of multidomain signals such as N5 in Fig. 2 in a system where inter-FPGA communication needs to be synchronous to a system clock (Virtual Clock) over a single-domain physical resource (channel wire). Previous work suggests that such a situation can be avoided by limiting the size of asynchronous-domain logic to one FPGA or by dedicating special inter-FPGA wires to transport the values (hard-wiring) [4]. Since hard-wired signals cannot be multiplexed to carry non-MTSD nets, pin limitation problems [2] can

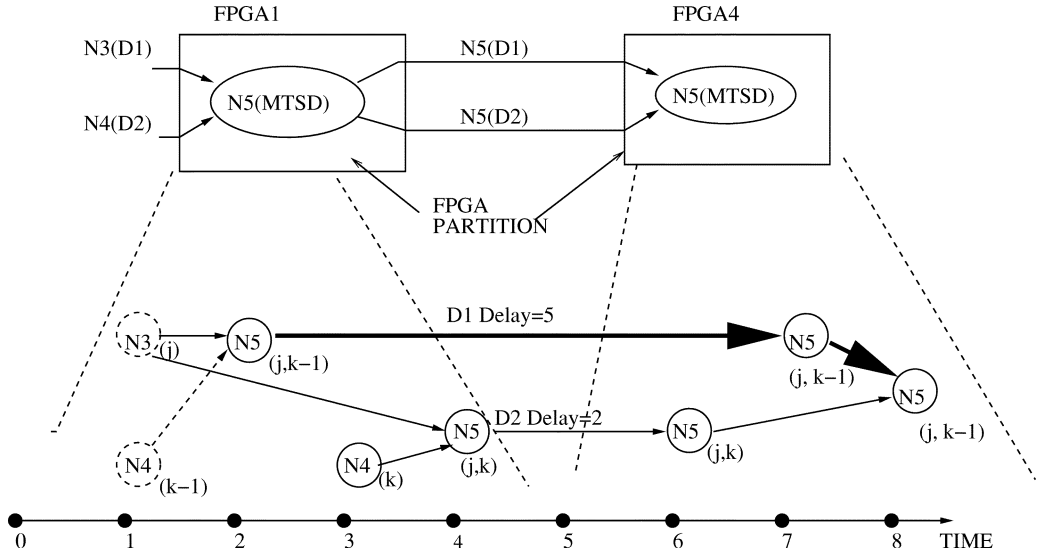


Fig. 4. An example of the multidomain causality problem.

result, leading to reduced system performance. To avoid this problem, it is desirable to split a multidomain value into constituent domain values, route (schedule) the values in respective domains, and recover the multidomain value at the destination FPGA. This solution poses another problem because of unpredictable route timing that is inherent in statically routed systems [2].

As shown in the Fig. 3, communication for each asynchronous clock domain takes place over a different set of inter-FPGA channels. In the case of N5, paths using both domain1 (D1) and domain2 (D2) channels are needed to transport N5 between FPGA1 and FPGA4. The disjoint nature of multiple routing paths for the same logical signal can lead to causality concerns at the destination FPGA. As a result of unpredictable routing delays due to routing congestion, it is possible for the domain1 (D1) value of N5 to start from the source FPGA sooner than the domain2 (D2) value but still arrive after the D2 value reaches its destination. This arrival order can violate the causality principle, resulting in an incorrect result at FPGA4. Fig. 4 illustrates such a case where a D1 version of signal N5 departs from FPGA1 at $t = 2$ while the D2 version departs at $t = 4$, after a new value of N4 has been created. Due to route congestion, the D1 value reaches FPGA4 after the D2 version. Here, notation $N(j)$ indicates the value of signal N due to jth event in the domain to which the signal N belongs. For a multidomain signal, $N(j, k)$ indicates the value of the signal N due to jth event on the first domain and kth event on the second domain. Using combinational rules, when multiple versions of a signal in asynchronous domains are merged at a destination, the most recently arriving version is used in subsequent calculation. As a result, the late arriving, older D1 value at $t = 7$ will be the final value of the signal at FPGA4 and the newer value that arrived at $t = 6$ will be lost. A requirement in transporting multidomain signals is to ensure that causality of events is guaranteed within each of the constituent domains, irrespective of routing delays.

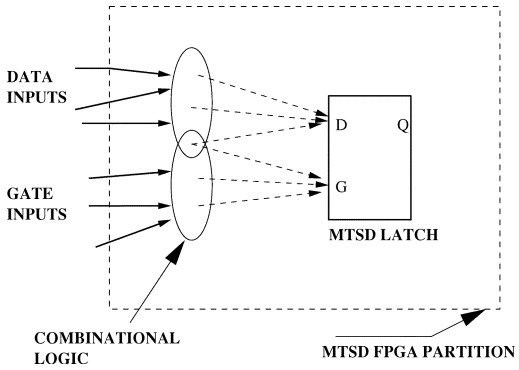


Fig. 5. An example of an MTSD latch.

C. Hold Time Problem in Multidomain Circuits

The correct functioning of state elements requires that data signals arrive at an element a certain period of time (setup time) before the triggering signal and are held steady for a certain period of time (hold time) after the triggering signal arrives. If the triggering signal arrives at a time when the data signal is invalid, a violation occurs and causes incorrect operation of the circuit. Consider a simple latch shown in Fig. 5, which has combinational logic sourcing its Gate and Data inputs. In the waveforms shown in Fig. 6, D, G, and Q represent Data, Gate, and Output waveforms of the latch. Fig. 6(a) shows an ideal zero delay functional modeling of the latch. The edge on user clock CLK at $t = t1$ causes a change in Gate and Data values at the same instant of time and the old value "A" gets stored in the latch as a result. Fig. 6(b) shows more realistic waveforms where routing delays cause the Gate and Data to arrive at the latch inputs at different points in time in response to CLK. A problem arises if the new data "B" reaches the latch sooner than the new gate and overwrites the old value "A," which is not recoverable. This scenario can happen if the routing delay on the Gate path is greater than the routing delay on the Data path due to combinational logic in those paths. This potential delay imposes a constraint (called the **DG constraint**) on latch scheduling for every (Data,

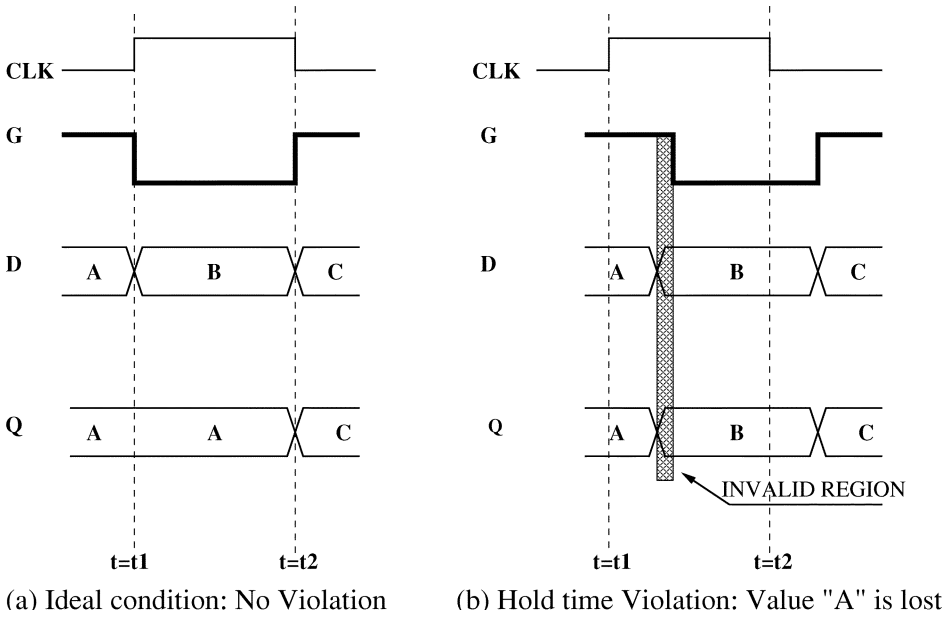


Fig. 6. A hold time violation in an MTSD latch.

Gate) input pair. In a case where both Gate and Data paths are in the same domain, it is easy for a scheduler to compute regions of time when Gate is invalid and mask those regions so that latches are not evaluated. This solution fails if Data and Gate nets are multidomain nets because regions of validity for latch evaluation in one domain may conflict with regions of invalidity in other domains. The key challenge here is to satisfy hold time requirements for every (D, G) pair in each of the constituent domains.

IV. Definitions

The following definitions pertain to multidomain scheduling. These definitions are used throughout the remainder of the manuscript.

Multitransition Net: A net which is combinationally reachable from the output of state elements in distinct clock domains. An MTSD net changes value in response to two or more asynchronous clocks ($|Td| > 1$) where Td is a set of domains in which a net transitions. In Fig. 2, for net N5 $Td = \{domain1, domain2\}$, for net N6 $Td = \{domain1\}$.

Multisample Net: A net which combinationally reaches the D input of state elements in two or more distinct clock domains. A net whose value is sampled in response to multiple asynchronous clocks ($|Sd| > 1$) where Sd is a set of domains in which the net value is sampled. In Fig. 2, for net N5 $Sd = \{domain1, domain2\}$, for N1 $Sd = \{domain1\}$.

MTSD Net: A net which transitions and is sampled in more than one domain. A net is an MTSD net if

$$|Td(n) \cap Sd(n)| > 1$$

where $Td(n)$ is the set of domains in which net n transitions and $Sd(n)$ is the set of domains in which net n is sampled.

MTSD Gate: Any combinational gate whose output is connected to an MTSD net. In Fig. 2, gate G1 is an MTSD gate.

MTSD State: A latch/flip–flop whose gate/clock input is sourced by a multitransition net.

Clock Domain: Encapsulation of user logic that is driven by a design clock or set of design clocks that are phase related. A clock domain demarcates the region of influence of a user clock (or phase-related set) within the design. The number of clock domains is equal to the number of asynchronous (sets of) clocks in the design.

MTSD Domain: Encapsulation of MTSD logic (gates, states, memories, and nets) that are all MTSD with only single-domain nets at the interface.

Domain Channel: A collection of inter-FPGA physical wires that transport a larger collection of logical signals that belong to a given domain. Multiple logical signals are transported within a user cycle along a single physical wire using time division multiplexing [2].

Block: During design compilation the user design is partitioned into chunks of size that are small enough to fit into an FPGA. It is at the block boundary that all the inter-FPGA communication (routing) takes place.

MTSD Block: A block that contains only MTSD gates, nets, and state elements.

RouteLink: An interpartition connection between two block terminals. A RouteLink is different from a net in that it is associated with a particular clock domain and a specific user clock. A multitransition net can result in multiple RouteLinks.

MTSDLinks: A set of RouteLinks for an MTSD net (one for each of its transition domains) that collectively transport the net between two MTSD blocks placed on two FPGAs.

DG Constraint: A latch scheduling constraint imposed on every (**D**ata, **G**ate) pair of same-domain block inputs. This constraint requires that the Gate input must be valid before the Data input. It is necessary to satisfy this constraint to prevent latch Hold time violations during scheduling.

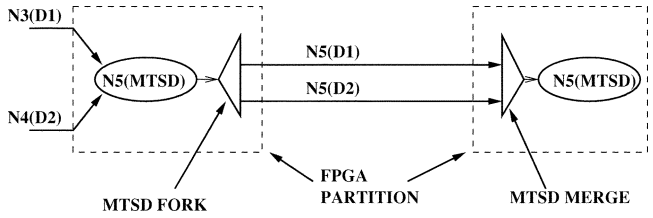


Fig. 7. Multidomain data transport.

V. The Approach

Observation 1: For any relationship $Ri(A, B)$ in a multidomain circuit containing domains A and B, it is sufficient to satisfy $Ri(A)$ and $Ri(B)$ for correct functional verification.

For example, in the circuit shown in Fig. 2, it is only necessary to satisfy the timing closure property for the same domain paths *FF1.Q–N3–N5–FF3.D* and *FF2.Q–N4–N5–FF4.D* but not for cross domain paths *FF1.Q–N3–N5–FF4.D* or *FF2.Q–N4–N5–FF3.D*. Similarly, hold time must be satisfied for each same domain (D, G) pair. With this observation, multidomain problems can be reduced to sets of functional requirements (i.e., timing closure, causality and setup/hold times) within each constituent domain.

A. Multidomain Data Transport

Our approach to transporting a multidomain signal between FPGAs is to decompose the signal into a set of single-domain signals, one for each constituent domain. These component signals can then be freely time-multiplexed with other signals and transported using respective domain channels. Component domain signals may travel independently toward the destination FPGA and may cross multiple intermediate FPGAs. The signals are merged at the destination FPGA to regenerate the multidomain signal such that the causality of events is maintained. As shown in Fig. 7, two new operators are introduced, a FORK operator at the source FPGA and a MERGE operator at the destination, to facilitate the decomposition and causal merging of component signals. From *Observation 1*, flow and dependence relationships along inter-FPGA paths are based on combinational signals from the same domain. This observation allows component signals to be routed in parallel in their respective domains and merged at their final destination. Causal merging can be achieved by dynamically selecting an appropriate single-domain signal at a MERGE point based on the causal order at the corresponding FORK point. One way of ensuring causal correctness is by requiring that the effective *route lengths* along these reconvergent paths are equal. This step is accomplished by first routing all component signals in parallel such that they all have lengths less than or equal to a specific **TargetLength**, the maximum of the shortest achievable route lengths among all paths. This step is followed by the synthesis of delay compensation flip–flops within paths which are shorter than TargetLength to absorb remaining time slack. If routes of length TargetLength cannot be achieved during initial scheduling, TargetLength is incremented by 1 and routing is restarted.

B. Setup/Hold Time Constraints on a MTSD Latch

Observation 2: For a multidomain latch, instantaneous Setup time violations are correctable whereas instantaneous Hold time violations are not.

Due to different routing and combinational delays, latch data and gate values may arrive at different points in time (i.e., Virtual Clock cycles). This necessitates the use of *Observation 2* in determining latch evaluation scheduling. The basis of the observation stems from analyzing two problem situations.

1) A latch is evaluated with a NEW gate value against an OLD data value. This is a setup time violation because NEW data is not ready and stable when the NEW gate value arrives.

2) A latch is evaluated with a NEW data value against an OLD gate value. This is a hold time violation because data is not held stable sufficiently long enough.

When a latch is evaluated with OLD data against a NEW gate that has just changed from closed to open, temporary corruption of the latch may occur. However, the eventual arrival of NEW data results in reevaluation of the latch and restoration of the correct latch value. As a result, a functional setup violation is correctable.

In contrast, when a latch is evaluated with an OLD gate that is open and NEW data, the correct latch value may be irretrievably lost since the OLD latch data is no longer available. As a result, a functional hold time violation is not correctable.

We use notation $V(Ai, Bk)$ to indicate the value of signal V which occurs in response to the ith clock edge of domain A and the kth clock edge of domain B. For any latch with Data $D(Ai, Bk)$ and Gate $G(Aj, Bk)$ on some clock edge k in Domain B, three possible conditions exist with respect to Domain A:

- $(i < j) \Longrightarrow$ instantaneous Setup time violation;
- $(i == j) \Longrightarrow$ both Setup and Hold time satisfied;
- $(i > j) \Longrightarrow$ instantaneous Hold time violation.

Observation 1 implies that every edge k in domain B requires an evaluation of the latch with $D(Ai, Bk)$ against $G(Aj, Bk)$ which satisfies both setup and hold time with respect to B. *Observation 2* implies that when performing such an evaluation, it is legitimate to have $i < j$ or $i == j$ but not legitimate to have $i > j$. The symmetrical relationship holds with respect to evaluations against domain A. This relationship can be extended to an arbitrary number of domains. The implication of these observations is that every edge for any domain results in an evaluation which satisfies both setup and hold time with respect to that domain. Any evaluation not satisfying setup time against a domain is subsequently followed by a correcting evaluation against that domain.

These observations lead to our approach to MTSD latch scheduling, which ensures that hold time violations do not occur and setup time violations do not propagate. Our new static scheduling algorithm meets these constraints by ensuring that latch Data inputs do not arrive before associated Gate values for any edge in any domain and that both Data and Gate changes, triggered off a specific clock edge, arrive prior to subsequent clock edges from that domain.

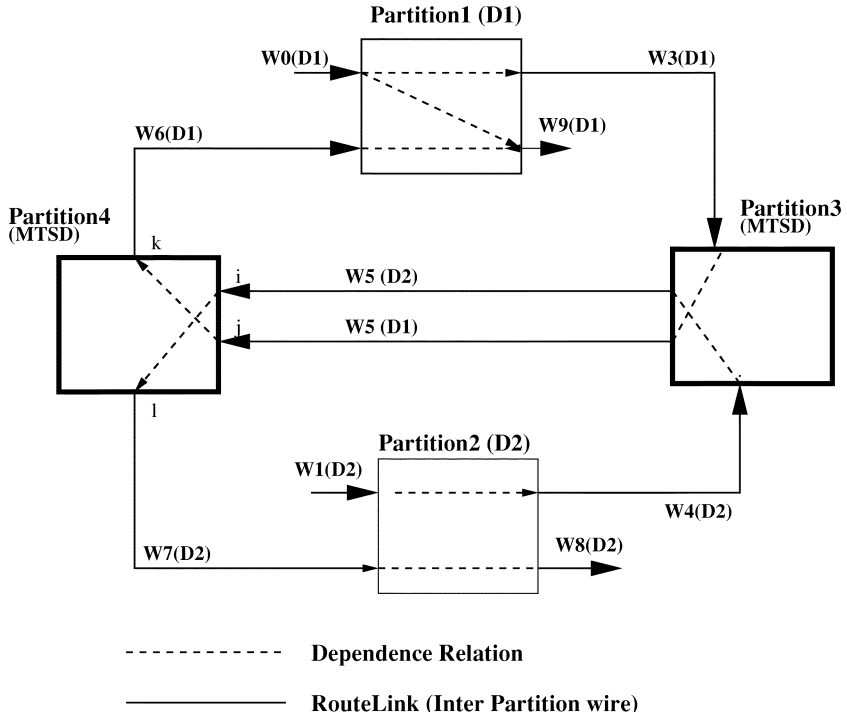


Fig. 8. MTSD dependency and depth computation.

C. Transforming MTSD Flip–Flops

MTSD flip–flops are not covered by *Observation 2* and are more difficult to address than MTSD latches. Our approach handles MTSD flip–flops by transforming them into master–slave latch pairs and then subjecting them to the same processing as other MTSD latches.

VI. Dependency and Depth Computation

Two important timing analysis computations, dependency and depth, are performed to aid the scheduling process. The purpose of these computations are two-fold: first, to ensure that the logical data dependency along multi-FPGA combinational paths is correctly implemented in the schedule and second, to prioritize routing of critical paths based on the depth of logic [2]. This depth information is also used to determine the execution order of sequential components (such as memories and latches) inside blocks that contain combinational flow paths.

A. Hold Time Problem in Multidomains

A RouteLink, as defined in Section IV, is used to represent a logical connection between block terminals that are placed on two different FPGAs. Two RouteLinks are linked by a parent–child relationship if they are combinationally connected. In addition, each RouteLink is assigned an integral number, depth, based on the depth of its parent RouteLinks and the estimated distance between its source and destination FPGAs. As a result, the inter-FPGA communication of the entire design can be represented as a partial order of RouteLinks sorted by depth. Routing according to this partial order automatically satisfies the dependency flow and ensures that critical paths are routed first. Fig. 8 shows an example design with four partitions. Partition1 contains a block with strictly domain D1 logic, Partition2 contains a block with strictly domain D2 logic, and Partition3 and Partition4 contain MTSD blocks. The partitions are interconnected with a set of RouteLinks labeled W0 through W9. Each link is associated with a specific domain of transition. MTSD nets are split into component domain signals. For example, wires W5(D1) and W5(D2) carry D1 and D2 versions of MTSD wire W5 from Partition3 to Partition4.

The transport of a given MTSD signal between FORK and MERGE operators is split into a group of RouteLinks which collectively transport the MTSD value across FPGAs. As mentioned in Section V-A, our scheduling approach ensures causality along reconvergent MTSD paths by requiring that the routing length of each domain path is the same. To support causal transport, it is necessary to schedule all related MTSD RouteLinks together so that they all have equal route length. Previously [2], [16], dependency and depth information was determined only along same-domain paths since these paths are strictly governed by the synchronous clock edges in a given domain. This analysis is extended to include cross-domain paths so that a partial order of RouteLinks is created that is not only logically consistent in any single domain, but also is causally correct along reconvergent multidomain paths. Causality is achieved by sorting all RouteLinks by depth that is normalized over all constituent domains.

To support MTSD scheduling, two types of dependency are computed. Same-domain dependency tracks link dependencies within a single domain and MtsdDependency tracks link dependencies across all domains including cross-domain paths.

Dependency determination is a multistep process. Initially, terminal-to-terminal dependencies are determined for each block. For each block input terminal i, the following are calculated:

- Child $[i]$ = *set of block outputs in the same domain that can be combinationally reached from block input* i;
- $Mtsd$Child $[i]$ = *set of all block outputs that can be combinationally reached from block input* i.

Child$[i] = \emptyset$ if the terminal is terminated exclusively by one or more state elements. Similarly, Parent$[j]$ and $Mtsd$Parent$[j]$ sets are determined to express inverse relationships. For example, in Partition4, which contains an MTSD block, the following domain relationships hold:

- Child$[i] = \{l\}$;
- $Mtsd$Child$[i] = \{k, l\}$;
- Parent$[l] = \{i\}$;
- $Mtsd$Parent$[l] = \{i, j\}$.

Following the evaluation of parent and child relationships, the same-domain and MTSD depths of each interpartition wire are determined. These depths are computed in reverse order (this is an implementation choice; forward evaluation could be performed as well) recursively from the RouteLink dependent sets such that for each RouteLink i

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

Similarly $Mtsd$Depth is expressed as

$$Mtsd\text{Depth}[i] = \begin{cases} 0 & \text{if } Mtsd\text{Child}[i] = \emptyset \\ 1 + \max\limits_{j \in Mtsd\text{Child}[i]} & \\ \cdot MtsdDepth[j] & \text{otherwise.} \end{cases} \quad (2)$$

Consider depth evaluation for the partitioned circuit shown in Fig. 8. In this example, it is known that initially the same-domain and $Mtsd$Depths of wire W8 = 4 and W9 = 1 due to downstream paths that are not shown in the figure. From these initial conditions, the remaining depths can be determined.

In Table I, related MTSD links W5(D1) and W5(D2) have different same-domain depths but equal $MtsdDepths$. During routing, same-domain paths are scheduled independently to promote optimal scheduling. As shown in Fig. 9, $Mtsd$Depth is used to sort all RouteLinks in all domains to produce a partial order that is consistent across all domains.

B. MTSD Latch Signal Evaluation Ordering

As noted in Section V-B, in order to satisfy the hold time (DG Constraint) of an MTSD latch, signals must be scheduled such that Gate signal information from a given domain arrives at the latch at or before the time the same-domain Data value arrives. This imposes an ordering requirement between the set of RouteLinks fanning into the D-input of the latch and the set of RouteLinks fanning into the G-input of the latch. The same RouteLink can reach multiple latch inputs imposing additional ordering requirements between latches. The following describes the scheme used to compute the evaluation order of RouteLinks and latches.

To aid in latch ordering, each MTSD partition is analyzed to create RouteLink sets for each latch. Example dependencies are shown in Fig. 10.

TABLE I
SAME-DOMAIN DEPTHS AND $MTSDDEPTHS$ FOR ROUTELINKS IN FIG. 8

RouteLink	Depth	MtsdDepth
W9	1	1
W6	2	2
W8	4	4
W7	5	5
W5(D1)	3	6
W5(D2)	6	6
W4	7	7
W3	4	7
W1	8	8
W0	5	8

• **D-INPUT Set**: Group of all RouteLinks that reach the Data terminal of the latch via combinational logic. This includes any link that reaches both Data and Gate (called DG input).

• **G-INPUT Set**: Group of all RouteLinks that reach the Gate terminal of the latch via combinational logic. None of these inputs also fan out to the latch Data input.

• **D-OUTPUT Set**: Group of all RouteLinks which are dependent children of links in the D-INPUT set or the latch output.

Given these RouteLink sets, a temporal dependency is added between same-domain D-INPUT and G-INPUT RouteLinks. An additional RouteLink is created for each latch to represent latch dependencies in the overall dependency flow. Fig. 11 shows the ordering requirements imposed on RouteLinks due to this temporal dependency. These requirements order the RouteLinks from least constrained to most constrained. If the RouteLinks at the block terminals are scheduled in dependency order, the DG constraints can be easily satisfied at the latch input terminals. Note that the above approach cannot handle multipartition cyclic dependencies from D-OUTPUT links to G-INPUT links. Currently, this limitation is avoided by encapsulating such cycles in a single combinational module and placing it in the FPGA containing the latch.

C. Latch Groups

A block terminal can combinationally reach multiple latch inputs, necessitating ordering requirements between latches. Each of these latches can in turn impose conflicting ordering requirements for the input RouteLink at the block terminal. It is therefore necessary to determine an evaluation order that satisfies latch ordering constraints for latches that share common RouteLinks. As shown in Fig. 12, this limitation imposes additional constraints on the order in which RouteLinks are scheduled.

DD-Type: Consider the case shown in Fig. 12(A) where a block terminal Pi combinationally reaches Data inputs of two or more latches. In order to satisfy DG constraints, RouteLinks in the G-INPUT sets of both latches must be ordered before the RouteLink for this block terminal. This ordering is achieved by combining these latches into a latch group and merging their respective D-INPUT and D-OUTPUT sets so that the latches can be evaluated together as a group. The individual latch RouteLinks are replaced by a single RouteLink to represent the group in the dependency flow graph.

DG-Type: If a block terminal Pi reaches the Data input of latch L1 and the Gate input of latch L2, as shown in

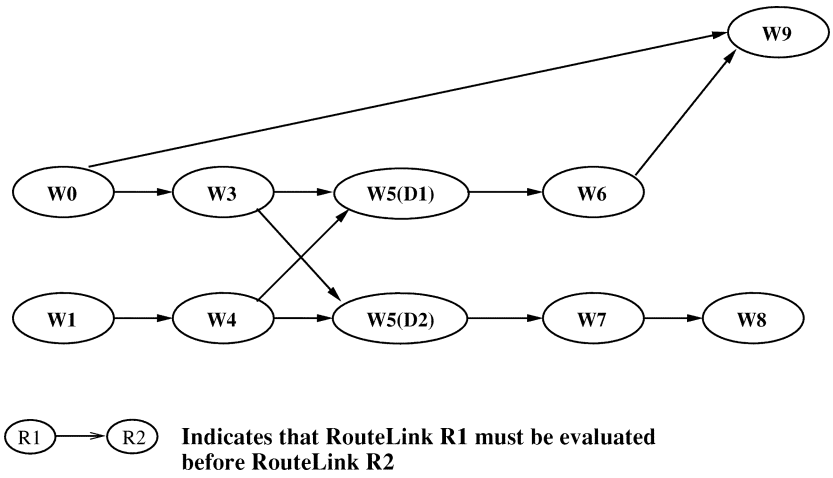


Fig. 9. A partial order of RouteLinks sorted by $MtsdDepth$.

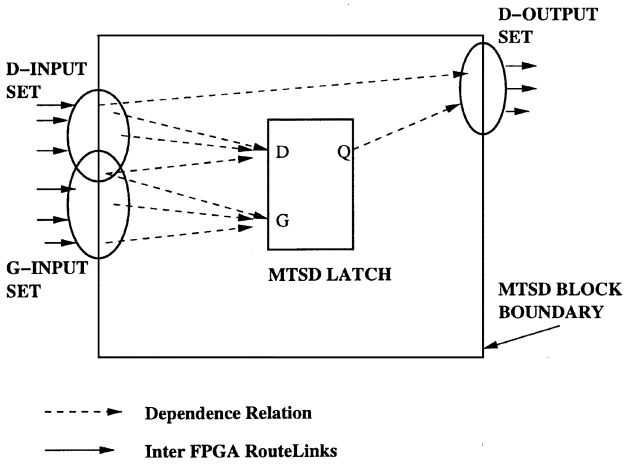


Fig. 10. An example of MTSD latch signal dependencies.

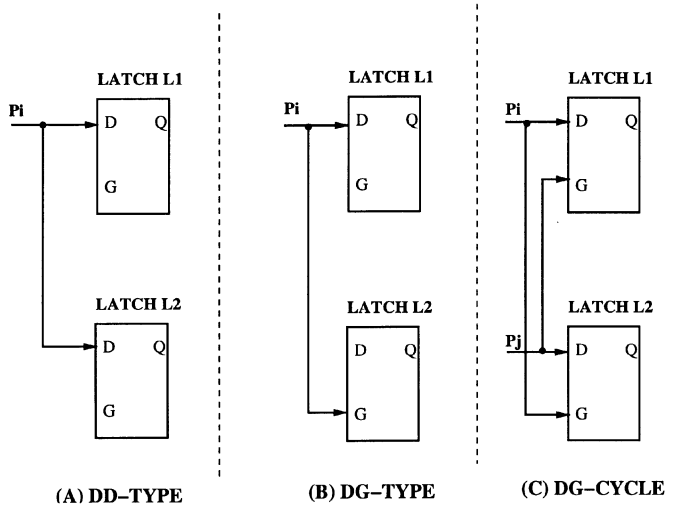


Fig. 12. MTSD latch relationships.

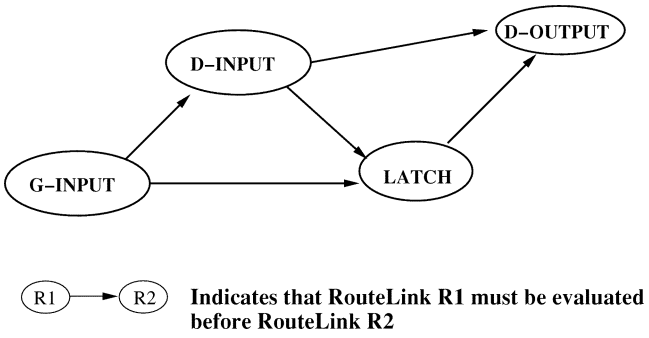


Fig. 11. A partial order of RouteLinks sorted based on DG constraints.

Fig. 12(B), then latch L2 must be evaluated before latch L1, since Gate terminals must be evaluated before Data terminals. This restriction forms a Parent–Child dependency relationship between L2 (parent) and L1 (child). During scheduling, two arrays, DGChild(L) and DGParent(L), are used to maintain the list of latches which must be evaluated before and after latch L, respectively.

DG Cycle: Consider a case where a block terminal Pi reaches the Data input of latch L1 and the Gate input of latch L2, as shown in Fig. 12(C). Additionally, block terminal Pj reaches the Gate terminal of latch L1 and the Data terminal of latch L2. These relationships lead to conflicting ordering requirements since latch L1 requires Pj to be ordered before Pi and latch L2 requires just the opposite. Whenever there is a cyclical DG relation involving two or more latches, the only way to satisfy DG constraints on all latches is to evaluate all of them at the same point in time, similar to DD-type latches.

D. Computing MTSD Latch Dependency

The DG constraint implies that: D-INPUT terminals must be evaluated after all of the dependent G-INPUT terminals are evaluated in each domain, but before the latch itself is evaluated. This constraint must hold valid in each of the same domain (Di, Gj) pairs for Di in the D-INPUT set and Gj in the G-INPUT set. In addition to dependency due to combinational logic, two types of latch dependencies are introduced into the system.

- Dependency introduced between terminals in the D-INPUT set and terminals in the G-INPUT set.
- Dependency introduced between latches/latch groups due to DG relationships.

These dependencies are used to order latch RouteLinks with other RouteLinks that are connected at the block boundary.

VII. Static Scheduling

A list scheduling algorithm is used to route communication paths between blocks. This is a reverse scheduling algorithm which routes paths starting from primary outputs to primary inputs. The described techniques are also applicable to forward scheduling which routes from primary inputs to primary outputs. In this section, the basic steps involved in static routing are described. In the next section, the specific steps involved in MTSD latch scheduling are described.

Combinational dependency analysis is performed on the placed but unrouted design to support the creation of an ordered set of RouteLinks for routing. At the end of the dependency and depth computation step, a list of RouteLinks that collectively represents the communication between all FPGA partitions is produced. RouteLink depths represent the longest (worst case) time required to propagate through the network from the source FPGA to the destination FPGA. The reverse scheduler starts from RouteLinks terminating at primary outputs or internal state elements and schedules one link in the partial order at a time progressively moving toward the primary inputs.

The goal of the algorithm is to compute the latest time, called the **ReadyTime**, at which a value must arrive at a given block or latch terminal for further evaluation and/or propagation. This time commitment flows in a reverse fashion from a child RouteLink to all its parent RouteLinks which in turn compute and propagate ReadyTime commitments to their parents. *All delays and ReadyTimes are represented in terms of integral time units with a Virtual Clock cycle as the basic unit.*

A. Algorithm 1: Static Routing

The core scheduling algorithm involves the following steps.

1) Sort RouteLinks in descending order based on MtsdDepths. The scheduling of RouteLinks that are sorted by MtsdDepth ensures that all dependent RouteLinks in all domains are scheduled before descendant RouteLinks.

2) Initialize ReadyTime for each RouteLink that terminates at a primary output or internal state element with the combinational/propagational delay involved in moving a value from a block terminal to its final destination (output or state). Initialize ReadyTimes of all other RouteLinks to 0.

3) For each RouteLink(Pi, Pj) in the sorted list:

a) Find the shortest path "sp" from Pi to Pj such that data arrives by ReadyTime(Pj) using a modified Dijkstra's algorithm [3]. ReadyTime(Pj) is either initialized in Step 2) or computed for a previous RouteLink in Step 3e).

b) Check if routing resources are available in the domain channels along the path for the given time slot(s). If routing resources are not available, increment ReadyTime(Pj) and go to Step 3a). This finding indicates that the final value will arrive at Pj sooner than needed. This is not a problem, in general, but has an efficiency impact on MTSD routing, described subsequently in Algorithm 2.

c) Reserve wiring resources along the path sp. This reservation involves selecting a set of physical wire and time slot pairs along the path segments from the destination FPGA to the source FPGA.

d) Compute DepartureTime(Pi) at the source Pi, $DepartureTime(Pi) = ReadyTime(Pj) + PathLength(sp)$

e) Update input ReadyTimes at the block. For each terminal Pk in Parent(Pi)

$$ReadyTime(Pk) = \mathrm{MAX}\left(DepartureTime(Pi) + Delay(Pk\ to\ Pi)\right).$$

B. Routing MTSD Data Paths

As shown in Fig. 4, MTSD paths are given special treatment; they are split into a group of RouteLinks that belong to different domains. These links collectively transport single-domain versions of the MTSD value across FPGAs. If the route scheduler can schedule these RouteLinks such that they all take an equal number of Virtual Clocks to propagate, the causally correct value can be obtained at the destination.

The group of all related RouteLinks of an MTSD net n is referred to as MTSDLinks(n). All RouteLinks in MTSDLinks(n) must be scheduled at the same time since scheduling one path may affect another. It is necessary to determine a schedule such that all RouteLinks have the same effective length. The longest domain path from a source FPGA to the destination FPGA determines the minimal route length. As described in Section V-A, this length is the $\mathrm{TargetLength}$.

In determining routing for each MTSD net, four variables are considered.

- $DT\mathrm{required}$ (RequiredDepartureTime), the time at which a signal must depart a source block terminal to satisfy TargetLength requirements.
- $DT\mathrm{actual}$, the time at which a signal departs the source block.
- $AT\mathrm{required}$ (RequiredArrivalTime), the time at which a signal must arrive at a destination block to satisfy the TargetLength requirement. This value is the ReadyTime computed during the scheduling of dependent RouteLinks, as explained in Algorithm 1.
- $AT\mathrm{actual}$ (ActualArrivalTime), the time at which a signal is scheduled to arrive at the input of the destination block.

C. Algorithm 2: MTSDLinks Routing

This algorithm is a specialization of Step 3a) of Algorithm 1. The basic algorithm is as follows.

1) Compute $\mathrm{TargetLength}$ for MTSD RouteLinks. Even if source and destination FPGAs for an MTSD net are the same, the path lengths of different domain paths may vary based on channel domain designation

$$\mathrm{TargetLength} = \max_{Ri \in \mathrm{MTSDLinks}(n)} \left(\min\left(\mathrm{Distance}(Ri)\right)\right).$$

2) For each RouteLink in MTSDLinks compute DTrequired, the time by which a signal must leave a source block

$$DT\mathrm{required}(Ri) = AT\mathrm{required} + \mathrm{TargetLength}.$$

3) For each RouteLink in MTSDLinks find a schedule using Dijkstra's algorithm such that the value arrives at the destination

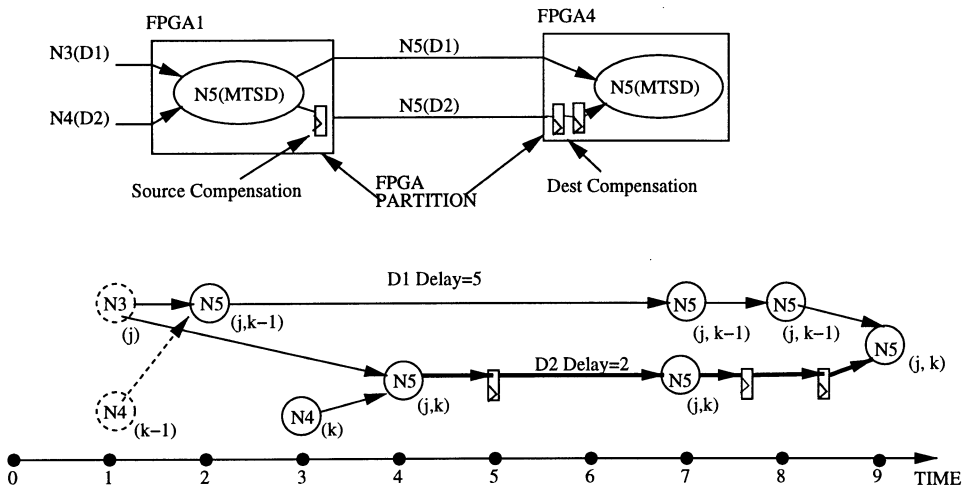


Fig. 13. Delay compensation to achieve a causally correct merge at the route destination.

at or before ATrequired and has a length less than or equal to the $\mathrm{TargetLength}$.

4) If Step 3) is successful for all RouteLinks in MTSDLinks, go to Step 3a) of Algorithm 1.

5) If Step 3) failed because a feasible schedule cannot be found, then:

a) delete all the schedules for links already routed.
b) increment $\mathrm{TargetLength}$ by 1.
c) go to Step 3).

D. Delay Compensation Synthesis

At the completion of Algorithm 2, it is known that all RouteLinks of an MTSD net have been routed successfully. However, each RouteLink in MTSDLinks(n) may require a different number of Virtual Clocks to traverse source-destination paths. In order to equalize the delay for all RouteLinks of MTSDLinks(n), it is necessary to add extra Virtual Clocks to all paths that initially require fewer Virtual Clocks than the longest RouteLink path. This can be accomplished by adding delay compensating flip–flops under one of two scenarios.

- Source Compensation: Virtual Clock triggered flip–flops can be added in the source FPGA at the FPGA boundary such that

$$\mathrm{SourceCompensation} = DT\mathrm{required} - DT\mathrm{actual}.$$

This insertion prevents a signal from being sampled in any domain before it is ready.

- Destination Compensation: Virtual Clock triggered flip–flops can be added in the destination FPGA at the FPGA input boundary such that

$$\mathrm{DestCompensation} = AT\mathrm{actual} - AT\mathrm{required}.$$

This approach ensures that the domain data reaches the destination in a causally correct fashion.

Delay compensation is implemented by synthesizing Virtual Clock triggered flip–flops in each single-domain path as shown in Fig. 13 for the example in Fig. 4. The lower half of the figure shows one possible way to correctly transport multidomain net N5 to the destination so that causality is preserved. Note that three flip–flops are inserted in the D2 path to compensate for the difference in D1 versus D2 routing delay (5 versus 2).

E. Scheduling MTSD Latches

MTSD paths must be scheduled such that hold time requirements are satisfied on every MTSD latch in each of the constituent domains.

Fig. 14 illustrates the basic steps involved in latch scheduling. The goal of this algorithm is to schedule each latch such that Hold time and Setup time constraints are satisfied in each constituent domain. Our solution to the hold time problem is to schedule D-INPUT RouteLinks *before* (a result of reverse routing) G-INPUT RouteLinks but *after* the latch itself is scheduled. The setup time is ensured by scheduling D-OUTPUTs before the latch is scheduled so that only the final correct value for the latch is propagated to its fanouts. Due to the latch order described in Section VI-B, by the time a latch is evaluated, the DepartureTimes of all the terminals in the D-OUTPUT set are known. In Fig. 14, arrows indicate the flow of ReadyTime, the time at which a value must be ready for consumption by dependent logic. The ReadyTime evaluation sequence is indicated by the numbers in the parentheses. This algorithm computes the final ReadyTime on D-INPUTs and the lower bound for the ReadyTimes on G-INPUTs.

$\mathrm{MinDelay}(i, L)$ and $\mathrm{MaxDelay}(i, L)$ are determined prior to scheduling for each block input terminal i to latch L. This determination requires intrablock path analysis since there can be multiple combinational paths from a block input terminal to a latch.

F. Algorithm 3: Latch Scheduling

The Latch Scheduling algorithm below is a specialized version of Step 3a) of Algorithm 1. The basic algorithm that is invoked for each latch RouteLink, L, involves the following steps.

1) Compute the initial ReadyTimes for each terminal Di in the D-INPUT(L) set based on the DepartureTimes of their fanouts (D-OUTPUTs). These values are not final ReadyTimes because they do not take into account the latch's $\mathrm{ReadyTime}$. This step is same as Step 3e) in Algorithm 1, but repeated here for sake of completeness.

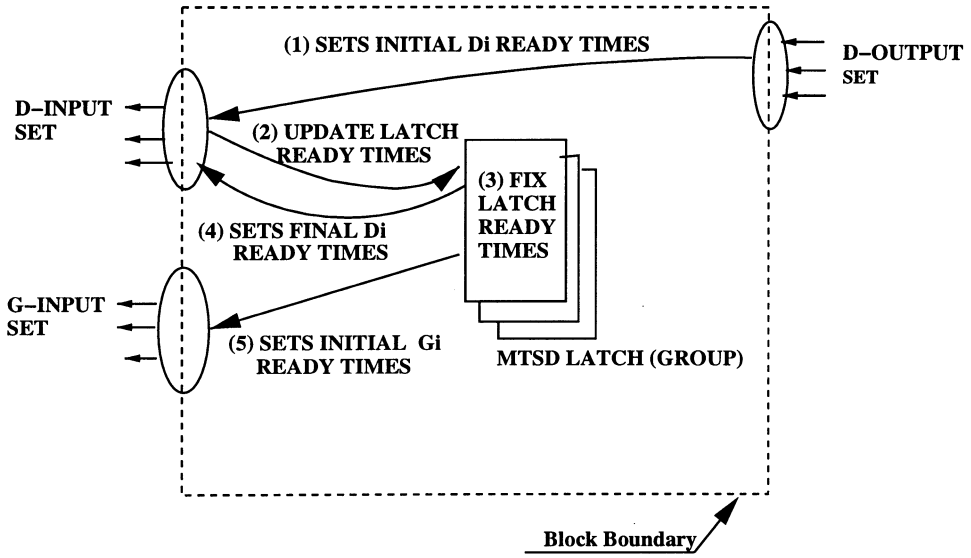


Fig. 14. An example of MTSD latch evaluation.

2) Evaluate the difference between each $\mathrm{ReadyTime}(Di)$ and $\mathrm{ReadyTime}(L)$. If the difference is less than the minimum delay from Di to the latch L, update $\mathrm{ReadyTime}(L)$.

For each Di in D-INPUT(L)

$$\mathrm{ReadyTime(L)} = \mathrm{MAX}\left(\mathrm{ReadyTime}(Di) - \mathrm{MinDelay}(Di \text{ to } L)\right).$$

3) If latch L has a DG relationship on other latches, take the maximum ReadyTime of the child latches.

For each child latch Lc in DGChild(L)

$$\mathrm{ReadyTime(L)} = \mathrm{MAX}\left(\mathrm{ReadyTime}(L), \mathrm{ReadyTime}(Lc)\right).$$

4) For each Di in D-INPUT(L):

a) Compute the RequiredReadyTime. The value is called **required** ReadyTime because if data arrives any sooner than this time, there is a risk of violating the DG Constraint

$$\mathrm{RequiredReadyTime(Di)} = \mathrm{ReadyTime}(L) + \mathrm{MinDelay}(Di \text{ to } L).$$

b) Compute the final ReadyTime. This is the time that is used by parent links of Di for further computation

$$\mathrm{ReadyTime(Di)} = \mathrm{MAX}(\mathrm{ReadyTime}(Di), \mathrm{RequiredReadyTime}(Di)).$$

c) If $\mathrm{ReadyTime}(Di)$ is greater than $\mathrm{RequiredReadyTime}(Di)$, add delay compensation in the Di to L path to ensure that Data does not arrive at the latch sooner than required [as mentioned in Step 3b) of Algorithm 1]. A delay equal to $DelayCompensation(Di, L)$ is injected into the path from Di to latch L by adding a chain of Virtual Clock triggered flip–flops

$$\mathrm{DelayCompensation(Di, L)} = \mathrm{ReadyTime}(Di) - \mathrm{RequiredReadyTime}(Di).$$

5) Propagate $\mathrm{ReadyTime}(L)$ to each of the terminals in G-INPUT(L) as initial $\mathrm{ReadyTime}(Gi)$. This is the *initial ReadyTime* because there could be other dependent children on Gi which can further alter its ReadyTime.

For each Gi in G-INPUT(L)

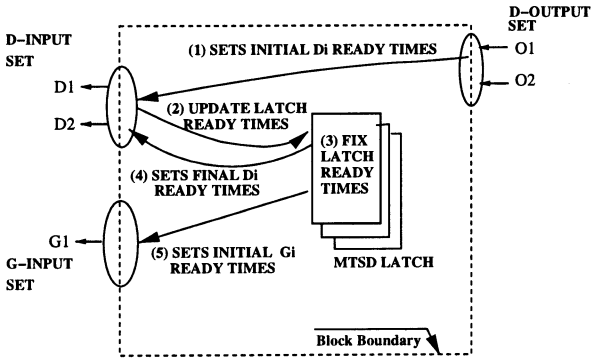
$$\mathrm{ReadyTime(Gi)} = \mathrm{MAX}(\mathrm{MaxDelay(Gi \text{ to } L)} + \mathrm{ReadyTime}(Gi, \mathrm{ReadyTime}(L))).$$

The above algorithm guarantees that the ReadyTimes at the Gate input terminals are always greater than or equal to the ReadyTimes at the Data input terminals for every same domain (Data, Gate) terminal pair of any latch. This relationship ensures that the Gate value always arrives before the Data value and that the DG constraints are satisfied at the latch. In the above equations, MinDelay values for paths from Data terminals to latches have been used rather than the MaxDelay values used for Gate terminal to latch paths. This relationship ensures that the delay from any Gi to a latch does not exceed the delay from Di to the latch after compensation (performed in Step 4c). Without this compensation it is still possible to violate the hold time at the latch even if DG constraints at the block boundary are met. Fig. 15 illustrates Algorithm 3 with a simple example.

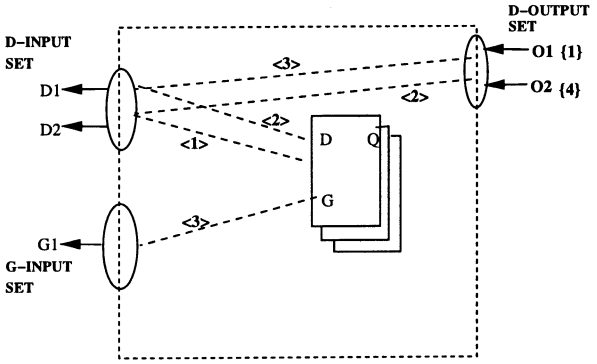
VIII. Results

A. Experimental Results

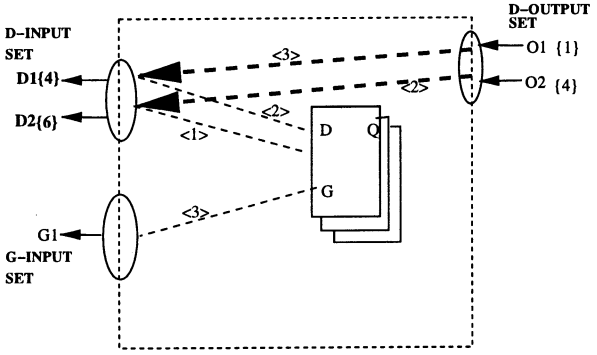
The algorithms described in this paper have been implemented and integrated into the Ikos VirtuaLogic Compiler [8] for the VStation5M Emulator. Three industrial designs (a telecom design and two graphics processors) containing asynchronous domains have been compiled using the VirtuaLogic compiler. As shown in Table II, Design2 has the largest percentage of MTSD logic. Each of the designs has multiple user clocks that are phase-locked to each other within each domain. Table III compares the results of scheduled MTSD Virtual routing to hard-wired routing. Virtual routed wires and pins are multiplexed to achieve better FPGA pin utilization while hard-routed wires require dedicated physical wires and



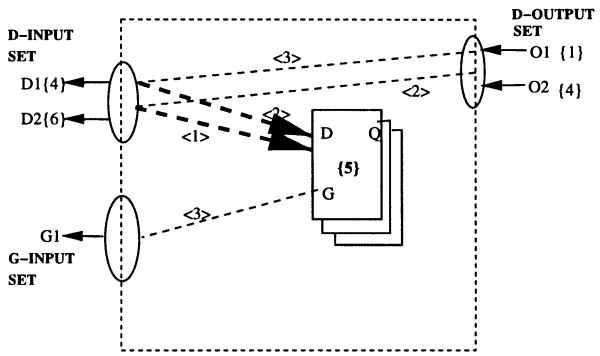
(1) MTSD Latch Scheduling Algorithm.
Arrows indicate the direction of ReadyTime Flow



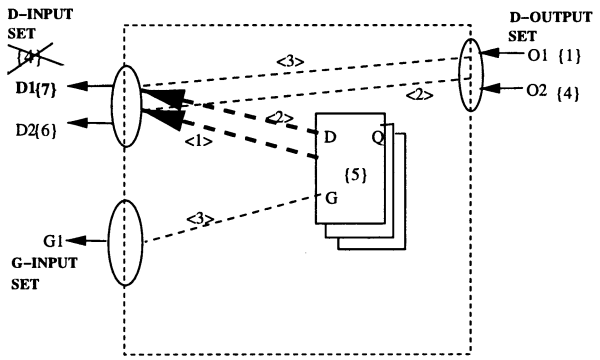
(2) Given: Combinational Path Lengths <x> and
Output Terminal ReadyTimes {y}



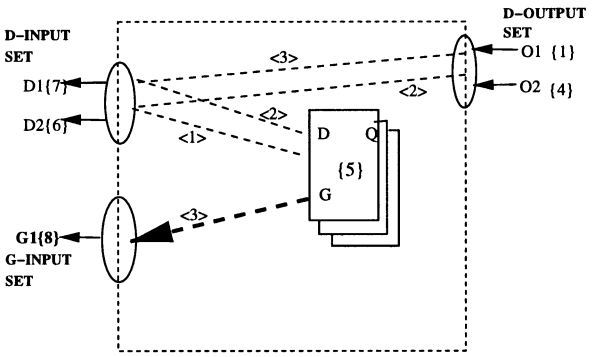
(3) STEP1: Set initial Di ReadyTimes. Note that
ReadyTime flows from right to left (reverse scheduling).
Initial ReadyTime(D1) = ReadyTime(O1){1} + Delay<3> ={4}



(4) STEP2,3: Compute Latch ReadyTimes.
ReadyTime(Latch)= Max{(4–2), (6–1)}={5}



(5) STEP4: Set FINAL Di ReadyTimes.
Final ReadyTime(D1) = ReadyTime(Latch){5}+Delay<2> = {7}



(6) STEP 5: Set initial G ReadyTimes.
Initial ReadyTime(G1) = ReadyTime(Latch){5}+Delay<3> = {8}

Fig. 15. An example of the Latch Scheduling Algorithm.

TABLE II
MTSD DESIGN STATISTICS

Testcase	Design1	Design2	Design3
Num. primitive Gates	1687000	101000	958000
Num. MTSD Gates	4100	10000	3400
Num. Clock Domains	3	2	3
Clock Domains	d1 d2 d3	d1 d2	d1 d2 d3
Num. User Clocks	2 4 6	1 6	1 2 4
Num. MTSD Flipflops	219	220	304
Num. MTSD Latches	120	320	10
Application field	Graphics	Telecom	Graphics

pins. To determine the results for hard-routing experiments a prerouting step was performed to reserve physical pins between source and destination FPGAs for each MTSD wire. These pins were removed from consideration during subsequent virtual routing of non-MTSD wires. From Table III it can be seen that the number of Virtual Clocks in the critical path for Design2 is much higher than the other two designs. This is because experiments for Design2 were dominated by transactions with emulation memory. MTSD routing results in a smaller number of Virtual Clocks (leading to faster overall execution) as compared to the hard-wired approach. This observation results from some physical wires being removed for hard-wiring and the remaining wires carrying a greater load of non-MTSD communication. Maximum emulation clock speeds in the last

TABLE III
COMPILER RESULTS: VIRTUAL ROUTING VERSUS HARD ROUTING

	Design1	Design2	Design3
Num. MTSD Paths	173	213	100
Num. MTSD FPGAs	23	24	8
Num. FPGA LUTS	1500	2400	1359
Clock Domains	d1 d2 d3	d1 d2	d1 d2 d3
Num. Non MTSD FPGAs	11 43 180	4 7	10 6 85
Critical Path (Virtual Clocks) Hard Routing	42 47 49	85 131	16 47 39
Critical Path (Virtual Clocks) Virt. Routing	37 38 46	68 108	20 40 31
Approx. Max Speed MTSD Hard Routing	346 KHz	129 KHz	354 KHz
Approx. Max Speed MTSD Virt. Routing	369 KHz	157 KHz	410 KHz

two rows of Table III were determined based on a 34-MHz Virtual Clock on a VStation-5M Emulator.

B. Theoretical Results

In this section we derive theoretical gate utilization for MTSD logic emulation with and without virtual wires and show that emulation with virtual wires scales with increasing FPGA device size. It was shown previously [2] that isolating logic (such as MTSD partitions) in specific FPGAs and *hard-wiring* them to other system FPGAs will lead to decreasing FPGA utilization as FPGA gate capacities scale. In this analysis, we consider FPGA gate utilization for logic emulation when MTSD and single-domain logic is distributed *evenly* across all FPGAs. Two distinct cases are considered for this even distribution: MTSD signals are hard-wired for communication between FPGAs and virtual wires are used for communication between FPGAs. In both cases, virtual wires are used to communicate non-MTSD single-domain signals.

1) Rent's Rule: To complete this analysis we use a formulation that is similar to the one provided in [2]. The basic relationship between an amount of logic and required I/O for the logic can be characterized by the Rent's rule [12] equation as follows:

$$\textbf{Rent's Rule:} \qquad P = KG^B \tag{3}$$

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.

To determine FPGA utilization it is necessary to determine the fraction of design gates G_c implemented in the FPGA versus total available per-FPGA gates, G_f. Design gates G_c can be separated into two categories, MTSD and single-domain design gates (G_{mtsd} and G_{sd}, respectively), such that $G_c = G_{\mathrm{mtsd}} + G_{sd}$.

2) Hard-Wired MTSD Logic: In the first case that is considered, MTSD and single-domain logic is partitioned evenly throughout all system FPGAs. Inter-FPGA MTSD nets are hard-wired to inter-FPGA pin and wire resources such that each logical signal is assigned to dedicated pin and wire resources. From

TABLE IV
PARAMETERS FOR SCALABILITY COMPARISON

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

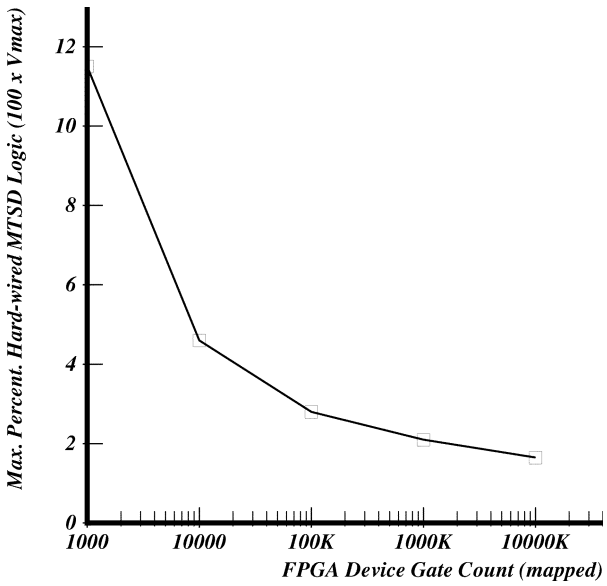


Fig. 16. Maximum percentage of MTSD gates ($100 \times v_{\max}$) per partition—MTSD hard wires.

(3), the number of FPGA pins required per FPGA for the MTSD portion of each partition is

$$P_{\mathrm{mtsd}} = dK_c(G_{\mathrm{mtsd}})^{B_c} \tag{4}$$

where d indicates the average distance, in terms of FPGA boundary crossings, for each wire [2], and K_c and B_c are design dependent. For an FPGA to contain single-domain logic ($G_{sd} > 0$) with signals that are routed with virtual wires, at least one FPGA pin must be available that is not used for P_{mtsd}.

The gate cost of per-FPGA virtual wires *overhead* (the logic used to multiplex pins) for per-FPGA single domain logic (G_{sd}) is $V_0 + V_1 \times dP_{sd}$ where V_0 is the per-FPGA cost associated with control circuitry and V_1 is the cost associated with each logic I/O of the single-domain logic. The total number of single domain inter-FPGA signals leaving G_{sd} is P_{sd} and d is the same distance factor used to amortize intermediate hops in (4). From [2], the virtual wires gate overhead associated with per-FPGA single-domain logic is

$$G_v = V_0 + V_1\, dK_c{G_{sd}}^{B_c} \tag{5}$$

which is obtained by substituting $K_c{G_{sd}}^{B_c}$ for P_{sd} in the virtual wires gate overhead equation. These gates allow for the multiplexing of device pins for single-domain logic. In total, the gate count for an FPGA (G_f) containing hard-wired MTSD logic,

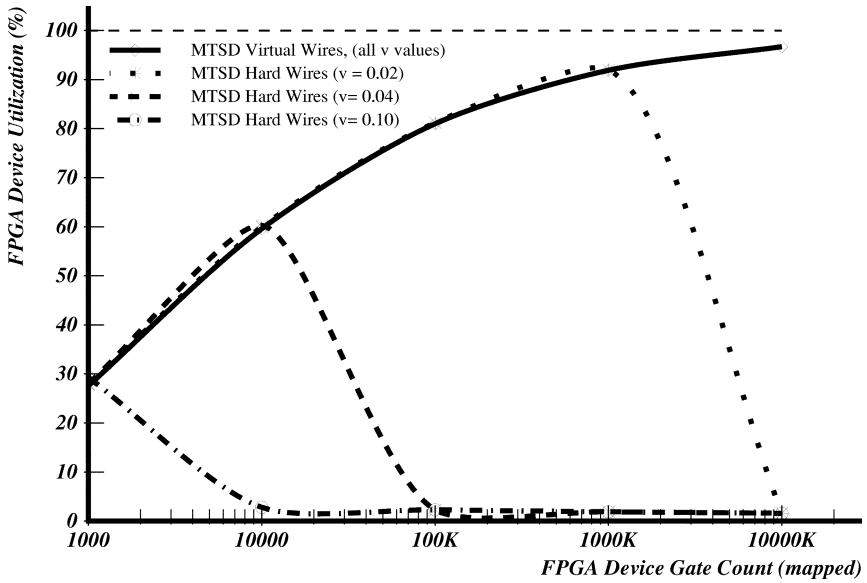


Fig. 17. Scalability with FPGA device size for FPGAs with MTSD logic.

single-domain logic, and virtual wires overhead for the single domain logic can be expressed as

$$G_f = G_c + G_v = (G_{\mathrm{mtsd}} + G_{sd}) + G_v \tag{6}$$

where G_c represents total user design gates in each partition. Given pin limitations, the available pins on an FPGA as a function of gate count are

$$P_f = K_f {G_f}^{B_f} \tag{7}$$

where K_f and B_f are the Rent parameters of the FPGA technology. In order for an FPGA to be able to support both single-domain logic which uses virtual wires and hard-wired MTSD logic, P_f, the total number of FPGA pins, must be greater than P_{mtsd}, the pins required for MTSD logic. If $P_f = P_{\mathrm{mtsd}}$, the partition contains only MTSD logic and single-domain logic must be placed in additional, separate FPGAs.

3) Virtual Routed MTSD Logic: In the second case that is considered, MTSD logic is distributed evenly across all system FPGAs and inter-FPGA MTSD nets are routed to adjacent FPGAs via *virtual wires*. Since *both* MTSD and single-domain logic use virtual routing, per-FPGA virtual wires overhead can be characterized by

$$G_v = V_0 + V_1 d {K_c}{G_c}^{B_c}. \tag{8}$$

For the virtual routing MTSD case, combined overhead and design gates leads the same relationship as (6) with G_v replaced by the relationship above.

4) Analysis of Results: Using the results from previous sections, we compare achievable FPGA device utilization (G_c/G_f) for devices containing MTSD logic. Parameters for the comparison are listed in Table IV. Both hard wire and virtual wire routing of MTSD signals are considered as FPGA size increases. In the following, we represent the per-device relationship between MTSD and single-domain logic gates via a fraction v such that v represents the fraction of the user design that is MTSD. Since logic is distributed evenly, v also represents the fraction of per-FPGA design gates which are MTSD

$$G_c = G_{\mathrm{mtsd}} + G_{sd} \tag{9}$$

$$G_{\mathrm{mtsd}} = v \times G_c \tag{10}$$

$$G_{sd} = (1 - v) \times G_c. \tag{11}$$

Pin requirements for hard-wired MTSD logic can be determined by inserting G_{mtsd} in (4). If $P_{\mathrm{mtsd}} = P_f$, single-domain logic cannot be accommodated in the FPGA due to the lack of available pins. This point can be represented with a maximum allowable fraction $v_{\max}$ of MTSD logic for a given FPGA gate count G_f. For designs at this $v_{\max}$ value or higher, FPGAs that contain MTSD logic must be dedicated and overall logic utilizations in these partitions are correspondingly low.

Fig. 16 shows that as FPGA gate capacity increases the maximum fraction of each FPGA that contains MTSD logic decreases significantly. This drop-off is further reflected in Fig. 17 which shows that the per-FPGA utilization of FPGAs decreases as FPGA size increases across a range of v values. Once v reaches $v_{\max}$, only MTSD logic can remain in the device due to pin limitations and overall logic utilization drops dramatically. In cases where *virtual wires* are used to route MTSD nets, high per-FPGA logic utilization is maintained regardless of v since both MTSD and single-domain pins are multiplexed. Fig. 17 shows that while FPGA utilization improves when MTSD logic is routed with virtual wires via approaches developed in this paper, evenly distributing MTSD logic with interconnect via hard wires is not scalable as FPGA gate capacities increase. Note that for virtual wires, relative G_v overhead versus design gates decreases as FPGA gate counts increase.

IX. Conclusion and Future Work

In this manuscript, a new general approach for dealing with multiple asynchronous clock domains in parallel functional verification systems has been presented. A new scheduling algorithm has been developed that allows transported signals to be

split into several single-domain versions and transmitted across inter-FPGA channels dedicated to signals sourced by a single clock. These constituent signals are subsequently merged together at the routing destination to form a causally correct result. The scheduling algorithm also statically determines multidomain latch modeling for parallel verification equipment so that setup and hold violations are avoided. The approach has been demonstrated on a VirtuaLogic emulation system for three large commercial benchmark designs. The algorithms were integrated into a commercial compiler for logic emulation. Experimental and theoretical results show that the approach is scalable and provides good modeling fidelity. As a result of this scalability, an improvement in overall system performance was obtained.

We plan to extend this approach to deal with hard-wired cores and MTSD I/O signals. The heterogeneous nature of these blocks presents special considerations for scheduling and interfacing.

ACKNOWLEDGMENT

The authors wish to thank C. Selvidge of Mentor Graphics who was a key technical force in support of this project. They also appreciate the assistance of S. Arole, V. Garg, A. Glaser, S. Krishnamoorthy, and P. Fariborz in the preparation of this manuscript.

REFERENCES

[1] J. Babb, R. Tessier, and A. Agarwal, "Virtual wires: Overcoming pin-limitations in FPGA based logic emulators," in *Proc. IEEE Workshop FPGA Based Custom Computing Machines*, Apr. 1993, pp. 142–151.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, "Logic emulation with virtual wires," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 609–626, June 1997.

[3] T. Corman *et al.*, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1992.

[4] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki, and A. Agarwal, "Emulation of a sparc microprocessor with the MIT virtual wires emulation system," in *Proc. IEEE Workshop FPGA's for Custom Computing Machines*, Apr. 1994, pp. 14–22.

[5] J. Gallagher, "Prototypes ensure pre-verification," *EE Times*, June 2000.

[6] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, M. Wang, and J. Nishimura, "Hardware emulation for functional verification of K5," in *Proc. Design Automation Conf.*, June 1996, pp. 315–318.

[7] S. Hauck, "Multi-FPGA systems," Ph.D. dissertation, Dept. Computer Science and Engineering, Univ. Washington, June 1995.

[8] Mentor Graphics Corporation. VirtuaLogic datasheet. [Online]. Available: http://www.ikos.com/products/VStation/index.html.

[9] H. Krupnova and G. Saucier, "FPGA-based emulation: Industrial and custom prototyping solutions," in *Proc. Int. Conf. Field-Programmable Logic Applications*, Aug. 2000, pp. 68–77.

[10] M. Kudlugi, C. Selvidge, and R. Tessier, "Static scheduling of multiple asynchronous domains for functional verification," in *Proc. Design Automation Conf.*, June 2001, pp. 647–652.

[11] ——, "Static scheduling of multidomain memories for functional verification," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2001, pp. 219–224.

[12] B. Landman and R. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Trans. Comput.*, vol. C-20, pp. 1469–1479, Dec. 1971.

[13] L. Malaniak, "Hardware emulation draws speed from innovative 3D parallel processing based on custom ICs," *Electron. Design Mag.*, May 1994.

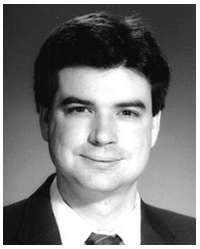
[14] Cadence Design Systems, Quickturn Emulation Division. Cobalt data sheet. [Online]. Available: http://www.quickturn.com/products/cobalt.htm.

[15] S. Patkar and P. Kurup, "ASIC design flow scores on first pass," *Integrated Systems Design Mag.*, Aug. 1997.

[16] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb, "TIERS: Topology independent pipelined routing and scheduling for virtual wire compilation," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, Feb. 1995, pp. 25–31.

Murali Kudlugi received the B.E. degree in computer science and technology from Bangalore University, in 1988, and the M.S. degree in computer science and engineering from the Indian Institute of Technology, Madras, in 1991. He is currently pursuing the Ph.D. degree at the University of Massachusetts, Amherst.

He is an Engineering Manager at the Emulation Division of Mentor Graphics Corporation, Waltham, MA. His research interests include system verification, HDL simulation, logic emulation, and synthesis.



Russell Tessier received the B.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1989, and the S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1992 and 1999, respectively.

He is an Assistant Professor of electrical and computer engineering at the University of Massachusetts, Amherst. He is a founder of Virtual Machine Works, a logic emulation company, and has also worked at BBN and Ikos Systems. Currently, he leads the Reconfigurable Computing Group at the University of Massachusetts. His research interests include computer architecture, field-programmable gate arrays, and system verification.