[18] I. Ghosh, S. Dey, and N. K. Jha, "A fast and low cost testing technique for core-based system-on-chip," in *Proc. ACM/IEEE Design Automation Conf.*, June 1998, pp. 542–547.
[19] K. Chakrabarty, "Design of system-on-a-chip test access architectures using integer linear programming," in *Proc. IEEE VLSI Test Symp.*, Apr. 2000, pp. 127–134.
[20] ——, "Design of system-on-a-chip test access architectures under place-and-route and power constraints," in *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 432–437.
[21] M. Nourani and C. Papachristou, "An ILP formulation to optimize test access mechanism in system-on-chip testing," in *Proc. IEEE Int. Test Conf.*, Oct. 2000, pp. 902–910.
[22] E. J. Marinissen, R. Kapur, and Y. Zorian, "On using IEEE P1500 SECT for test plug-n-play," in *Proc. IEEE Int. Test Conf.*, Oct. 2000, pp. 770–777.
[23] S. Koranne, "A novel reconfigurable wrapper for testing embedded core-based SOC's and its assiciated scheduling algorithm," *J. Electron. Testing: Theory Applicat.*, vol. 18, pp. 415–434, Aug. 2002.
[24] S. K. Goel and E. J. Marinissen, "A novel test time reduction algorithm for test architecutre design for core-based system chips," in *Dig. Papers IEEE Eur. Test Workshop*, May 2002, pp. 41–46.
[25] V. Iyengar, S. K. Goel, E. J. Marinissen, and K. Chakrabarty, "Test resource optimization for multisite testing of embedded-core-based SOC's using ATE with memory depth constraints," in *Dig. Papers IEEE Eur. Test Workshop*, May 2002, pp. 29–34.
[26] V. Iyengar, K. Chakrabarty, and E. J. Marinissen, "Integrated wrapper/TAM co-optimization, constraint-driven test scheduling, and tester data volume reduction for SOCs," in *Proc. ACM/IEEE Design Automation Conf.*, June 2002, pp. 686–690.
[27] J. Aerts and E. J. Marinissen, "Scan chain design for test time reduction in core-based ICs," in *Proc. IEEE Int. Test Conf.*, Oct. 1998, pp. 448–457.
[28] F. F. Hsu, K. M. Butler, and J. H. Patel, "A case study on the implementation of the illinois scan architecture," in *Proc. IEEE Int. Test Conf.*, Oct. 2001, pp. 538–547.
[29] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. Int. Symp. Circuits Syst.*, May 1989, pp. 1929–1934.
[30] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, B. Keller, and B. Koenemann, "OPMISR: The fundation for compressed ATPG vectors," in *Proc. IEEE Int. Test Conf.*, Oct. 2001, pp. 738–747.
[31] M. L. Bushnell and V. D. Agrawal, *Esentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Norwell, MA: Kluwer, 2000.
[32] R. Kapur, M. Lousberg, T. Taylor, B. Keller, P. Reuter, and D. Kay, "CTL, the language for describing core-based test," in *Proc. IEEE Int. Test Conf.*, Oct. 2001, pp. 131–139.
[33] E. J. Marinissen, "The role of test protocol expansion in automated test generation for embedded-core-based system ICs," *J. Electron. Testing: Theory Applicat.*, vol. 18, pp. 435–454, Aug. 2002.
[34] I. Goulden and D. Jackson, *Combinational Enumeration*. New York: Wiley, 1983.
[35] A. Sivaram, "Split timing mode (STM)—Answer to dual frequency domain testing," in *Proc. IEEE Int. Test Conf.*, Oct. 2001, pp. 738–747.

Optimal Path Routing in Single- and Multiple-Clock Domain Systems

Soha Hassoun, *Senior Member, IEEE*, and Charles J. Alpert, *Senior Member, IEEE*

***Abstract*—Shrinking process geometries and the increasing use of intellectual property components in system-on-chip designs give rise to new problems in routing and buffer insertion. A particular concern is that cross-chip routing will require multiple clock cycles. Another is the integration of independently clocked components. This paper explores simultaneous routing and buffer insertion in the context of single- and multiple-clock domains. We present two optimal and efficient polynomial algorithms that build upon the dynamic programming fast path framework. The first algorithm solves the problem of finding the minimum latency path for a single-clock domain system. The second considers routing between two components that are locally synchronous yet globally asynchronous to each other. Both algorithms can be used for interconnect planning. Experimental results verify the correctness and practicality of our approach.**

***Index Terms*—Algorithms, integrated circuit interconnection, multiple-clock domain system, routing, signal synthesis, single-clock domain system, system-on-chip (SoC).**

I. Introduction

Three distinct trends will pose new routing challenges in future system-on-chip (SoC) designs. First, SoCs will utilize several intellectual property (IP) components, both soft and hard, like embedded processors and memories. This aspect will allow IP reuse and it will reduce time to market. A shortest path for a signal between two chip components may thus be obstructed by IP blocks. Second, the drive for higher performance will continue to push the clocking frequencies. Third, shrinking process geometries and improvement in process technologies allows building bigger dies. Multiple clock cycles will be required to cross a chip. Furthermore, if the IPs are clocked at different frequencies, as is the case with a hard IP that often has a fixed clock period, then the signal route must cross from one time domain (of the sender) to another (that of the receiver), latched through the proper circuitry. In contrast to a system with a single clock, or a *single-clock domain* system, a system with multiple clocks is often referred to as a *multiple-clock domain* system. The clocking scheme is referred to as globally asynchronous locally synchronous (GALS) [3], [10].

When multiple clock cycles are needed to route a signal across a chip clocked by the same clock or its derivative, three solutions are possible. The first solution is combinational where the delay of the buffered signal from sender to receiver is more than one clock cycle. The receiver then utilizes circuitry to count a predesignated number of cycles before latching new data. The disadvantage of this technique is that consecutive sends cannot be overlapped and the throughput of the channel is seriously degraded. A second solution is pipelining the routed signal through synchronizers (edge-triggered registers or level-sensitive latches). Buffers are inserted whenever needed to boost the electric signal and optimize the delays in between the synchronizers. The clock signal is routed to each of the synchronizers. The third solution is wave pipelining. It eliminates one or more register along the

Manuscript received August 2, 2002; revised May 19, 2003. This work was supported in part by the National Science Foundation under CAREER Grant 0093324 and by a Tufts University Mellon grant.

S. Hassoun is with the Computer Science Department, Tufts University, Medford, MA 02155 USA.

C. J. Alpert is with IBM Austin Research Laboratories, Austin, TX 78758 USA.

Digital Object Identifier 10.1109/TCAD.2003.818378

route, thus allowing the simultaneous existence of several wavefronts along a wire. The key in making wave pipelining a success is ensuring that successive waveforms do not interfere. While reducing the clock load and easing clock distribution, wave pipelining is very sensitive to delay, process, and temperature variations—effects that are even more pronounced for long routes.

When routing a signal between two synchronous domains clocked by different and unrelated clocks, the critical issue that must be addressed is *metastability*: the clock of the latching register and the data may switch simultaneously. The register output then settles into an undefined region—neither a logical high nor a logical low. Several solutions have been proposed to alleviate this problem [4], [11], [14], [15]. The quality of the solution depends on the mean time-to-failure, which often requires an increase in latency or additional complication in the synchronization scheme.

This paper addresses two problems related to routing and buffering in future SoC designs. The first problem seeks an optimal buffered-routing path with synchronizer and buffer insertion within a single-clock domain system. The objective is to minimize the *cycle latency*, or equivalently, the total number of registers along the route. Constraints are imposed to ensure that when a signal is latched by a register, the resulting register-to-register delays are less than the clock period. We wish to avoid physical obstacles, to route through blocks such as data-paths, and to buffer the signal as needed to minimize the latency.

The second problem seeks to optimize the routing within a multi-clock domain system. Here, we adopt the multidomain communication circuitry proposed by Chelcea and Nowick [4] which buffers the signal from one clock domain to another via a special circuit structure, a multiclock first-in/first-out structure (MCFIFO). This MCFIFO both minimizes points of metastability in the design and allows for buffering (temporarily holding) data. If the MCFIFO is placed more than one sender clock cycle away from the sender, or more than one receiver clock cycle away from the receiver, then synchronization of the routed net to the appropriate clocks is needed. *Relay stations*, first proposed by Carloni *et al.* [2], are such synchronization elements. They provide both synchronization and temporary storage to allow the correct operation of the MCFIFO. While our algorithm specifically uses the relay stations and the MCFIFO, it can be easily adopted to utilize similar synchronization elements. Thus, our second problem considers simultaneous routing, buffer insertion, relay stations insertion, and MCFIFO insertion to achieve the minimum *time latency*. This corresponds to minimizing the latency of sending data along the net when the MCFIFO is empty.

Our proposed algorithms to solve these two problems are based on the the "fast path" framework proposed by Zhou *et al.* [17]. The actual "fast path" algorithm finds a minimum delay path for a net while simultaneously exploring all buffering and routing solutions. We extend the algorithm to handle additional *synchronization* elements such as registers, relay stations, and MCFIFOs while imposing the timing constraints required by the physical distances and the communication protocol. Recent work in routing across multiple clock cycles within a single domain adopts different approaches. Lu *et al.* describe a register/repeater block planning method during architectural floorplanning/interconnect planning stage [13]. Their method is based on identifying feasible regions in which flip–flops and repeaters can be arbitrarily inserted to satisfy both delay and cycle time constraints. Cocchini extends van Ginneken's dynamic programming framework [16] to optimally place registers and repeaters when given a tree routing topology [5]. Hassoun describes an adaptation to the fast path algorithm to construct a clocked buffered route using level-sensitive latches [9].

The algorithms presented here can be utilized either for interconnect planning purposes or for realizing the final routing implementation. During the design planning process, routing estimates can be achieved during architectural explorations to assess communication overhead once an initial floorplan is constructed. Wire widths and layer assignments are assumed. The early detection of communication overheads allows architects to explore microarchitecture tradeoffs that hide communication latencies. Once a route estimate is obtained, the RTL-level design description is updated to reflect the added latency associated with multicycle routing and to ensure a cycle-accurate design description that can be properly verified. The algorithms presented here could also be used during back-end design to synthesize physical routes. In this case, the route latency has been estimated, and the algorithm determines the optimal buffer and synchronizer placement. The addition of the registers may complicate the design flow, verification, and design-for-test effort. We, however, believe that verification and testing methodologies will evolve to accommodate the necessity of pipelined routes.

The remainder of the paper is as follows. Section II presents a detailed overview of the "fast path" algorithm. Section III introduces the single-domain routing and synchronized buffering problem and shows how to adapt the "fast path" framework to solve it. Section IV overviews MCFIFO and relay station communication schemes, thereby leading into a discussion of the problem of optimal path construction in designs realized using multiple clock domains. Finally, Section V presents experimental results and we conclude with Section VI.

II. Background: Fast Path Algorithm

In routing in single- and multiple-clock domains, we wish to explore all routing and synchronizer insertion options within a given routing area. Many aspects of the "fast path" algorithm [17] can be exploited to achieve this goal.

The fast path algorithm finds the minimum delay source-to-sink buffered routing path, while considering both physical obstacles (e.g., due to IP, memories, and other macro blocks) and wiring blockages (e.g., data path). To model physical and wiring blockages, one may construct a grid graph $G(V, E)$ (as in [1], [7], and [17]) over the potential routing area, whereby each node corresponds to a potential insertion point for a buffer or synchronization element, and each edge corresponds to part of a potential route. Edges in the grid graph which overlap with blockages are deleted, and nodes that overlap physical obstacles are labeled blocked. More precisely, we define a label function $p : V \leftarrow \{0, 1\}$ where $p(v) = 0$ if $v \in V$ overlaps a physical obstacle and $p(v) = 1$, otherwise.

For each edge $(u, v) \in E$, let $R(u, v)$ and $C(u, v)$ denote the capacitance and resistance of a wire connecting u to v. We use uniform capacitance and resistance for a given length assuming a fixed width and layer assignment. Let $R(g)$, K(g), and $C(g)$, respectively, denote the resistance, intrinsic delay, and input capacitance of a given buffer or synchronization element g. These values are determined for each buffer or synchronization element in the routing library. We use the resistance-capacitance π-model to represent the wires, a switch-level model to represent the gates, and the Elmore model to compute path delays.

A *path* from node s to t in the grid graph G is a sequence of nodes $(s = v_1, v_2, \ldots, v_k = t)$. An *optimized path* from s to t is a path plus an additional labeling m of nodes in the path. We have $m(s) = g_s, m(t) = g_t$, and $m(v_i) \in I \cup \{0\}$, where I is the set of buffers or synchronization elements which may be inserted on a node in the path between source s and sink t. Here, g_s is the driving gate located at s, g_t is the receiving gate located at t, and each internal node v may be assigned a gate from the set I or to not have a gate (corresponding to $m(v) = 0$). A path is *feasible* if and only if $p(v) = 1$ whenever

Fast Path (G,B,s,t,m')

Input: $G(V,E) \equiv$ Routing grid graph
$B \equiv$ Buffer library
$s \equiv$ source node
$t \equiv$ sink node
$m' \equiv$ initial labeling

Vars: $Q \equiv$ priority queue of candidates
$\alpha = (c,d,m,v) \equiv$ Candidate at v

Output: $m \equiv$ Complete labeling of s-t path

1. $Q \leftarrow \{(C(m'(t)),0,m',t)\}$.
2. **while** ($Q \neq \emptyset$) **do**
3. $(c,m,b,u) \leftarrow extract\_min(Q)$
4. **if** $c = 0$ **then**
return labeling m.
5. **if** $u = s$ **then**
$d' \leftarrow d + R(m(s)) \cdot c + K(m(s))$
push $(0,d',m,u)$ onto Q and prune.
continue
6. **for each** $(u,v) \in E$ **do**
$c' \leftarrow c + C(u,v)$
$d' \leftarrow d + R(u,v)(c + C(u,v)/2)$
push (c',d',m,v) onto Q and prune
7. **if** $p(u) = 1$ **and** $m(u) = 0$ **then**
8. **for each** $b \in B$ **do**
$c' \leftarrow C(b)$
$d' \leftarrow d + R(b) \cdot c + K(b)$
$m(u) = b$
push (c',d',m,v) onto Q and prune

Fig. 1. Fast path algorithm.

$m(v) \in I$. We assume that m is initialized to $m(s) = g_s, m(t) = g_t$, and $m(v) = 0$ for $v \in V - \{s,t\}$.

Let B be a buffer library consisting of noninverting buffers.

The minimum-delay buffered path problem, or the "fast path," can be expressed as follows: Given a routing graph $G(V,E)$, the set $I = B$, and two nodes $s,t \in V$, find a feasible optimized path from s to t such that the delay from s to t is minimized.

This problem can be optimally solved by the fast path algorithm [17] and also by the shortest path formulation proposed by Lai and Wong [12]. The latter formulation can also be extended to wire sizing. We choose to extend the fast path algorithm to handle the next two formulations since it does not require any lookup table computation and is likely more efficient when there is no wire sizing.

The main idea behind the fast path algorithm is to extend Dijkstra's shortest path algorithm to do a general labeling based on Elmore delays. Let the quadruple $\alpha = (c,d,m,v)$ represent a partial solution at node v, where c is the current input capacitance seen at v, d is the delay from v to t, and m is a labeling for the buffered path from v to t. The solution $\alpha_1 = (c_1,d_1,m_1,v)$ is said to be inferior to $\alpha_2 = (c_2,d_2,m_2,v)$ if $c1 \geq c2$ and $d1 \geq d_2$. For any path from s to t through v, a buffer assignment of m_1 from v to t is guaranteed to not be better than a buffer assignment of m_2 from v to t. Thus, α_1 can be safely deleted (or pruned) without sacrificing optimality. Pseudo-code of the fast path algorithm [17] is given in Fig. 1.

The core data structure used by fast path is a priority queue of candidates that keys off of the candidate's delay value. The algorithm begins by initializing Q to the set containing a single sink candidate. Then, candidates are iteratively deleted from the Q and expanded either to add an edge (Step 6) or a buffer from the library (Steps 7 and 8). If the source is reached, it is pushed onto the Q in Step 5, and when it is eventually popped from the queue, it is returned as the optimum solution (Step 4). With each addition to the queue, candidates for the current node are checked for inferiority and then pruned accordingly.

If we assume that G has n vertices, $|E| \leq 4n$ (which is true for a grid graph), and $|B| = k$, the complexity of fast path is $O(n^2k^2 \log nk)$.

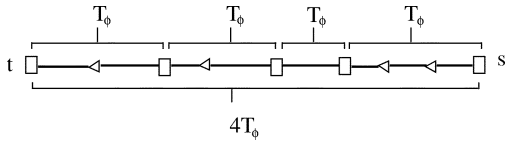

Fig. 2. Example of the single-clock domain routing. Latency is determined by the number of registers.

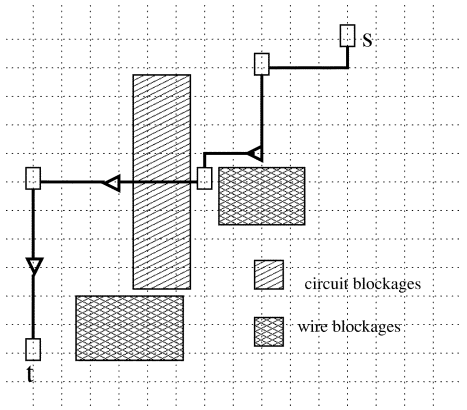

Fig. 3. Example of routing within a single-clock domain.

III. Single-Clock Domain Routing

We now explore the problem of finding a buffered routing path from s to t when multiple clock cycles are required. Routing over large distances in increasingly aggressive technologies will require several clock cycles to cross the die. Hence, one must periodically clock the signal by inserting synchronization elements (such as registers) along the signal path. In this case, one cannot simply treat a register like a buffer and add the register delay to the existing path delay in the fast path algorithm. The realizable delay between consecutive registers on a path will always be determined by the clock period, regardless of the actual signal propagation time. Register-to-register subpaths with delays larger than the permissible clock cycle are not permitted.

Let r denote the register to be used for insertion, T_ϕ the clock period, and $\mathrm{Setup}(r)$ to be the setup time for r. We extend the definition of node labeling to permit register assignment, i.e., $m(v) = r$ for any node $v \in V - \{s,t\}$. We also assume that the source and sink are synchronization elements, so that $g_s = g_t = r$. We define the clock period constraint as follows: a buffer-register path is *feasible* if and only if $p(v) = 1$ whenever $m(v) \in I$ and the buffered path delay between consecutive registers is less than or equal to $T_\phi - \mathrm{Setup}(r)$.

Since a register releases its signal with each clock switch, the s-t path *latency* is given by $T_\phi \times (p+1)$, where p is the number of registers on the s-t path. For example, Fig. 2 shows an s-t path with three registers between s and t, which means it takes four clock cycles to traverse from s to t. Note that in the figure the consecutive registers have different spacings, but the delay is always measured as T_ϕ between registers. Fig. 3 shows an example of a buffered-register path on a grid graph with both circuit and wire blockages. The physical area occupied by a register can be different from that occupied by a buffer. The register area is a function of the underlying circuit techniques. In addition, routing the clock to the register might cause added congestion. Our algorithm can be easily modified to allow register blockages that prevent inserting registers at undesirable grid points.

The problem of finding the minimum buffer-register path from s to t can be stated as follows.

Problem 1: Given a routing graph $G(V,E)$, the set $I = B \cup \{r\}$, and two nodes $s,t \in V$, find a feasible buffer-register path from s to t such that the latency from s to t is minimized.

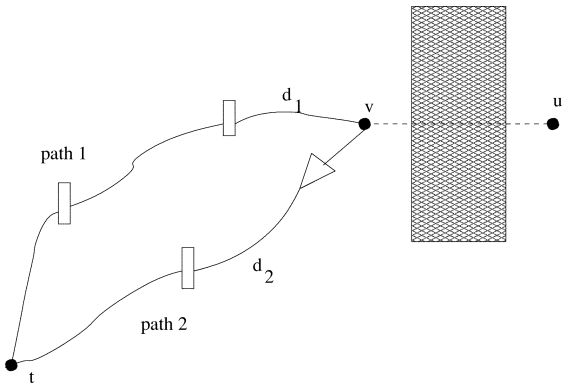

Fig. 4. Example of partial routing solutions that cannot be compared for pruning.

The objective is also equivalent to minimizing $|\{v \mid m(v) = r\}|$.

To solve Problem 1, one might initially try applying the fast path algorithm and treat the register like a member of the buffer library with the following caveat: candidates which violate the register-to-register delay constraint are immediately pruned. However, the fast path pruning scheme will not behave correctly.

Consider the two partial solutions from v to t in Fig. 4. Here, d_1 and d_2 are the delays from v to the first registers in the top and bottom paths, respectively. The top path has delay $2T_\phi + d_1$, while the bottom path has delay $T_\phi + d_2$. Since feasibility requires that both d_1 and d_2 be no greater than $T_\phi - \mathrm{Setup}(r)$, the bottom path delay is less than the top path delay. In addition, since there is a buffer on the bottom path close to v, v sees less downstream capacitance on the bottom path than on the top path. Since the top path is inferior to the bottom path in terms of capacitance and delay, the fast path algorithm would prune the candidate corresponding to the top path. However, consider continuing the route to node u on the other side of the circuit blockage from v. It is certainly possible that the delay from u to v plus d_2 exceeds the delay feasibility constraint, while the top path delay from u to v plus d_1 does not. In this case, only the top path can successfully be routed from v to u while still meeting feasibility requirements. Consequently, the top path cannot be pruned.

The key observation is that one should only prune subpaths by comparing them to other subpaths with the same number of registers. In the previous example, comparing a one-register path to a two-register path leads to an unresolvable inconsistency. Had the bottom path had two registers, then it could not have had smaller delay than the top path. Thus, one can still use the fast path algorithmic framework as long as candidate propagation proceeds in waves of partial solutions wherein each wave corresponds to a different number of registers. Fig. 5 shows how one can adapt the fast path framework to accomplish this in the registered-buffered path (RBP) algorithm.

The primary differences between RBP and the fast path algorithm are as follows.

1) RBP uses a second queue Q^* to store candidate solutions for the subsequent propagation wave. When a register is added to a candidate that is popped from Q it is added to Q^* and processed only after the current wave is completed. This pushing onto Q^* is accomplished in Step 8, whereby candidates are added only if the insertion of the register satisfies the clock feasibility constraint.
2) RBP combines Steps 4 and 5 from Fig. 1 into a single Step 4. RBP has the luxury of knowing that as soon as s is reached, a minimum latency solution is guaranteed; hence, it can immediately return the solution, as opposed to pushing it back onto the queue like fast path.

RBP Algorithm (G,B,s,t,m',r,T_ϕ)

Input: $G(V,E) \equiv$ Routing grid graph
$B \equiv$ Buffer library
$s \equiv$ source node
$t \equiv$ sink node
$m' \equiv$ initial labeling with $m'(s) = m'(t) = r$
$r \equiv$ register for clocking signal
$T_\phi \equiv$ required clock period.
Vars: $Q \equiv$ priority queue of candidates
$Q^* \equiv$ queue holding next candidate wave
$\alpha = (c,d,m,v) \equiv$ Candidate at v
$A \equiv$ Marking of registered nodes
Output: $m \equiv$ Labeling of complete s-t path

1. $Q \leftarrow \{(C(r), Setup(r), m', t)\}$.
   $Q^* = \emptyset$, $A(v) = 0$, $\forall v \in V$
2. **while** ($Q \neq \emptyset$) **or** ($Q^* \neq \emptyset$) **do**
   if ($Q = \emptyset$) **then**
   $Q = Q^*$, $Q^* = \emptyset$.
3. $(c,d,m,u) \leftarrow extract\_min(Q)$
4. **if** $u = s$ **then**
   $d' \leftarrow d + R(m(s)) \cdot c + K(m(s))$
   if $d' \leq T_\phi$ **then**
   return labeling m.
5. **for each** $(u,v) \in E$ **do**
   $c' \leftarrow c + C(u,v)$
   $d' \leftarrow d + R(u,v)(c + C(u,v)/2)$
   if $d' \leq T_\phi - K(r) - min(R(B \cup r))c'$ **then**
   push (c',d',m,v) onto Q and prune
6. **if** $p(u) = 1$ **and** $m(u) = 0$ **then**
7. **for each** $b \in B$ **do**
   $c' \leftarrow C(b)$, $m(u) = b$
   $d' \leftarrow d + R(b) \cdot c$ + K(b)
   if $d' \leq T_\phi - K(r)$ **then**
   push (c',d',m,u) onto Q and prune
8. **if** $A(u) = 0$ **and** $d + R(r) \cdot c + K(r) \leq T_\phi$ **then**
   $m(u) = r$, $A(u) = 1$
   push $(C(r), Setup(r), m, u)$ onto Q^*

Fig. 5. Registered–buffered path (RBP) algorithm.

3) When inserting a register for a candidate at node v, it is guaranteed to be the minimum latency candidate from v to t such that $m(v) = r$. Hence, there is no need to consider other candidates with $m(v) = r$. We use the array A to mark whether a solution with $m(v) = r$ has been generated for node r to prevent multiple candidates with registers inserted at v. Step 8 only adds a register to an existing candidate if none already exists.
4) The clock feasibility constraint is checked before pushing new candidates onto Q in Steps 7 and 8. This prevents solutions that can never lead to feasible solutions from further exploring the grid graph.

RBP proceeds by expanding all buffered-path solutions, just like fast path, until any further exploration violates the clock period constraint. At this point Q^* contains several newly generated candidates all of which are ready for wavefront expansion from a node with an inserted register. Step 2 dumps these candidates into Q and re-initializes Q^* to the empty set. These single-register candidates are then expanded, generating double-register candidates that are stored in Q^*, etc. When there are no blockages, the wave-front expansion looks like Fig. 6. Of course, with blockages, the wave fronts are not nearly as regular.

Let N be the number of nodes that can be reached from a given node in one clock cycle. When the clock period is sufficiently short, $N < n$, the complexity of the RBP algorithm is $O(nNk^2 \log Nk)$. The analysis is similar to that of the fast path algorithm [17]. Since the number of elements in the queue can be at most $O(Nk)$, the time required to insert an element into the queue is $O(\log(Nk))$. Because the number

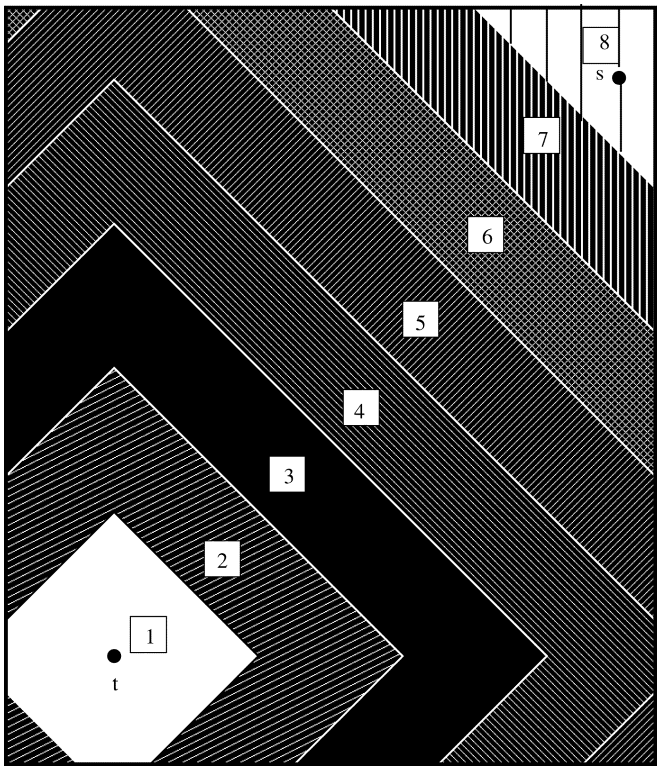

Fig. 6. Wave-front expansion.

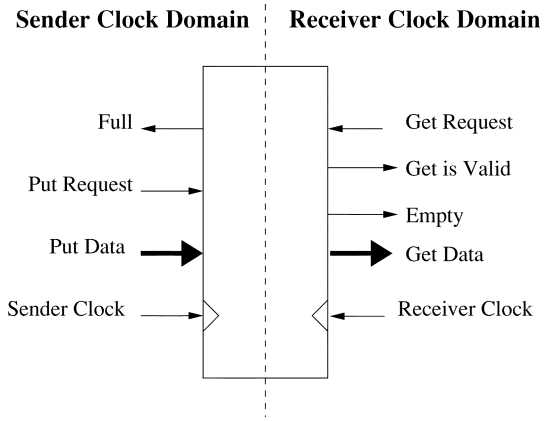

Fig. 7. Mixed clock FIFO that interfaces two different clock domains.

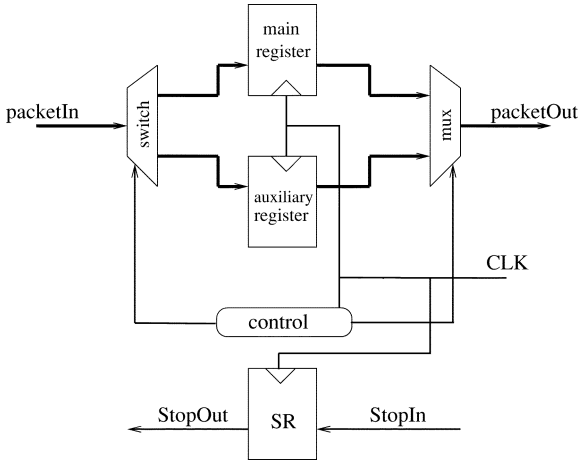

Fig. 8. Relay station.

of insertion operations of either buffers or latches is bounded by nNk^2, the complexity is $O(nNk^2log(Nk))$. This is a lesser time complexity than the fast path algorithm. The computational savings occurs because we do not have to waste resources exploring the many paths that violate the clock period constraint. This speedup can be seen in practice in the experimental results when observing the number of configurations that are examined as well as the run times.

While our algorithm optimizes for route latency, it can be easily modified to find the minimum-latency path that maximizes the sum of the slack at the source and at the sink. Each candidate solution will keep track of the sink slack value. When the algorithm reaches the source (i.e., in Step 4 in Fig. 5), *all* minimum-latency solutions are explored to find the one that maximizes the sum of the sink and source slacks. It is also interesting to note that the RBP algorithm could be implemented in an alternative manner. Instead of using two queues, one could use an array of queues indexed by the number of registers. A new candidate is then inserted into the queue indexed by the number of registers in the solution represented by that candidate. The time complexity of the algorithm would not change, though this implementation uses more memory.

IV. Multiple-Clock Domain Routing

A. Background

As mentioned in Section I, we adopt the MCFIFOs proposed by Chelcea and Nowick [4] to route a signal between two different clock domains. The MCFIFO is the basic entity that establishes data communication between two modules operating at different frequencies.

Like all FIFOs, the MCFIFO has a *put* interface to the sender and a *get* interface to the receiver. Each interface is clocked by the communicating domain's clock as illustrated in Fig. 7. If the full signal is not asserted, then the sender can request a put (*Put Request* signal) and data is placed on the *Put Data* wires. The data is latched into the FIFO at the next edge of the sender's clock. If the empty signal is not asserted, then the receiver can request data (*Get Request* signal). The data is then made available at the receiver's next clocking edge. The *Get is Valid* signal determines if the data is valid.

Because it may take multiple sender clock cycles to route a net from its source in the routing grid to the MCFIFO, and multiple receiver clock cycles to route the net from the MCFIFO to the sink, signals must be synchronized to the clock of each domain. Chelcea and Nowick extend the concept of a single-domain relay station [2] to do so. These stations essentially allow breaking long wires into segments that correspond to clock cycles and then a chain of relay stations act like a distributed FIFO.

The single-domain relay station is shown in Fig. 8. It contains a main register and an auxiliary one. Initially, both are empty and the control selects the main register for storing and reading the packet. When the *StopIn* signal is asserted, the next incoming packet is stored in the auxiliary register. *StopOut* is also asserted on the next clock cycle to indicate that the relay station is *Full* and cannot further accept new data.

To adapt the single-domain relay stations to interface properly with the MCFIFO, the relay stations bundle the *Put Request* and *Put Data* as the incoming packet, and the *Get is Valid* and *Get Data* signals as the outgoing packet. The full signal in the MCFIFO is used to stop the incoming flow of packets. An MCFIFO and two adjacent relay stations are shown in Fig. 9.

B. GALS Algorithm

Although for the multiclock domain problem we are using the MCFIFO and relay-stations that require bidirectional signal flow, we ab-

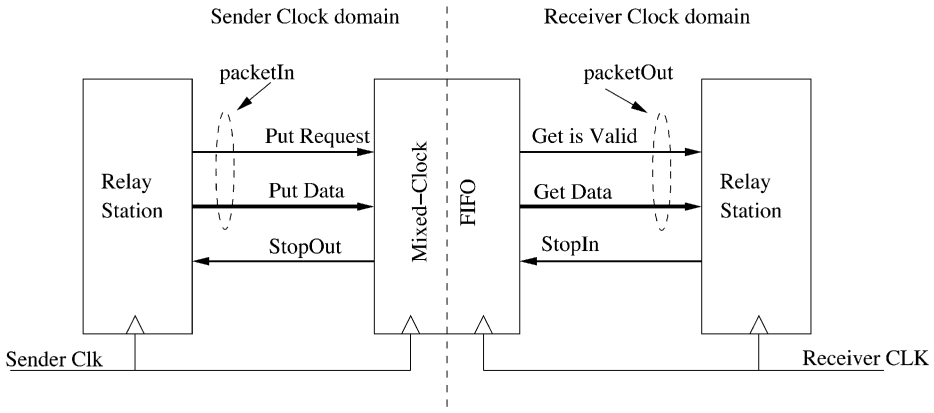

Fig. 9. Mixed clock FIFO and relay stations.

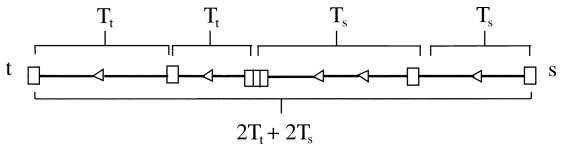

Fig. 10. Example of an MCFIFO-register routing. The MCFIFO in breaks the periods into two clock domains where the period is T_s on the source side of the MCFIFO and T_t on the sink side. The total latency is $2T_s + 2T_t$.

stract the communication as being single directional. We view relay station as a register r because both have similar delay properties. Given any buffered path between relay stations r_1 and r_2, if one assumes a single buffer type with the same delay characteristics as the register, then the Elmore delay from r_1 to r_2 is actually identical to the Elmore delay from r_2 to r_1. Inserting a buffer in our multiclock domain problem formulation actually means requiring the insertion of two buffers, one for each direction of signal flow.

Let f denote the MCFIFO element that must be inserted on the routing path, T_s to be the clock period before f, and T_t to be the clock period after f. Fig. 10 shows an example where there are two clock periods between s and the MCFIFO and two clock periods after the MCFIFO. Since the clocks have different periods, the total latency is given by $2T_s + 2T_t$. This corresponds to the signal flow assuming an empty MCFIFO and ignores the worst case synchronization delay within the MCFIFO that is common to all routing solutions.

Our set of insertable elements is now $I = B \cup \{r, f\}$. For a multiclock domain source-to-sink path (an MCFIFO path), we use the following conditions for feasibility: an MCFIFO path is *feasible* if and only if:

- $p(v) = 1$ whenever $m(v) \in I$;
- $m(v) = f$ for exactly one $v \in V$;
- buffered path delay between consecutive registers between s and f is less than or equal to $T_s - \mathrm{Setup}(r)$;
- buffered path delay between consecutive registers between f and t is less than or equal to $T_t - \mathrm{Setup}(r)$.

For example, Fig. 11 shows a solution on the routing graph with a single MCFIFO with latency $T_s + 2T_t$.

Thus, the multiple clock domain, buffered routing problem becomes as follows.

Problem 2: Given a routing graph $G(V, E)$, the set $I = B \cup \{r, f\}$, and two nodes $s, t \in V$, find a feasible MCFIFO path from s to t such that the latency from s to t is minimized.

One can adopt a similar framework as in the RBP algorithm potentially inserting an MCFIFO element for every candidate, wherever a register is inserted. We call this new algorithm GALS. There are several modifications to RBP that must be considered to obtain the GALS algorithm.

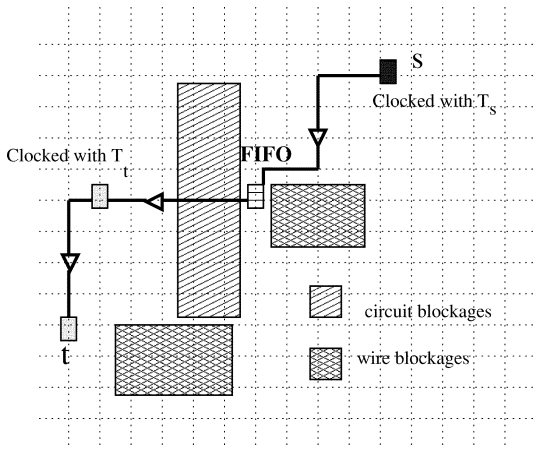

Fig. 11. Example illustrating routing within a multiple domain system.

1) A GALS candidate must know if the MCFIFO has been inserted, so now we use the six-tuple $\alpha = (c, d, b, v, z, l)$ where $z = 0$ if α does not contain an MCFIFO and $z = 1$ otherwise. Let $T(0) = T_t$ and $T(1) = T_s$ be a function which returns the required clock period for a given z value. The latency l is discussed below.
2) GALS pruning takes place only with candidates with the same value of z. Two candidates with opposing values of z are not directly compared for pruning. Instead of storing a single list of candidates for each grid node, now we store two lists: one for $z = 0$ and one for $z = 1$. If we have not yet inserted an MCFIFO onto the path, pruning is done with respect to the $z = 0$ list; otherwise, it is done with respect to the $z = 1$ list.
3) Because $T_s \neq T_t$, one cannot find identical elements for wave front expansion as easily as in the single-clock domain case. In RBP, the number of registers obviously determined the latency. For GALS, one four-register path may have latency $2T_s + 3T_t$ while another four-register path has latency $T_s + 4T_t$. Whichever path has smaller latency must be explored first. Thus, the candidate value l stores the latency from the most recently inserted register or MCFIFO back to the sink t. Just like the RBP algorithm, d still stores the combinational delay from the current node to the most recently inserted register or MCFIFO.
4) The elements in Q are still ordered by d, but the elements in Q^* are ordered by l. We define the operation $Q = \mathrm{ExtractAllMin}(Q^*)$ to pull all elements off of Q^* with

GALS Algorithm (G,B,s,t,m',r,f,T_s,T_t)

Input: $G(V,E) \equiv$ Routing grid graph
$B \equiv$ Buffer library
$s \equiv$ source node
$t \equiv$ sink node
$m' \equiv$ initial labeling with $m'(s)$, $m'(t)$
$r \equiv$ register for clocking signal
$f \equiv$ MCFIFO element
$T_t \equiv$ required clock period from f to t.
$T_s \equiv$ required clock period from s to f.

Vars: $Q \equiv$ priority queue keyed by d
$Q^* \equiv$ priority queue keyed by l
$\alpha = (c,d,m,v,z,l) \equiv$ Candidate at v
$A_0, A_1 \equiv$ Marking of registered nodes

Output: $m \equiv$ Labeling of complete s-t path

1. $Q \leftarrow \{(C(r), Setup(r), m', t, 0, 0)\}$.
   $Q^* = \emptyset$, $A_0(v) = A_1(v) = 0, \forall v \in V$
2. **while** ($Q \neq \emptyset$) **or** ($Q^* \neq \emptyset$) **do**
   if ($Q = \emptyset$) **then**
   $Q = ExtractAllMin(Q^*)$
   continue.
3. $(c,m,b,u,z,l) \leftarrow extract\_min(Q)$
4. **if** $u = s$ **then**
   $d' \leftarrow d + R(m(s)) \cdot c + K(m(s))$
   if $z = 1$ **and** $d' \leq T_s$ **then**
   return labeling m.
5. **for each** $(u,v) \in E$ **do**
   $c' \leftarrow c + C(u,v)$
   $d' \leftarrow d + R(u,v)(c + C(u,v)/2)$
   if $d' \leq T(z)$ **then**
   push (c', d', m, v, z, l) onto Q and prune
6. **if** $p(u) = 1$ **and** $m(u) = 0$ **then**
7. **for each** $b \in B$ **do**
   $c' \leftarrow C(b)$, $m(u) = b$
   $d' \leftarrow d + R(b) \cdot c$ + K(b)
   if $d' \leq T(z)$ **then**
   push (c', d', m, u, z, l) onto Q and prune
8. **if** $A_z(u) = 0$ **and** $d + R(r) \cdot c + K(r) \leq T(z)$ **then**
   $m(u) = r$, $A_z(u) = 1$
   push $(C(r), Setup(r), m, u, z, l + T(z))$ onto Q^*
9. **if** $z = 0$, $F(u) = 0$ **and** $d + R(f) \cdot c + K(f) \leq T(z)$ **then**
   $m(u) = f$, $F(u) = 1$
   push $(C(f), Setup(f), m, u, 1, l + T_t)$ onto Q^*

Fig. 12. GALS algorithm.

the same latency and load them into Q. This operation extracts the next wave front of elements with equal latency from Q^*.

5) In RBP, the first register inserted at a grid node v precludes the need to insert registers at v for any other path. RBP uses $A(v) \in \{0,1\}$ to represent whether a register had been seen in a path at v. In GALS, we need to separate the cases of inserting a register before f and after f. Let $A_0(v) \in \{0,1\}$ represent whether a register was inserted between f and t at v and $A_1(v) \in \{0,1\}$ represent whether a register was inserted between s and f at v. Also, let $F(v) \in \{0,1\}$ denote whether an MCFIFO was inserted at v.

Fig. 12 gives a template of the GALS algorithm. The main differences between GALS and RBP is the addition of Step 9 for inserting MCFIFO elements. Just like registers in RBP, GALS considers inserting an MCFIFO at each possible internal node as the wavefront expansion proceeds. Other differences include using $T(z)$ to look up the current clock period constraint, returning a solution in Step 4 only if it has an MCFIFO, and the wave-front queue mechanism of Step 2.

If N is the number of nodes that can be reached in $\max(T_s, T_t)$, the time complexity of GALS is $O(nNk^2 \log Nk)$, which is same as the RBP algorithm.

V. Experimental Results

We obtained code for the fast path algorithm from the authors of [17], then implemented RBP and GALS using this framework. The code is written in C and was run on a Sun Solaris Enterprise 250. To perform the experiments, we use estimated parameters for a 0.07-μ technology as reported by Cong and Pan [8]. We use a single buffer size of 100 times minimum gate width, triple wide wires, and assume delay characteristics for the register and MCFIFO to be identical to that of the buffer. As in [6], we use a 25 by 25 mm chip and place the source and sink 40 mm apart. These choices ensure that a significant number of clock cycles will be required to propagate a signal from s to t. Our first two experiments focus on the RBP algorithm while the third experiment focuses on the GALS algorithm.

A. Single-Clock Domain With Varying Period

Our first experiment investigates the behavior of the RBP algorithm as a function of the clock period. Given a grid separation of 0.125 mm and a grid size of 200 × 200, we varied the number of registers that can be placed along the path that is separated by 159 grid edges.

The results are summarized in Table I. The first data row in the table (with $T_\phi = \infty$) presents the results of running the fast path algorithm, where the reported latency is actually the minimum-buffered path delay. The other rows are the results of running the RBP algorithm with the indicated clock period.[1] The clock period and latencies are reported in picoseconds. The maximum and minimum register separation is given, as well as the separation between any inserted elements (i.e., a register or a buffer followed by a consecutive register or buffer). We make the following observations.

1) As the clock period decreases, the number of registers along the path increases while the number of buffers and the maximum and minimum register separation all decrease. However, the maximum and minimum separation between consecutive registers and buffers do not consistently decrease unless the clock period is so small that registers are inserted every one or every other grid point.
2) The number of configurations investigated (i.e., candidates popped off the queue Q in Fig. 5) decreases with decreasing clock period. This empirically confirms that the run time complexity of RBP becomes more efficient as the clock period decreases, because the space of feasible wavefront expansion in a single cycle is reduced.
3) Because RBP has additional overhead for register insertion, only when the clock period drops below a certain threshold do we see a run-time improvement over fast path, e.g., in this case it is for $T_\phi = 463$.

B. Single-Clock Domain With Varying Grid Size

Next, we investigate the behavior RBP as a function of the clock period and the grid separation. We experimented with three grid separations: 0.5, 0.25, and 0.125 mm. We summarize the results in Table II. The first data row for each grid size corresponds to the results of the fast path algorithm, while the others represent the results of running our RBP algorithm. We observe the following.

1) As the grid becomes more refined, fast path latency improves slightly, from 2741 to 2739 ps. The improvements may be more significant when blockages along the path are present.
2) It is possible to achieve a smaller latency with a more refined grid. For example, at a clock period of 925 and a 50 × 50 grid,

[1]The seemingly odd choices for the clock period are actually the fastest clock periods required to achieve the given number of registers (rounded to the nearest picosecond). For example, a T_ϕ of 686 is the fastest clock period that achieves a three-register solution.

TABLE I
FASTREGPATH STATISTICS AS A FUNCTION OF T_ϕ. MAXREGSEP/MINREGSEP REFERS TO THE MAX/MIN NUMBER OF GRID POINTS BETWEEN SUCCESSIVE REGISTERS. MAX R/B SEP REFERS TO THE MAXIMUM GRID SEPARATION BETWEEN A REGISTER OR A BUFFER AND THE FOLLOWING REGISTER OR BUFFER. ON THIS GRID, THE NUMBER OF GRID EDGES ALONG A SHORTEST PATH BETWEEN SINK AND SOURCE WAS 159

T_ϕ (ps)	Latency (ps)	Registers	Buffers	MaxRegSep	MinRegSep	Max R/B Sep	Min R/B Sep	Configs	MaxQSize	time (sec)
∞	27397	0	16	-	-	19	18	1014896	5951	28.95
1371	2742	1	14	160	160	21	19	918078	19759	35.41
925	2775	2	14	108	104	27	17	881092	19512	34.84
686	2744	3	12	80	80	22	19	805603	13518	30.90
551	2755	4	10	64	64	23	20	755814	12558	29.55
463	2778	5	11	54	50	29	17	694386	9981	27.50
398	2786	6	7	46	44	26	18	638676	9265	25.46
343	2744	7	8	40	40	21	19	571877	7978	22.88
261	2871	10	10	30	20	20	14	468975	6193	19.02
84	3360	39	0	8	8	8	8	78122	1722	6.57
67	4288	63	0	5	5	5	5	78246	1098	6.59
62	4960	79	0	4	4	4	4	78278	876	6.63
53	8480	159	0	2	2	2	2	78360	442	6.55
49	15680	319	0	1	1	1	1	78416	312	6.44

TABLE II
RBP PERFORMANCE AS A FUNCTION OF CLOCK PERIOD AND GRID SIZE. MAX. (MIN.) SEPARATION REFERS TO THE MAXIMUM (MINIMUM) BUFFER SEPARATION WHEN THE CLOCK PERIOD IS ∞, AND TO THE CORRESPONDING REGISTER SEPARATION OTHERWISE

Grid Separation: 0.5*mm*:50×50grid.														
Period(ps)	∞	1371	925	686	551	463	398	343	261	84	67	62	53	49
Registers	-	1	3	3	5	6	7	7	11	39	79	79	-	-
Buffers	15	14	12	12	10	6	7	8	0	0	0	0	-	-
Latency	2741	2742	3700	2744	3306	3241	3184	2744	3132	3360	5360	4960	-	-
Max. Sep.	5	40	26	20	15	13	11	10	7	2	1	1	-	-
Min. Sep.	5	40	2	20	5	2	3	10	3	2	1	1	-	-
time(s)	0.41	0.70	0.76	0.69	0.73	0.70	0.68	0.61	0.59	0.42	0.38	0.36	-	-
Grid Separation: 0.25*mm*:100×100grid.														
Period	∞	1371	925	686	551	463	398	343	261	84	67	62	53	49
Registers	-	1	2	3	4	5	7	7	10	39	79	79	159	-
Buffers	16	14	14	12	10	11	7	8	10	0	0	0	0	-
Latency	2740	2742	2775	2744	2755	2778	3184	2744	2871	3360	5360	4960	8480	-
Max. Sep.	10	80	54	40	32	27	22	20	15	4	2	2	1	-
Min. Sep.	9	80	52	40	32	25	6	20	10	4	2	2	1	-
time(s)	3.77	5.63	5.52	5.10	4.78	4.45	4.33	3.69	3.08	1.63	1.69	1.61	1.63	-
Grid Separation: 0.125*mm*:200×200grid.														
Period	∞	1371	925	686	551	463	398	343	261	84	67	62	53	49
Registers	-	1	2	3	4	5	6	7	10	39	63	79	159	319
Buffers	16	14	14	12	10	11	7	8	10	0	0	0	0	0
Latency	2739	2742	2775	2744	2755	2778	2786	2744	2871	3360	4288	4960	8480	15680
Max. Sep.	19	160	108	80	64	54	46	40	30	8	5	4	2	1
Min. Sep.	18	160	104	80	64	50	44	40	20	8	5	4	2	1
time(s)	28.95	35.41	34.84	30.90	29.55	27.50	25.46	22.88	19.02	6.57	6.59	6.63	6.55	6.44

the latency is 3700, but it is 2775 when we use a 100×100 grid. In some cases, no improvements were possible such as for clock periods 67 and 62.

3) With a coarse grid and at very small clock periods, it is impossible to find a routing solution as the grid separation demands placing registers less than one grid edge apart. The finer grid allows placing the registers closer. No solution for example is found At clock periods 53 and 49 for grid size 50×50, and for clock period 49 for grid size 100×100.
4) With larger clock periods, it is possible to achieve a latency close to the optimal buffered-path delay. For example, using a 200×200 grid, at all clock periods shown above 84 ps it is possible to be within one clock period from the optimal path delay of 2739.

C. GALS for Multiple-Clock Domains

Our final experiment explores the behavior of the GALS algorithm for different periods of the clock domain. Given our new problem statement, comparing to fast path is not possible. We simply illustrate results of the technique.

TABLE III
GALS STATISTICS AS A FUNCTION OF T_s AND T_t WITH A GRID SEPARATION OF 0.125 mm

T_s / T_t	300 / 300	200 / 300	300 / 200	300 / 400	400 / 300	250 / 300	300 / 250
Buffers	9	2	2	8	8	7	6
Reg-t	8	1	10	3	3	6	2
Reg-s	0	10	1	3	3	2	6
latency	3000	2800	2800	2800	2800	2850	2850

We ran GALS on the same test case in the previous experiment using a grid separation of 0.125 mm. Table III reports the number of buffers inserted, the number of registers on the sink side of the MCFIFO (Reg-t), the number of registers on the source side of the MCFIFO (Reg-s), and the latency. The relative values of Reg-t and Reg-s indicate whether the MCFIFO was placed close to the source or to the sink. For example, when $T_s = T_t = 300$, the algorithm places the MCFIFO close to the source, but when $T_s = 200$ it places it closer to the sink. Thus, we cannot generalize the behavior on the optimal location of the MCFIFO, it depends on the blockage map, clock periods T_s,

and T_t and the technology parameters. For all cases, we observe that the total latency is not significantly higher than the minimum source-sink delay of 2739 ps (from Table I).

VI. Conclusion

Automated buffered routing is a necessity in modern very large-scale integration design. The contributions of this paper are two new problem formulations for buffered routing for single- and multiple-clock domains. Both of these formulations address problems that will become more prominent in future designs. Any computer-aided design (CAD) tools currently performing buffer insertion will eventually have to deal with synchronizer insertion. Furthermore, any SoC routing CAD tools will have to handle routing across multiple clock domains due to the increasing use of IPs.

We solve both problems optimally in polynomial time via the RBP and GALS algorithms that build upon the fast path algorithm of [17]. Experimental results validate the correctness and practicality of the two algorithms for an aggressive technology.

Acknowledgment

The authors would like to thank H. Zhou for supplying fast path code and also to M. Thiagarajan for help with the figures and researching the background material on the MCFIFOs.

References

[1] C. J. Alpert, G. Gandham, J. Hu, S. T. Quay, J. L. Neves, and S. S. Saptnekar, "Steiner tree optimization for buffers and blockages and bays," *IEEE Trans. Comput.-Aided Design*, vol. 20, pp. 556–562, Apr. 2001.

[2] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1999.

[3] D. Chapiro, "Globally asynchronous locally asynchronous systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1984.

[4] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2001, pp. 21–26.

[5] P. Cocchini, "Concurrent flip-flop and repeater insertion for high performance integrated circuits," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, pp. 268–273, 2002.

[6] J. Cong, "Timing closure based on physical hierarchy," in *Proc. Int. Symp. Physical Design*, 2002, pp. 170–174.

[7] J. Cong, J. Fang, and K.-Y. Khoo, "An implicit connection graph maze routing algorithm for ECO routing," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, pp. 163–167, 1999.

[8] J. Cong and Z. Pan, "Interconnect performance estimation models for design planning," *IEEE Trans. Comput.-Aided Design*, vol. 20, pp. 739–752, June 2001.

[9] S. Hassoun, "Optimal use of 2-phase transparent latches in buffered maze routing," *Proc. IEEE Int. Symp. Circuits Syst.*, 2003.

[10] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Obert, P. Ellervee, and D. Lundqvist, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," presented at the *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 1999.

[11] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, "Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems," presented at the *Proc. 12th Annu. IEEE Int. ASIC/SOC Conf.*, 1999.

[12] M. Lai and D. F. Wong, "Maze routing with buffer insertion and wiresizing," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2000, pp. 374–378.

[13] R. Lu, G. Zhong, C. Koh, and K. Chao, "Flip-flop and repeater insertion for early interconnect planning," in *Proc. Design Automation Test Europe Conf. (DATE)*, 2002, pp. 690–695.

[14] J. Rabaey, *Digital Integrated Circuits*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[15] J.-N. Seizovic, "Pipeline synchronization," *Proc. IEEE ASYNC*, 1994.

[16] L. P. P. P. van Ginneken, "Networks for minimal Elmore delay," *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 865–868, 1990.

[17] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," *IEEE Trans. Comput.-Aided Design*, vol. 19, pp. 819–824, July 2000.

Fast Computation of Symmetries in Boolean Functions

Alan Mishchenko, *Member, IEEE*

***Abstract*—Symmetry detection in completely specified Boolean functions is important for several applications in logic synthesis, technology mapping, binary decision diagram (BDD) minimization, and testing. This paper presents a new algorithm to detect four basic types of two-variable symmetries. The algorithm detects all pairs of symmetric variables in one pass over the shared BDD of the multioutput function. The worst case complexity of this method is cubic in the number of BDD nodes, but on typical logic synthesis benchmarks the complexity appears to be linear. The computation is particularly efficient when the functions have multiple symmetries or no symmetries. Experiments show that the algorithm is faster than other known methods, and in some cases achieves a speedup of several orders of magnitude.**

***Index Terms*—Binary decision diagrams (BDDs), Boolean functions, recursive procedures, symmetric variables, symmetry, zero-suppressed binary decision diagrams (ZBDDs).**

I. Introduction

The problem of symmetry detection in Boolean functions has a long history and many applications, such as functional decomposition in technology-independent logic synthesis [6], [9], [11], Boolean matching in technology mapping [12], [13], and binary decision diagram (BDD) minimization [21].

Early methods to detect symmetries are based on checking the equality of two-variable cofactors of the function $F_{01} = F_{10}$ and $F_{00} = F_{11}$. Decomposition charts [18] and truth tables [4] have been used to compute and compare cofactors. Representing functions using BDDs [1] improved the efficiency of cofactor computation. However, computing multiple cofactor pairs is still expensive for large functions because repeated cofactoring leads to creating and deleting a large number of intermediate BDD nodes. In this paper, cofactor checking is referred to as the *naïve method* to detect symmetries.

To bypass the cofactor computation, recent more sophisticated approaches use dynamic BDD variable reordering [19], generalized Reed–Muller forms [23], and analysis of shared BDDs [16]. The latter approach was recently successfully applied to BDD minimization in [21].

Here we review the symmetry detection algorithm described in [16] and [21] as the most computationally efficient. This algorithm is based on the observation that the absence of symmetry between a pair of variables can, in many cases, be discovered without computing and comparing the cofactors. To prove the absence of symmetry counting

Manuscript received July 16, 2002; revised December 23, 2002 and May 7, 2003. This work was supported by a research grant from Intel Corporation.

The author is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA (e-mail: alanmi@eecs.berkeley.edu).

Digital Object Identifier 10.1109/TCAD.2003.818371