Simultaneous Short-Path and Long-Path Timing Optimization for FPGAs

Ryan Fung, Vaughn Betz, William Chow
Altera Corporation
Toronto Technology Center – Toronto, Canada
{rfung, vbetz, wchow}@altera.com

Abstract

This paper presents the Routing Cost Valleys (RCV) algorithm – the first published algorithm that simultaneously optimizes all short- and long-path timing constraints in a Field-Programmable Gate Array (FPGA). RCV is comprised of a new slack allocation algorithm that produces both minimum and maximum delay budgets for each circuit connection, and a new router that strives to meet and, if possible, surpass these connection delay constraints. RCV achieves excellent results. On a set of 100 large circuits, RCV improves both long-path and short-path timing slack significantly vs. an earlier Computer-Aided Design (CAD) system that focuses solely on long-path timing. Even with no short-path timing constraints, RCV improves the clock speed of circuits by 3.9% on average. Finally, RCV is able to meet timing on all 72 Peripheral Component Interconnect (PCI) cores tested, while an earlier algorithm fails to achieve timing on all 72 cores. Keywords: timing, routing, FPGA

1 Introduction

Long-path timing optimization is an essential feature of most, if not all, modern CAD tools. In order for designs to meet performance targets, designers specify long-path timing constraints, which indicate that the longest path between certain circuit endpoints must have a delay less than some value. Examples of long-path timing constraints include frequency requirements for clocks, setup times required at circuit primary inputs (T_{SETUP}), and maximum permissible clock-to-output delays at circuit primary outputs (*maximum* $T_{CLOCK\text{-}TO\text{-}OUTPUT}$). If a CAD tool fails to satisfy all the long-path timing constraints, time consuming manual intervention is often required; users may be forced to manually synthesize, place, and/or route parts of the design to achieve the needed delays. Consequently, much research has been devoted to the study of techniques for long-path timing optimization.

Satisfying all long-path timing constraints is not sufficient to guarantee design functionality, however. For a circuit to function correctly, short-path timing constraints must also be satisfied. Short-path timing constraints specify that the minimum path delay between two circuit endpoints must be greater than some value. Short-path timing constraints not only occur between registers in a chip to guarantee there are no hold time violations within the chip, they also occur on paths from the circuit primary inputs to registers (input T_{HOLD} requirements), and on paths from registers to circuit primary outputs (*minimum* $T_{CLOCK\text{-}TO\text{-}OUTPUT}$ requirements), to guarantee correct data transfers between chips.

Generally, if a design does not meet all long-path timing constraints, the design must be run at a frequency lower than originally desired. However, if a design fails to meet its short-path timing constraints, the design will typically fail to operate at any clock frequency. Despite this fact, short-path timing optimization has received very little research attention, both in academia and industry. This resulted in designers having to manually fix short-path timing violations – a laborious process that is becoming ever more painful as designs grow larger, and as clocking structures grow more complex. As well, since fixing short-path timing violations involves adding delay to portions of the circuit, manually fixing short-path violations often creates long-path violations, resulting in lengthy design iterations.

The FPGA industry has recently recognized that requiring manual optimization of short-path timing is no longer acceptable, and recent versions of Altera's Quartus® II CAD system [1] and Xilinx's ISE CAD system [2] both incorporate optimization algorithms for short-path timing constraints. This paper presents a new algorithm, RCV, which is the first published algorithm for simultaneously satisfying both short-path and long-path timing constraints in an FPGA. RCV is the algorithm used by Altera's Quartus II software.

The algorithm described in this paper has several advantages over prior techniques. First, it removes the need for designers to manually repair short-path timing constraint violations. Second, this algorithm meets short-path constraints by inserting extra routing delay where appropriate – hence it does not require any extra logic cells. This is better than the typical manual technique where designers insert logic cells configured as buffers to slow down signals, wasting logic. Third, this algorithm simultaneously optimizes both short-path and long-path timing constraints to maximize the chance of achieving a design implementation that meets all timing constraints. Finally, when the RCV technique is applied to designs with only long-path timing constraints, it produces a 3.9% improvement in the performance of designs compared to a state-of-the-art negotiated congestion-based router. This indicates RCV is very effective at optimizing long-path timing constraints and, hence, is also useful for designs with no short-path timing problems.

This paper is organized as follows. Section 2 outlines related prior work. Section 3 provides a precise problem definition. Section 4 describes the RCV algorithm and Section 5 presents experimental results. Section 6 concludes the paper.

2 Prior Work

2.1 FPGA Techniques

Prior to this work, FPGA vendors primarily attacked the short-path problem by building special features into their FPGAs – instead of employing more general CAD algorithms. First, FPGAs include dedicated low-skew clock networks. When a clock is routed on such a network, register transfers within the clock domain will not have hold-time (short-path) violations. Second, FPGAs include programmable delay chains in each IO cell. The designer or the CAD tool sets these delay chains to slow down incoming and outgoing signals appropriately to meet short-path timing constraints at the circuit primary inputs and outputs.

These hardware solutions fall short of the needs of modern FPGAs and designs in several ways. First, today's complex FPGA designs often contain more clocks than the number of available low-skew networks. This forces some clocks to use regular routing, which introduces significant clock skew and, hence, increases the chance of a short-path timing violation. Second, the increasing magnitude of process variation and the increasing speed of FPGAs are making it more difficult to design a clock network with sufficiently low skew that all possible register transfers are free of short-path timing problems. Third, many modern designs use Phase-Locked Loops (PLLs) to generate phase- and frequency-related clocks; those designs typically expect synchronous transfers between these clock domains. This often leads to short-path timing problems. Fourth, programmable delay chains require a large amount of area, so they are typically only used to slow down signals entering and leaving the chip. This presents a problem because delay can only be added to timing paths at the point they intersect the chip periphery. Often several timing paths begin at the same input IO, pass through a common delay chain, and terminate at various registers throughout the chip. Each of those paths would "prefer" a different delay chain setting to ensure short-path timing constraints can be met while still satisfying long-path timing constraints. However, since all those paths pass through the same delay chain, a compromise is the best the programmable delay chain solution can achieve. Figure 1 illustrates the problem. If the goal is to program the delay chain to achieve IO $T_{HOLD} \leq 0$ and $T_{SETUP} \leq 3$ ns, no setting of the delay chain can meet both these timing constraints. To satisfy $T_{HOLD} \leq 0$ at Register A, the delay chain must be set to at least 1 ns, but to satisfy $T_{SETUP} \leq 3$ ns at Register B, the delay chain must be set to 0 ns (turned off).

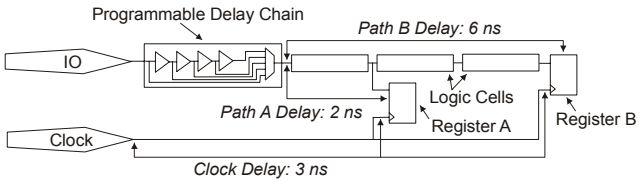

Figure 1 Example of programmable delay chain use.

2.2 ASIC Techniques

Typically, Application-Specific Integrated Circuit (ASIC) CAD tools fix short-path timing violations by inserting chains of buffers on some connections to slow down paths that are too fast. Shenoy et al [3] present two algorithms to help address the short-path buffer insertion problem – a greedy algorithm (with quadratic-time complexity in connection count) and one based on linear programming. Both algorithms are used to determine the minimum amount of delay that should be added to connections to satisfy short-path timing – without creating a long-path violation. In FPGAs, the equivalent technique inserts logic cells configured as buffers on some connections. Logic cell insertion is highly inefficient in FPGAs, however, since a significant portion of the logic capacity of an FPGA can be consumed by these logic cell buffers.

2.3 Long-Path Timing Optimization

The RCV algorithm builds on two prior long-path timing optimization algorithms: slack allocation and negotiated-congestion routing.

2.3.1 Slack Allocation

Timing constraints are specified on paths in a circuit. The end-points of paths are typically registers, primary inputs, or primary outputs – with zero or more levels of combinational logic between them. In general, however, a path can be any series of connections in a circuit. The number of possible paths in a circuit is exponential in the number of connections. Explicitly monitoring all these paths during CAD optimization would be highly inefficient, both in terms of memory and run-time. Long-path slack allocation is a well-known technique that produces a maximum delay budget for each connection in a circuit. If the design can be implemented such that each connection has a delay less than its maximum delay budget, all long-path timing constraints will be satisfied.

Various techniques have been discussed for long-path slack allocation. All these techniques rely on long-path timing analysis which computes connection slacks – where a connection slack is the minimum slack of all paths passing through that connection.

The Zero-Slack Algorithm (ZSA) is developed in [4]. ZSA starts with a set of connection delays that result in all long-path slacks being positive. It then iterates between allocating slack to increase the connection delays and performing timing analyses to update the connection slacks. During each iteration, ZSA identifies the path with the smallest positive slack and distributes the slack to the connections of the respective path by increasing the connection delays. Eventually, all the positive path slack is allocated, and every connection has zero slack. The final set of connection delays can be used as maximum delay budgets. CPU time is quadratic in the number of connections.

The Iterative-Minimax-PERT algorithm [5] improves on ZSA by introducing a faster slack allocation algorithm. This algorithm defines weights that can be used to distribute slacks non-uniformly – connections with larger weights are allocated more slack. Path weights can be computed from the connection weights, where the weight of a path is the sum of the weights of its connections. The slack allocated to each connection, *c*, is:

$$slack\_allocated(c) = \frac{slack(c) \cdot weight(c)}{max\_weight\_of\_all\_paths\_through(c)} \quad (1)$$

This technique has linear-time complexity in the number of connections because each slack-allocation iteration uses (1) to try to distribute all the remaining slack throughout the design and, in practice, only a few iterations are required to converge.

The Limit-Bumping Algorithm [6] proposes the use of connection lower delay bounds. By ensuring that the maximum delay budget of each connection is larger than its lower delay bound, many unrealizable solutions are avoided. To facilitate this, a weighting scheme is proposed that encourages removal of delay from connections that are further from their lower delay bounds. Finally, this algorithm is capable of handling problems where some connection slacks are initially negative.

2.3.2 FPGA Routing

While there are many FPGA routing techniques, only a minority of the algorithms developed explicitly analyze and optimize circuit timing, and all of those algorithms focus solely on meeting long-path timing constraints [6][7][8][9][10].

Frankle, in [6], describes a router that attempts to route connections so each has a delay less than a maximum delay budget, determined from a long-path slack allocation. Connections with maximum delay budgets closer to their lower-bound "achievable" delays are routed first. If some connections

cannot be routed with delay less than their maximum delay budgets, the corresponding maximum delay budgets are increased by 20%, and a rip-up and re-try procedure is invoked.

In [8], Ebeling et al develop the Pathfinder negotiated-congestion routing algorithm. This general algorithm has become a very successful routing technique for FPGAs, and also underlies the VPR router [9]. The academic FPGA routers with the lowest wiring requirements on a set of standard benchmarks [9][11] are based on negotiated congestion. This indicates that the negotiated congestion framework is excellent for FPGA routing, where wiring is generally quite limited.

A negotiated congestion router begins by picking a set of routing resources – wires and block input/output pins – to implement each connection. The routing resources are initially selected so each connection is routed in minimum delay, while accepting "congestion." Congestion occurs when multiple nets use the same routing resource – an electrical short, indicating an illegal routing. After the initial routing of connections in minimum delay, the router iteratively re-routes all nets encountering congestion. The router inner-loop uses a routing-resource cost to "score" the use of resources as it does a directed search through a graph representing the routing fabric from the source to the sink of a connection. The congestion component of that cost is used to gradually resolve congestion (over several routing iterations) by encouraging connections to take detours around congested resources. The delay portion of the router cost tries to keep critical connection delays to a minimum, and makes the router timing-driven. More specifically, from [9], the delay cost of a partial routing path, r, for a connection c is:

$$delay\_cost(r,c) = CRIT_{LONG\text{-}PATH}(c) \cdot T_{TERP}(r, sink(c)) \quad (2)$$

The long-path connection criticality, $CRIT_{LONG\text{-}PATH}$, indicates the importance that a connection be routed with small delay [9]:

$$CRIT_{LONG\text{-}PATH}(c) = max\left(0.99 - \frac{slack(c)}{D_{MAX}}, 0\right) \quad (3)$$

D_{MAX} is the longest path delay in the circuit. Connections with small long-path slack will tend to get $CRIT_{LONG\text{-}PATH}$ values near 1. $T_{TERP}(r,sink(c))$ is the total estimated routing path delay, which is a function both of the partial routing path, r, being considered by the router and the destination ($sink(c)$):

$$T_{TERP}(r, sink(c)) = T_{KNOWN}(r) + \alpha \cdot T_{ESTIMATE}(r, sink(c)) \quad (4)$$

The delay of the partial routing path, $T_{KNOWN}(r)$, can be computed accurately. However, the delay from the end of r to the destination, $T_{ESTIMATE}(r,sink(c))$, is not precisely known as the router evaluates (4); a look-ahead function is used to estimate how much additional delay will be incurred. Larger values of α make the search more directed, potentially at the expense of quality; many FPGA routers use values of α near 1.

The total cost of a partial routing path, r, is:

$$total\_cost(r,c) = delay\_cost(r,c) + [1 - CRIT_{LONG\text{-}PATH}(c)] \cdot total\_congestion(r,c) \quad (5)$$

This results in critical connections avoiding detours more than non-critical connections – critical connections penalize delay and ignore congestion to a greater extent.

3 Problem Formulation

We represent a circuit as a directed graph $G(V,E)$ in which each vertex, v, represents a block input pin or output pin, and each edge, e, represents either a connection, c, from a block output pin to a block input pin, or a dependency from an input pin to an output pin of a block. Each edge has an associated delay. The delays of the edges representing connections from block output pins to input pins can be altered by the FPGA placement and routing tool, while the delays of the dependency edges within blocks are generally fixed.

Timing constraints are applied to paths in G. A long-path timing constraint states that the total delay of a path must be less than some value. For example, if the user has a frequency requirement of 100 MHz for some clock, clk, the following constraint is implied for all paths from register output nodes, $sreg$, to register input nodes, $dreg$, clocked by clk:

$$T_{SLOW}(sreg,dreg) + T_{SLOW}(clk,sreg) - T_{FAST}(clk,dreg) \le 10\ ns, \quad \forall\ sreg, dreg \in clk\ clock\ domain \quad (6)$$

$T_{SLOW}(sreg,dreg)$ is the largest delay of any path from node $sreg$ to node $dreg$, while $T_{FAST}(clk,dreg)$ is the smallest path delay from the clk node to register $dreg$. Therefore, for each pair of registers in the clock domain, there are three paths that affect long-path constraint satisfaction – two clock paths and the register-to-register data path. The two clock path delays are usually similar because low-skew global networks are generally used for clock distribution. Even when that is not the case, most CAD tools attempt to control the skew on the clock paths, and hence optimization of the register-to-register path delay is sufficient to satisfy most long-path timing constraints. Consequently, in this paper, we adjust only the data-path delay and leave clock-path delays constant. Re-arranging (6) to reflect this yields:

$$T_{SLOW}(sreg,dreg) \le MAX_T(sreg,dreg), \quad \forall\ sreg, dreg \in clk\ clock\ domain, where \quad MAX_T(sreg,dreg) = T_{FAST}(clk,dreg) - T_{SLOW}(clk,sreg) + 10\ ns \quad (7)$$

A short-path timing constraint states that the delay along a path should be no less than a particular value. For example, a $T_{HOLD} = 0$ constraint on a circuit's primary inputs implies:

$$T_{FAST}(src\_io,dst\_reg) - T_{SLOW}(clk(dst\_reg),dst\_reg) \ge 0, \quad \forall\ src\_io, dst\_reg \in circuit\ G \quad (8)$$

Assuming that we optimize only the IO-cell-to-register path delays, while the clock-path delays are constant, (8) becomes:

$$T_{FAST}(src\_io,dst\_reg) \ge MIN_T(src\_io,dst\_reg), \quad \forall\ src\_io, dst\_reg \in circuit\ G, \quad where\ MIN_T(src\_io,dst\_reg) = T_{SLOW}(clk(dst\_reg),dst\_reg) \quad (9)$$

Therefore, the simultaneous short-path and long-path optimization problem can be summarized as:

$$MIN_T(src,dst) \le T_{FAST}(src,dst) \le T_{SLOW}(src,dst) \le MAX_T(src,dst), \quad \forall\ src,dst \in circuit\ G \quad (10)$$

$MIN_T(src, dst)$ is the minimum delay allowed for all data paths between the source and destination based on the designer's short-path timing constraints and the relevant clock-path delays; $MAX_T(src, dst)$ is the maximum delay allowed for all paths between the source and destination based on the designer's long-path timing constraints and the relevant clock-path delays. The goal of this work is to implement a design with connection delays that lead to the satisfaction of (10).

4 Algorithm Description

We attack the simultaneous short- and long-path timing optimization problem in two phases. First, we use a new slack allocation algorithm to convert the path-based timing constraints of (10) into connection-based delay-budget constraints. Second, we develop a new FPGA routing algorithm, which is guided by a

combination of these delay budgets and connection slacks, to meet the circuit timing constraints.

4.1 Short-Path and Long-Path Slack Allocation

The new slack allocation algorithm extends [4], [5], and [6] to consider short-path timing constraints as well as long-path timing constraints. It produces minimum delay budgets in addition to maximum delay budgets and introduces upper delay bounds to complement lower delay bounds. These upper delay bounds result in the algorithm producing better maximum delay budgets, and are essential to computing reasonable minimum delay budgets. These minimum and maximum delay budgets can be used to guide an optimization algorithm to satisfy all short-path and long-path timing constraints. While implementing each connection with a delay between its minimum and maximum delay budgets is a sufficient condition for meeting all timing constraints, it is not a necessary one. All short-path and long-path timing constraints can be satisfied with some connections violating their minimum and maximum budgets, as long as other connections achieve "sufficient margin".

For each connection from a block output to a block input, c, minimum and maximum delay budgets, D_{BUDGET_MIN} and D_{BUDGET_MAX}, are determined that satisfy the following condition:

$$D_{BOUND\_LOWER}(c) \le D_{BUDGET\_MIN}(c) \le D_{BUDGET\_MAX}(c) \le D_{BOUND\_UPPER}(c) \quad (11)$$

Both lower and upper delay bounds, D_{BOUND_LOWER} and D_{BOUND_UPPER}, are useful for modeling limits on achievable delays. For example, there may be a lower bound on a connection delay because the FPGA floorplan prevents two blocks from getting closer than a certain distance. There may be an upper bound on a connection delay because the router needs to use a dedicated resource to route a connection – in this case, the lower and upper bounds will be equal. Both delay bounds are important to provide achievable delay budgets and to avoid "wasting slack". If the maximum delay budget exceeds the upper delay bound for a connection, the slack allocated above the upper delay bound is wasted in the sense that the connection can not be implemented with that delay. It would have been better to allocate that slack to another connection; in general, the larger the separation between minimum and maximum delay budgets, the more flexibility an optimization algorithm has to satisfy timing.

4.1.1 Basic Algorithm

Figure 2 summarizes the short-path and long-path slack allocation algorithm. This algorithm calls both short-path and long-path Static Timing Analyses (STA). $D_{BOUND\_LOWER}\{C\}$ represents the set of lower-bound delays for connections in the circuit, and so on.

The algorithm starts with "temporary delays", D_{TEMP}, equal to the lower delay bounds. The maximum delay budget iterations allocate positive long-path slack, adjusting D_{TEMP} appropriately. When the iterations complete, the maximum delay budgets are set to D_{TEMP}. Note that the final maximum budgets of all connections that initially have non-positive slacks will be equal to D_{BOUND_LOWER} because only positive slack is allocated; therefore, the algorithm tries to minimize the magnitude of any unavoidable long-path violations.

Next the minimum delay budget iterations begin. Since only positive short-path slack is allocated, D_{TEMP} for each connection will never increase. This guarantees that D_{BUDGET_MIN} will be less than or equal to D_{BUDGET_MAX}. By keeping D_{TEMP} above D_{BOUND_LOWER}, the algorithm permits short-path slack to be allocated only to connections that can achieve lower delays.

```
Input: Long-path and short-path timing
     constraints, D_BOUND_LOWER{C}, and D_BOUND_UPPER{C}.
Output: D_BUDGET_MIN{C} and D_BUDGET_MAX{C}.

D_TEMP{C} = D_BOUND_LOWER{C}

/* perform maximum delay budget iterations */
iterate until stopping condition met {
  perform long-path STA using D_TEMP{C}
  allocate positive long-path slacks using
     Minimax-PERT and update D_TEMP{C}
  D_TEMP{C} = min (D_TEMP{C}, D_BOUND_UPPER{C})
}

D_BUDGET_MAX{C} = D_TEMP{C}

/* perform minimum delay budget iterations */
iterate until stopping condition met {
  perform short-path STA using D_TEMP{C}
  allocate positive short-path slacks using
     Minimax-PERT and update D_TEMP{C}
  D_TEMP{C} = max (D_TEMP{C}, D_BOUND_LOWER{C})
}

D_BUDGET_MIN{C} = D_TEMP{C}
```

Figure 2 Basic Short-Path and Long-Path Slack Allocation

Two weighting schemes were tested. The first was a unit weighting scheme. The second was a weighting scheme, similar to that used in [6], which favours adding (or removing) delay to connections that are further from their respective upper (or lower) delay bounds; those connections can better accommodate the delay change. Both schemes produced comparable final results.

The stopping condition in Figure 2 consists of two parts. First, there is an absolute limit on the number of iterations. The absolute limit ensures the algorithm has linear-time complexity in connection count – which is important for today's large designs. We found that the number of maximum delay budget iterations can be limited to 20 and the number of minimum delay budget iterations can be limited to 5 without affecting result quality. Second, the largest D_{TEMP} change in any connection is measured each iteration. When it drops below 200 ps, the iterations terminate because very little progress is being made. Stopping iterations when either of these two conditions is satisfied reduces the run-time for slack allocation by nearly 50% vs. using the first stopping condition alone, without affecting result quality. Consequently, slack allocation consumes less than 2% of the placement and routing time, on average.

4.1.2 Algorithm Enhancements

The basic algorithm does not take short-path timing into account when it determines D_{BUDGET_MAX}. Since D_{BUDGET_MIN} is less than D_{BUDGET_MAX}, for each connection, the basic algorithm can fail to find D_{BUDGET_MIN} values large enough to meet all short-path timing constraints, even if a solution exists.

Figure 3 illustrates a situation where the basic algorithm will fail to find a set of delay budgets that can satisfy the timing constraints. With the indicated lower-bound delays, the path delay from IO to Register A must be increased by at least 1.1 ns to satisfy the short-path constraint, $T_{HOLD} = 0$. Unfortunately, the logic cell and Register A are connected via a constant delay resource. The connection from the IO to the logic cell, c', is the only connection to which the slack allocator can add delay. This connection has only 2 ns of long-path slack because of the T_{SETUP} requirement of 3 ns. That means 55% of the long-path slack needs to be allocated when determining $D_{BUDGET\_MAX}(c')$ or the

algorithm will later not be able to create a sufficiently large $D_{BUDGET\_MIN}(c')$. Since there are 8 connections to which the long-path slack can be distributed, it is highly unlikely that sufficient slack will be allocated to *c'*. In fact, if the algorithm of Figure 2 is applied to this circuit, the final worst-case slacks achieved are 668 ps (T_{SETUP}) and -738 ps (T_{HOLD}) – a timing violation.

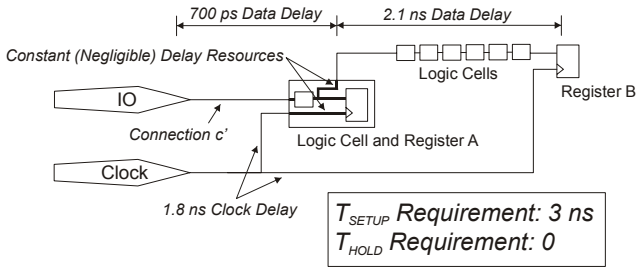

Figure 3 Example illustrating failure of the basic algorithm.

To improve the basic algorithm, we add a pre-processing step that iterates between short- and long-path slack allocation to modify the initial D_{TEMP} values. The pseudo-code in Figure 4 replaces the $D_{TEMP}\{C\} = D_{BOUND\_LOWER}\{C\}$ line in Figure 2 to improve robustness:

```
/* start of basic algorithm */

D_TEMP{C} = D_BOUND_LOWER{C}

iterate until stopping condition met {
  perform short-path STA using D_TEMP{C}
  allocate negative short-path slack using
     Minimax-PERT and update D_TEMP{C}
  D_TEMP{C} = min (D_TEMP{C}, D_BOUND_UPPER{C})

  perform long-path STA using D_TEMP{C}
  allocate negative long-path slack using
     Minimax-PERT and update D_TEMP{C}
  D_TEMP{C} = max (D_TEMP{C}, D_BOUND_LOWER{C})
}

/* continue basic algorithm */
```

Figure 4 D_{TEMP} pre-processing algorithm.

By iterating between allocating short-path and long-path negative slack, the pre-processor adjusts $D_{TEMP}\{C\}$ so that connections that need more delay, for short-path timing, have more delay before long-path positive slack allocation (in Figure 2). Notice there is only one iteration loop in Figure 4; that is, short-path negative slack may not fully allocated before long-path negative slack allocation is performed. It is unnecessary to fully allocate short-path negative slack before long-path negative slack allocation because only an adjustment of the delay starting point represented by $D_{TEMP}\{C\}$ is needed each iteration, not perfect convergence. In practice, the single loop in Figure 4 is enough to lead to good delay budgets for the routing algorithm described in 4.2; this is because the routing algorithm tries to achieve margin when satisfying the delay budgets so perfect delay-budget convergence is not essential. The stopping criteria for Figure 4 is similar to that described for the algorithm of Figure 2. The stopping condition is satisfied when either 10 iterations have been performed or the maximum D_{TEMP} change of all connections is less than 5 ps in some iteration. The iteration count restriction ensures this pre-processing algorithm also has linear-time complexity in connection count. Also, in practice, the pre-processing algorithm converges quickly and the run-time impact is negligible compared to the rest of slack allocation. Going back to the Figure 3 example, with this D_{TEMP} pre-processing step, the worst-case slacks achieved are 817 ps (T_{SETUP}) and 179 ps (T_{HOLD}) – both constraints are satisfied.

4.2 Using Delay Budgets to Guide Routing

With a few exceptions (described in Section 4.3), we found that effective optimization of short-path timing constraints can be achieved by modifying the routing algorithm alone, leaving synthesis and placement only aware of long-path timing constraints. That is, even though earlier phases make decisions that the router can not reverse, the router can almost always find an appropriate place to add delay to solve short-path violations. The router benefits from the fact that all other phases of optimization are finished, so it can model delays very accurately. Furthermore, short-path optimization in an FPGA router is effective because modern FPGA routing fabrics are relatively flexible and routing delay is a large fraction of total delay.

All elements in the FPGA general-purpose routing fabric can be used to "slow down" connections; most connections can be "slowed" dramatically (provided congestion is not a problem) by selecting spirals of routing resources. Resources, such as IO delay chains, if properly represented in the routing fabric graph, can also be selected and configured by the router to help "slow down" connections.

We use a negotiated congestion router with a modified delay cost and look-ahead function to achieve desirable routing delays.

4.2.1 Delay Portion of the Routing Cost

The delay budgets produced by the slack allocation algorithm described in Section 4.1 are used to augment the delay portion of the partial routing path cost. To generate these delay budgets, the slack allocation algorithm needs lower and upper delay bounds. An initial minimum-delay routing of all connections, ignoring congestion, provides the lower bound delays needed. The delay upper bounds for connections forced to use dedicated resources are set to the dedicated resource delays. The delay upper bounds, for other connections, are set to a very large delay (100 ns).

The delay portion of the routing cost is illustrated in Figure 5. The cost vs. total estimated routing path delay profile looks like a valley with a gently sloping bottom and steep sides. This similarity led to the algorithm's name – Routing Cost Valleys (RCV).

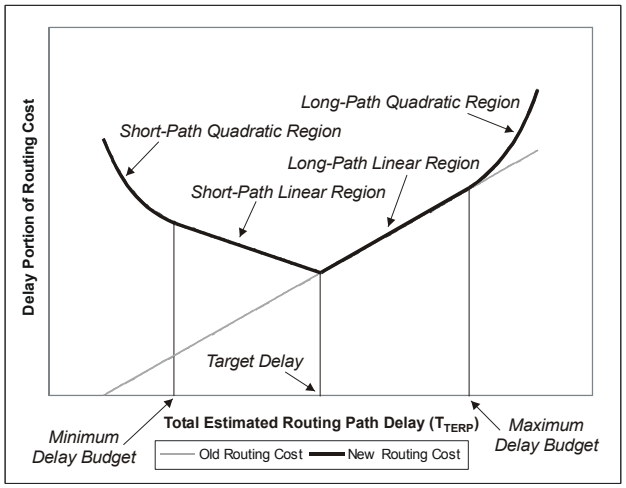

Figure 5 Illustration of the delay portion of the routing cost.

The minimum delay cost is achieved when the router achieves the "target" delay, D_{TARGET}, of a connection:

$$D_{TARGET}(c) = \min(0.5 \cdot \lfloor D_{BUDGET\_MIN}(c) + D_{BUDGET\_MAX}(c) \rfloor, \; D_{BUDGET\_MIN}(c) + 1.0\ ns) \quad (12)$$

The algorithm is restricted from always aiming for the middle of the delay budget window to avoid adding excessive delay to connections with large maximum delay budgets – in practice, a restriction of 1 ns above the minimum delay budget is sufficient. Choosing the target delay in this manner avoids wasting routing resources, which speeds up routing convergence both by limiting the scope of the graph search and by reducing congestion (preventing excessive wire use). Limiting routing resource usage also avoids unnecessary power consumption.[1]

When the anticipated total connection delay is within the delay budgets, only linear costs are seen. The slope of the line to the right of the target delay is the long-path connection criticality (between 0 and 1). We determine $CRIT_{LONG\text{-}PATH}(c)$ from a generalized version of (3) that handles the variety of timing constraints available in commercial CAD tools. The magnitude of the slope of the line to the left of the target delay is the short-path connection criticality:

$$CRIT_{SHORT\text{-}PATH}(c) = \left(\frac{D_{TARGET}(c) - D_{BOUND\_LOWER}(c)}{D_{TARGET}(c)} \right)^{\beta} \quad (13)$$

$CRIT_{SHORT\text{-}PATH}$ grows larger as more delay must be added above the lower-bound connection delay. β (> 0) is used to control how much extra emphasis the router should place on connections that need a significant amount of delay added. Larger values of β lead the router to focus more heavily on a smaller number of connections – those that need large percentage increases in delay. We found experimentally that a value of 0.5 produces good results, indicating it is best to consider most connections that need delay increase to be short-path critical.

For delays outside the delay budgets, a quadratic cost is added, on top of the linear cost, to heavily penalize such routing paths. Since costs are used to penalize delay budget violations, the delay budgets will be enforced unless there is significant congestion; in that case, congestion is resolved while sacrificing timing quality as little as possible.

The new delay cost (which replaces (2)) of a partial routing path, *r*, for connection, *c*, can be summarized as:

$$\begin{aligned} delay\_cost(r,c) = {} & CRIT_{LONG\text{-}PATH}(c) \cdot T_{TERP}(r, sink(c)) + \\ & [CRIT_{SHORT\text{-}PATH}(c) + CRIT_{LONG\text{-}PATH}(c)] \cdot \\ & \max(0, D_{TARGET}(c) - T_{TERP}(r, sink(c))) + \\ & \frac{\left(\max(0, T_{TERP}(r, sink(c)) - D_{BUDGET\_MAX}(c))\right)^2}{100\ ps} + \\ & \frac{\left(\max(0, D_{BUDGET\_MIN}(c) - T_{TERP}(r, sink(c)))\right)^2}{100\ ps} \end{aligned} \quad (14)$$

The 100 ps denominators normalize the quadratic costs relative to the linear costs. 100 ps was selected since it corresponds roughly with the smallest delay increment that can be reliably achieved in the FPGA routing fabric.

It should be noted that the delay cost formulation just described is not used for all connections. The short-path linear and quadratic costs are removed for connections that have a short-path slack of at least 1 ns with lower-bound delays. This prevents the addition of delay to achieve unnecessary short-path margin at the expense of long-path margin and CPU time.

[1] To this end, the long-path criticality is also restricted to be >= 0.1.

4.2.2 Routing Look-ahead Function

This modified router places more stringent accuracy requirements on the routing look-ahead function. In traditional negotiated congestion routers, a look-ahead function that conservatively (and systematically) underestimates delay is typical – underestimating delay increases CPU time but facilitates the search for the best routing path because the router is trying to minimize connection delay [9]. In RCV, however, there are many potential routing paths which will have similar delay cost, since we are not searching for the minimal delay routing path, but rather a routing path with a "target" delay that may be well above the minimum achievable. Therefore, for RCV, the look-ahead function must accurately estimate delays. If the function underestimates delay, the router will add delay close to the connection source, anticipating quick routing paths to the sink. Closer to the sink, however, the router will find it can not meet D_{TARGET}, because it added too much delay earlier. This will force the router to backtrack to explore lower delay paths from the source – increasing routing time. Conversely, if the look-ahead function overestimates delay, the router will pick a low-delay routing path near the source anticipating a large delay increase closer to the destination. Close to the destination, the router will realize it has arrived there using too little delay and will use a lot of resources around the sink to achieve D_{TARGET}. This increases the likelihood of congestion around the sink, which may force the router to backtrack to explore higher delay paths from the source.

We use a look-ahead function that anticipates a minimum delay routing to the destination (ignoring congestion) – the FPGA routing fabric is pre-buffered and is regular enough that minimum delay routes can be accurately predicted. For long-path critical connections, as mentioned earlier, the "optimistic" look-ahead function facilitates the search for the best routing path. For short-path critical connections, the function encourages the router to add enough delay to meet short-path constraints close to the connection source. If congestion prevents the acquisition of additional resources close to the source, the router will grab the additional resources opportunistically before it reaches the sink – minimizing the need for backtracking.

4.3 Dedicated Resource Avoidance

Sometimes synthesis or placement decisions can force certain connections to be routed using dedicated resources, which have a fixed delay. Examples of such dedicated resources are the carry chain circuitry and the dedicated look-up table to register routing in Stratix™ FPGAs [12]. When synthesis or placement forces the use of such a dedicated connection, the router has no ability to change the connection delay and, hence, no ability to solve short-path timing violations using that connection.

We modified the FPGA placement algorithm to ensure that all short-path critical paths have at least one connection to which delay can be added. This is achieved by identifying connections that: (a) are part of paths that could violate short-path timing; (b) could tolerate additional delay, without violating a long-path timing constraint; and (c) might be forced to use dedicated routing resources in some placements. After enumerating all such connections, the placer forbids any placement in which dedicated routing must be used for any of those connections.

5 Experimental Results

The experimental results from two sets of designs will be presented. The first set consists of 100 representative FPGA designs gathered from Altera customers – with all user

constraints (timing, placement, and routing) removed to avoid ambiguity in what is being measured. These designs have 3,264 to 67,311 logic elements (median of 12,072 logic elements) and target a range of Altera Stratix devices and packages [12]. The second set consists of 72 master-target 66-MHz PCI cores compiled into a range of Altera Stratix devices and packages [12], for the two fastest speed grades. Half of these cores contain only timing constraints while the other half contain timing and locked-IO constraints. These cores have 1,171 to 1,878 logic elements (median of 1,353 logic elements). PCI cores are measured because they are representative of typical FPGA customer designs with challenging IO timing.

All the experiments were run with version 4.0 of Altera's Quartus II Software [1] on 3.066 GHz Intel Pentium 4 machines. Without RCV, the Quartus II software only attempts to meet long-path constraints through most of the CAD flow. The Quartus II software only tries to address short-path constraints by setting the delay chains in the IO cells appropriately; however, as described in 2.1, this technique is not very powerful. With RCV, the Quartus II software simultaneously optimizes long-path and short-path timing during routing; the remainder of the CAD flow is unchanged, so placement is only aware of long-path constraints and "intelligent" IO delay chain setting is still performed.

No routing failures were observed while conducting the experiments below – this despite the limited routing available in an FPGA. This routing success rate is achieved because costs are used to enforce delay budgets rather than hard limits. RCV applies "pressure" to find a good routing solution for timing; however, if a design is facing routing difficulty, increasing congestion penalization gracefully "pushes" the router to sacrifice timing quality to achieve a routing solution.

5.1 Customer Design Benchmarks

5.1.1 Long-Path Results

This experiment measures the improvement in long-path results that can be achieved by replacing the traditional delay cost of a negotiated congestion router with that of the RCV algorithm. For this experiment, the Quartus II Software was instructed to optimize only clock frequency, F_{MAX} (measured by the frequency of the slowest clock in circuits with multiple clocks). Figure 6 summarizes the results. There is an average F_{MAX} improvement of 3.9%, at a run-time cost of a 35.6% extra router time, and a total placement-and-routing CPU time increase of 9.3% – including the time needed to compute delay budgets. There was an additional cost of 2.8% extra wire use; however, since no routing failures were observed, this shows that the router can leverage "available wire" in devices to achieve better timing.

An upper bound on router F_{MAX} performance can be computed from a minimum delay routing solution that ignores congestion (allowing shorts). Before RCV, the final F_{MAX} was, on average, 12.3% worse than this F_{MAX} bound. With RCV, the final F_{MAX} is, on average, only 8.3% below this bound.

The RCV delay cost is the key to these excellent results. Traditional negotiated congestion assigns a fixed criticality, or cost, per unit of delay, for each connection. The result is that non-critical connections often pay so little attention to delay that they become critical and slow the circuit. In RCV, however, once the delay of a connection goes beyond D_{BUDGET_MAX}, the router knows that this connection could now limit the speed of the circuit, and aggressively tries to avoid further delay increases. At the same time, RCV is more sophisticated than routers that simply try to route each connection in less delay than its maximum delay budget (such as [6]). In designs that are pushing the limits of FPGA speeds (for example, the design spec is "as fast as possible"), it is almost inevitable that some connections can not be routed within their delay budgets. Often, RCV is able to cover the violation of a connection delay budget by achieving delays less than D_{BUDGET_MAX} on other connections. This is achieved using the long-path criticality term in (14), which encourages delay reduction beyond that required by D_{BUDGET_MAX}, in proportion to the importance of a connection to the circuit timing.

5.1.2 IO T_{SETUP} and T_{HOLD}

This experiment measures the effectiveness of the RCV algorithm on 100 real customer designs with artificial, but "typical of common usage", timing constraints. The Quartus II Software was instructed to optimize three types of long-path timing constraints simultaneously: (i) maximize clock frequency (F_{MAX}), (ii) meet a T_{SETUP} constraint of 5.75 ns (this affects all primary input to register transfers), and (iii) meet a *maximum* $T_{CLOCK\text{-}TO\text{-}OUTPUT}$ constraint of 10 ns (this affects all register to primary output transfers). One type of short-path timing constraint was also set: meet a T_{HOLD} constraint of 0 (this affects all primary input to register transfers).

Table 1 presents the overall results. RCV improves performance on all four types of timing constraints, at the cost of 14% higher CPU time and 8.4% additional wire.

5.1.3 Register-to-Register Internal T_{HOLD}

This experiment measures how well the RCV algorithm solves T_{HOLD} violations internal to an FPGA on the set of 100 customer designs. For this experiment, the Quartus II Software was instructed to: (i) optimize clock frequency (F_{MAX}) and (ii) attempt to prevent internal T_{HOLD} violations (between registers).

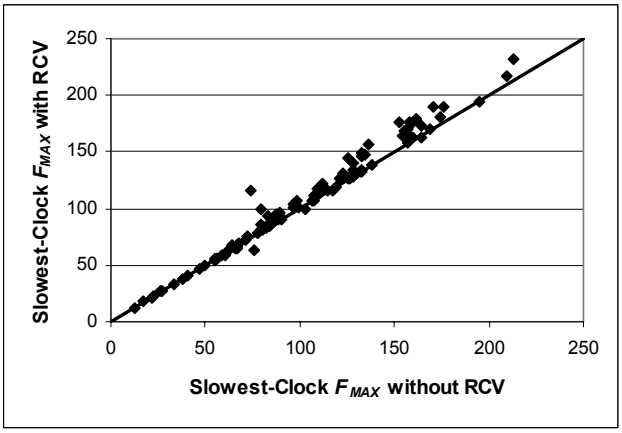

Figure 6 F_{MAX} improvement with RCV.

Table 1 Effect of RCV on 100 designs with F_{MAX} and short-path/long-path IO timing constraints.

	Without RCV	With RCV
Average F_{MAX}	87.52 MHz	91.34 MHz
Average Worst T_{SETUP} Slack	0.15 ns	0.39 ns
Average Worst $T_{CLOCK\text{-}TO\text{-}OUTPUT}$ Slack	-2.23 ns	-1.94 ns
Average Worst T_{HOLD} Slack	-0.81 ns	0.27 ns
Average Place and Route Time	741 seconds	846 seconds

Of the 100 customer designs, 18 had internal T_{HOLD} violations without RCV. All these designs had complex clocking, such as gated clocks or locally routed clocks. With RCV, only 5 of the designs had internal T_{HOLD} violations. RCV managed to achieve a 3.4% F_{MAX} improvement despite also focusing on short-path timing, but there is a placement-and-routing time increase of 20.9% and a 30.5% increase in wire used. Again, since no routing failures were observed, the router was using "available wire" to improve timing. Table 2 summarizes the internal T_{HOLD} results. Most of the small and moderate violations are repaired by RCV – only severe violations remain.

5.2 PCI Cores

PCI cores represent a highly challenging combined short- and long-path timing optimization problem, due to the many tight timing requirements on IO-to-register transfers in the PCI specification (the IO T_{SETUP} and T_{HOLD} constraints). Figure 7 shows that without RCV, the Quartus II software meets the short-path (T_{HOLD}) constraints on only 19 of the 72 PCI cores tested, and it fails to meet the long-path (T_{SETUP}) constraints on all of the cores. Figure 8 shows the comparable results with RCV enabled. All of the 72 PCI cores meet their short-path (T_{HOLD}) and long-path (T_{SETUP}) constraints – a vast improvement.

The PCI specification also includes two more long-path constraints – a 66-MHz clock frequency requirement and *maximum* $T_{CLOCK\text{-}TO\text{-}OUTPUT}$ requirements that affect register to primary output paths. The Quartus II software meets these requirements with and without RCV.

6 Conclusion

This paper introduced RCV, the first published algorithm to simultaneously optimize short-path and long-path timing constraints in FPGAs. RCV comprises a new slack allocation algorithm and a new routing formulation. The slack allocation algorithm is the first to incorporate upper delay bounds and compute minimum delay budgets. The router incorporates a new delay cost formulation, using the delay budgets from slack allocation, to enable satisfaction of both short- and long-path timing constraints, without requiring any additional FPGA logic.

Experimental results show that RCV outperforms earlier approaches used to satisfy short- and long-path timing constraints. Using only FPGA IO delay chains to solve short-path violations failed to meet the timing constraints on all 72 PCI cores tested, while RCV met the constraints on all of the cores. On a set of 100 benchmark circuits, with short- and long-path timing constraints, RCV improved the short-path T_{HOLD} and the long-path T_{SETUP} timing, on average, by 1.08 ns and 0.24 ns, respectively. On a set of 100 benchmark circuits, with no short-path timing constraints, RCV achieves 3.9% higher circuit speed than a traditional negotiated congestion router, indicating that RCV outperforms this highly successful algorithm, even on the well-studied long-path-only timing problem.

7 References

[1] "Quartus II Software", www.altera.com.

[2] "ISE Logic Design Tools", www.xilinx.com.

[3] N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli, "Minimum Padding to Satisfy Short Path Constraints," *ICCAD*, 1993, pp. 156-161.

[4] P. S. Hauge, R. Nair, and E. J. Yoffa, "Circuit Placement for Predictable Performance", *ICCAD*, 1987, pp. 88-91.

[5] H. Youssef and E. Shragowitz, "Timing Constraints for Correct Performance", *ICCAD*, 1990, pp. 24-27.

[6] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing", *DAC*, 1992, pp. 536-542.

[7] Y. S. Lee and A. Wu, "A Performance and Routability-Driven Router for FPGAs Considering Path Delays", *DAC*, 1995, pp. 557-561.

[8] C. Ebeling, L. McMurchie, S. A. Hauck, and S. Burns, "Placement and Routing Tools for the Triptych FPGA", *IEEE Trans. on VLSI*, Dec. 1995, pp. 473-482.

[9] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, 1999.

[10] S. Lee and M. Wong, "Timing-Driven Routing for FPGAs Based on Lagrangian Relaxation," *IEEE TCAD*, April 2003, pp. 506-511.

[11] "The FPGA Place and Route Challenge", http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html.

[12] "Stratix Device Family Data Sheet", www.altera.com.

Table 2 Internal T_{HOLD} violation repair with RCV.

Magnitude of Worst-case T_{HOLD} Violation			
Without RCV	With RCV	Without RCV	With RCV
0.086 ns	No Violation	3.130 ns	No Violation
0.289 ns	No Violation	3.301 ns	0.913 ns
0.374 ns	No Violation	3.562 ns	No Violation
0.403 ns	No Violation	3.887 ns	No Violation
0.418 ns	No Violation	4.009 ns	No Violation
1.103 ns	No Violation	4.898 ns	4.916 ns
1.633 ns	No Violation	5.774 ns	5.971 ns
1.862 ns	No Violation	8.928 ns	8.500 ns
2.207 ns	No Violation	19.228 ns	18.647 ns

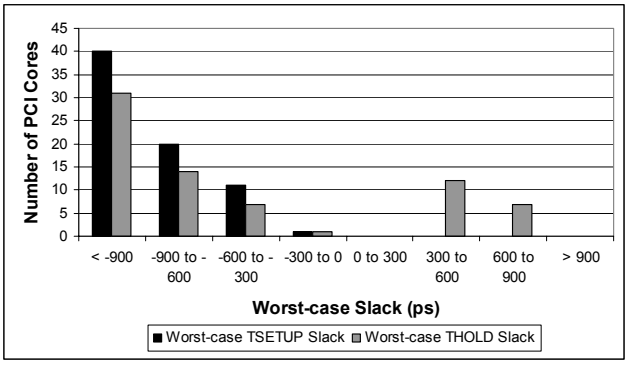

Figure 7 PCI core IO timing performance without RCV.

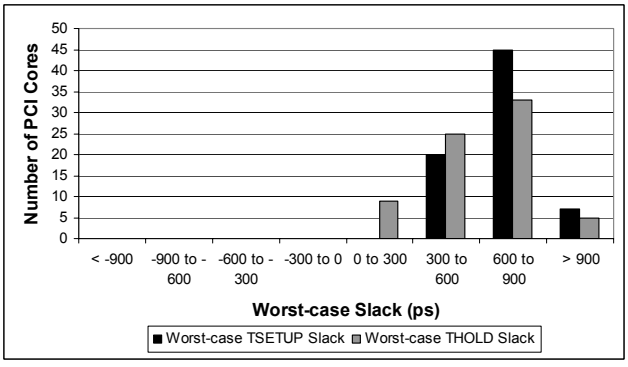

Figure 8 PCI core IO timing performance with RCV.