Optimal Buffered Routing Path Constructions for Single and Multiple Clock Domain Systems

Soha Hassoun, Charles J. Alpert*, *and* Meera Thiagarajan
Tufts University, Medford, MA
*IBM Austin Research Laboratory, Austin, TX

Abstract

Shrinking process geometries and the increasing use of IP components in SoC 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 optimal and efficient polynomial algorithms that can be used to estimate communication overhead for interconnect and resource planning in single and multi-clock domain systems. Experimental results verify the correctness and practicality of our approach.

1 Introduction

Three distinct trends will pose new routing challenges in future SoC designs. First, SoCs will utilize several IP (Intellectual Property) components, both soft and hard, like embedded processors and memories. 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 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 Asynchronous, or *GALS* [3, 8].

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 insertion, 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.

The second problem seeks to optimize the routing within a multi-clock domain system. The critical issue in latching a signal from another clock domain is *metastability*: the clock of the latching register and the data 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, 12, 11, 9]. In this work we adopt the multi-domain communication circuitry proposed by Chelcea and Nowick [4] which buffers the signal from one clock domain to another via a special circuit structure, a *Multi-Clock FIFO*. 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. We use *Relay Stations* [2, 4] along with the MCFIFO to construct an optimal routed path between two signals in different clock domains. 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*.

Our proposed algorithms to solve these two problems are based on the the "Fast Path" framework proposed by Zhou, Wong, Liu, and Aziz [13]. 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.

The presented algorithms can be utilized within various points of the design flow. For example, during the design planning process, routing estimates can be achieved during architectural explorations to assess communication overhead once an initial floorplan is constructed. Early detection of communication overheads would allow architects to explore microarchitecture tradeoffs that hide communication latencies. The algorithms could also be used during backend physical design to realize actual synthesized, physically realizable paths.

The remainder of the paper is as follows. Section 2 presents a detailed overview of the "Fast Path" algorithm. Section 3 introduces the single-domain routing and synchronized buffering problem and shows how to adapt the "Fast Path" framework to solve it. Section 4 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 5 presents experimental results. We conclude in Section 6.

0-7803-7607-2/02/$17.00

2 Background: The 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[13] 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 [13, 1, 6]) 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 wiring 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. 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. We use the resistance-capacitance π-model to represent the wires b, 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 not have a gate (corresponding to $m(v) = 0$). A path is *feasible* if and only if $p(v) = 1$ whenever $m(v) \in I$. 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 non-inverting 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 [13] and also by the shortest path formulation proposed by Lai and Wong [10]. 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 [13] is given in Figure 1.

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

Figure 1. The Fast Path algorithm.

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 $O(n^2k^2 \log nk)$.

3 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 sub-paths 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 $Setup(r)$ 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 $T_\phi - Setup(r)$.

Figure 2 shows an example of a buffered-register path on a grid graph with both circuit and wire blockages. Because a register releases its signal with clock cycle, 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. The *s*-*t* path has three registers between s and t. It thus takes four clock cycles to traverse from s to t despite the different spacings between consecutive registers.

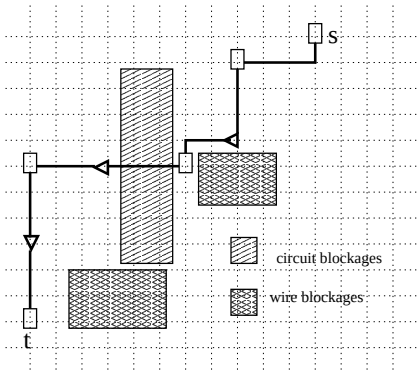

Figure 2. An example of routing within a single clock domain

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

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.

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.

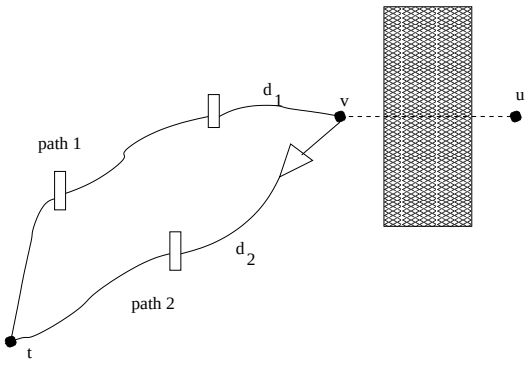

Figure 3. An example of partial routing solutions that cannot be compared for pruning.

Consider the two partial solutions from v to t in Figure 3. 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 both d_1 and d_2 must be no greater than $T_\phi - Setup(r)$, 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 to other sub-paths 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. Figure 4 shows how one can adapt the Fast Path framework to accomplish this in the Registered-Buffered Path (RBP) algorithm. We describe some of the RBP's key features:

- 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. register is added. 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.

- RBP combines Steps 4 and 5 from Figure 1 into a single Step 4. RBP has the luxury of knowing that as soon

RBP Algorithm $(G, B, s, t, m', r, T_\phi)$

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^*

Figure 4. The Registered Buffered Path algorithm.

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.

- 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 Figure 5. Of course, with blockages, the wave fronts are not 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 $O(nNk^2 logn k)$ – 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.

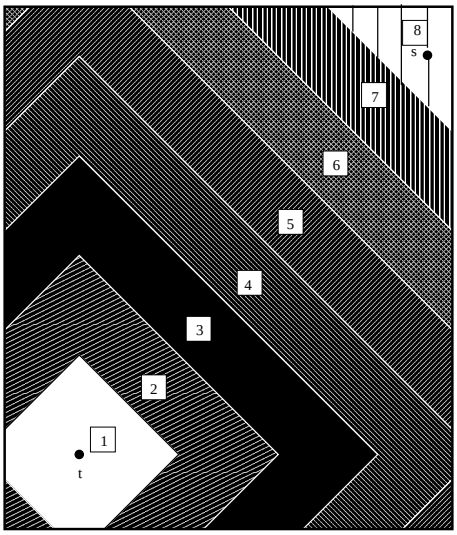

Figure 5. Wave-front expansion (as in [5]).

4 Multiple Clock Domain Routing

4.1 Background

As mentioned in Section 1, 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.

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 cycle to route the net from the MCFIFO to the sink, signals must be synchronized to the clock of each domain. Chelcea and Nowick utilize the concept of a 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. Each relay stations has auxiliary storage that allows buffering one additional data if needed.

An MCFIFO and two adjacent relay stations that could potentially be used between a source and a sink is shown

in Figure 6. The data flow is from left to right. The Stop signals flow in the opposite direction to indicate that the relay stations cannot accept additional data.

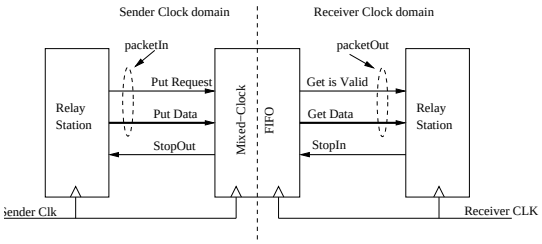

Figure 6. The Mixed Clock FIFO that interfaces two different clock domains[4].

4.2 The GALS Algorithm

Although for the Multi-Clock domain problem we are using the MCFIFO and relay-stations that require bidirectional signal flow, we abstract 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 multi clock domain problem formulation actually means the insertion of two buffers, one for each direction of signal flow.

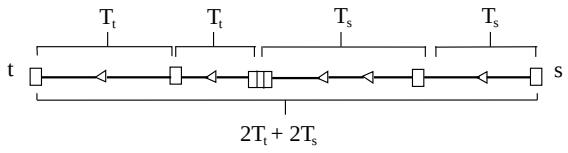

Figure 7. An example of an MCFIFO-register routing. The MCFIFO in breaks the periods into two clock domains where the period is T_s on the source side of the MCFIFO and T_t on the sink side. The total latency is $2T_s + 2T_t$.

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. Figure 7 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 multi-clock domain source-to-sink path (an MCFIFO path), we use the following conditions for feasibility: an MCFIFO path is *feasible* if and only if

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 of candidates
$\alpha = (c, d, m, v) \equiv$ Candidate at v
Output: $m \equiv$ Complete labeling of 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**
   $Q = ExtractAllMin(Q^*)$
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^*

Figure 8. The GALS Algorithm.

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

Thus, the multiple clock domain, buffered routing problem becomes:
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 for Globally Asynchronous, Locally Synchronous. There are several modifications to RBP that must be considered to obtain the GALS algorithm:

- 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.

- GALS pruning only takes place 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.

- 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 $T_s + 4T_t$. The path with the 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.

- The elements in Q are still ordered by d, but the elements in Q^* are ordered by l. We define the operation $Q = ExtractAllMin(Q^*)$ to pull all elements off of Q^* with the same latency and load them into Q. This operation extracts the next wave front of elements with equal latency from Q^*.

- 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.

Figure 8 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 $O(nNk^2 logn k)$ which is same as the RBP algorithm.

5 Experimental Results

We obtained code for the Fast-Path algorithm from the authors of [13], 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 0.07μ technology as reported by Cong and Pan [7]. 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 [5], 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 experiment investigates the behavior of the RBP algorithm as a function of the clock period. Given a grid separation of $0.125mm$ and a grid size of 200 x 200, we varied the number of registers that can be placed along the path that is separated by 159 grid edges.

Table 1. 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.

T_ϕ (ps)	Latency (ps)	Regs	Buffers	Configs	time (sec)
∞	2739	0	16	1014896	28.95
1371	2742	1	14	918078	35.41
925	2775	2	14	881092	34.84
686	2744	3	12	805603	30.90
551	2755	4	10	755814	29.55
463	2778	5	11	694386	27.50
398	2786	6	7	638676	25.46
343	2744	7	8	571877	22.88
261	2871	10	10	468975	19.02
84	3360	39	0	78122	6.57
67	4288	63	0	78246	6.59
62	4960	79	0	78278	6.63
53	8480	159	0	78360	6.55
49	15680	319	0	78416	6.44

The results are summarized in Table 1. 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. The seemingly odd choices for the clock period are the fastest clock periods required to achieve the given number of registers (rounded to the nearest picoseconds). For example, a T_ϕ of 686 is the fastest clock period that achieves a three register solution. The clock period and latencies are reported in picoseconds.

We make the following observations. First, as the clock period decreases, the number of registers along the path increase while the number of buffers decrease. Second, The number of configurations investigated (i.e., candidates popped off the queue Q in Figure 4) decreases with decreasing clock period. This empirically confirms that the run time complexity of RBP becomes more efficient as the

Table 2. GALS statistics as a function of T_s and T_t with a grid separation of 0.125 mm.

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

clock period decreases, because the space of feasible wavefront expansion in a single cycle is reduced. Finally, 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 $T\_phi = 463$.

Our second experiment explores the behavior of the GALS algorithm for different periods of the clock domain. We ran GALS using a grid separation of 0.125 *mm*. Table 2 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 $T_s = 200$ and $T_t = 300$, the algorithm places the MCFIFO close to the sink, but when $T_s = 300$ and $T_t = 200$ it places it closer to the source. 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 the previous experiment.

6 Concluding Remarks

Automated buffered routing is a necessity in modern VLSI 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 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 [13]. Experimental results validate the correctness and practicality of the two algorithms for an aggressive technology.

This work touches upon two of the routing challenges in single and multi-clock domain systems. Several more challenges remain, e.g., interconnect planning, removing registers and allowing for wave pipelining to alleviate clock distribution problems, and buffered routing with transparent latches.

Acknowledgments

The authors are grateful to Hai Zhou for supplying Fast Path code. Part of this research was funded by the National Science Foundation through CAREER grant 0093324.

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 Transactions on Computer-Aided Design*, 20(4):556–562, April 2001.

[2] L Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli. "A Methodology for Correct-by-Construction Latency Insensitive Design". In *Proc. of the IEEE International Conference on Computer-Aided Design (ICCAD)*, 1999.

[3] D. Chapiro. *Globally Asynchronous Locally Asynchronous Systems*. PhD thesis, Stanford University, 1984.

[4] T. Chelcea and S. Nowick. "Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, pages 21–6, 2001.

[5] J. Cong. "Timing Closure Based on Physical Hierarchy". In *Proceedings of the International Symposium on Physical Design*, pages 170–174, 2002.

[6] J. Cong, J. Fang, and K.-Y. Khoo. "An Implicit Connection Graph Maze Routing Algorithm for ECO Routing". In *Proc. of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 163–167, 1999.

[7] J. Cong and Z. Pan. "Interconnect Performance Estimation Models for Design Planning". *IEEE Transactions on Computer-Aided Design*, 20(6):739–752, June 2001.

[8] 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". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1999.

[9] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner. Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems. In *Twelfth Annual IEEE International ASIC/SOC Conference*, 1999.

[10] M. Lai and D. F. Wong. "Maze Routing with Buffer Insertion and Wiresizing". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, pages 374–378, 2000.

[11] J. Rabaey. *"Digital Integrated Circuits"*. Prentice Hall, 1996.

[12] J-N. Seizovic. "Pipeline Synchronization". In *IEEE ASYNC*, 1994.

[13] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz. "Simultaneous Routing and Buffer Insertion with Restrictions on Buffer Locations". *IEEE Transactions on Computer-Aided Design*, 19(7):819–824, July 2000.