Slack Allocation and Routing to Improve FPGA Timing While Repairing Short-Path Violations

Ryan Fung, Vaughn Betz, *Member, IEEE*, and William Chow

***Abstract*—This work presents the first published algorithm to simultaneously optimize both short- and long-path timing constraints in a Field-Programmable Gate Array (FPGA): the Routing Cost Valleys (RCV) algorithm. RCV consists of two components: a new slack allocation algorithm that determines both a minimum and a maximum delay budget for each circuit connection, and a new router that strives to meet and, if possible, surpass these connection delay constraints. RCV improves both long-path and short-path timing slack significantly versus an earlier Computer-Aided Design (CAD) system, showing the importance of an integrated approach that simultaneously optimizes both types of timing constraints. It is able to meet long-path and short-path timing on all 157 Peripheral Component Interconnect (PCI) cores tested, while an earlier algorithm failed to achieve timing on 75% of the cores. Even in cases where there are no short-path timing constraints, RCV outperforms a state-of-the-art FPGA router and improves the maximum clock speed of circuits by an average of 3.2% (and up to 24.7%).**

***Index Terms*—FPGA, routing, slack allocation, timing**

I. Introduction

LONG-PATH timing optimization is a key feature of many academic and all commercial FPGA CAD flows. Long-path timing constraints indicate that the longest path delay between certain circuit endpoints must be less than some value in order for a design to meet its performance goals. Examples of long-path timing constraints include clock frequency requirements, setup times required at circuit primary inputs relative to some clock (T_{SETUP}), and maximum permissible clock-to-output delays at circuit primary outputs (maximum $T_{CLOCK\text{-}TO\text{-}OUTPUT}$). Long-path timing optimization for FPGAs has been extensively researched, and is crucial to achieving the best performance in a device. For example, a recent commercial FPGA CAD system achieves 50% higher circuit performance with full-effort timing-driven placement and routing than with algorithms that focus only on wirelength [1], at a cost of 5x increased run time. If a CAD tool fails to satisfy all long-path timing constraints, the designer must generally either re-design the circuit or perform some level of manual synthesis, placement and/or routing. Both of these options are very time-consuming and reduce designer productivity, underscoring the need for the highest quality automatic optimization possible.

To create a functional design, however, one must satisfy not only long-path timing constraints, but also all short-path timing constraints. Short-path timing constraints specify that the minimum path delay between two circuit endpoints must be greater than some value. Such constraints occur not only between registers in a chip to guarantee there are no hold-time violations within the chip, but also 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, it must be run at a lower frequency. However, if a design fails to meet its short-path constraints, it will fail to operate at any frequency. Despite this fact, short-path timing optimization has received little research attention in academia and, until recently, little attention in industrial FPGA CAD tools. This resulted in FPGA designers having to manually fix short-path violations – a laborious process that is becoming ever more painful as designs grow, and clocking structures increase in complexity. 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 [2] and Xilinx's ISE CAD system [3] both incorporate optimization algorithms for short-path 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, and which has been incorporated into Altera's Quartus II CAD system.

The RCV algorithm described in this paper has several advantages over prior techniques. First, it removes the need for designers to manually repair short-path violations. Second, this algorithm meets short-path constraints by inserting extra routing delay where appropriate. This wastes no logic, in contrast with the typical manual technique of inserting logic cells configured as buffers to slow down signals. Third, RCV simultaneously optimizes to meet both short-path and long-

Manuscript received March 28, 2007.

The authors are with Altera Corporation, Toronto Technology Center, Toronto, Canada (e-mails: {rfung, vbetz, wchow}@altera.com).

path timing constraints, maximizing the chance of achieving a design implementation that meets all constraints. Fourth, the algorithm includes a new technique to increase the probability of closing timing despite inaccuracy in delay estimation during optimization. Finally, the RCV algorithm outperforms prior approaches and increases design performance even when applied to the traditional long-path timing optimization problem with no short-path constraints.

This paper is organized as follows. Section II outlines related background and prior work. Section III provides a precise problem definition and Section IV describes the RCV algorithm. Section V presents experimental results. Section VI summarizes our conclusions.

II. Background and Prior Work

A. FPGA Architecture for Short-Path Optimization

Prior to this work, FPGA vendors primarily attacked the short-path problem by building special hardware 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 to meet short-path constraints at the FPGA periphery.

These hardware solutions fall short of the needs of modern FPGAs and designs. First, today's complex FPGA designs often contain more clocks than low-skew networks. This forces some clocks to use regular routing, which introduces clock skew and, hence, increases the chance of a short-path 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 at the point they intersect the chip periphery and this limits their utility. For example, several timing paths may 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 can be met while still satisfying long-path constraints. However, since all those paths pass through the same delay chain, a compromise setting must be used instead. Fig. 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[1], no setting of the delay chain can meet both constraints. To satisfy $T_{HOLD} \leq 0$ at Register A, the delay chain must be set to at least (3 ns - 2 ns) = 1 ns, but to satisfy $T_{SETUP} \leq 3$ ns at Register B, the delay chain must be set to (3 ns + 3 ns - 6 ns) = 0 ns (turned off).

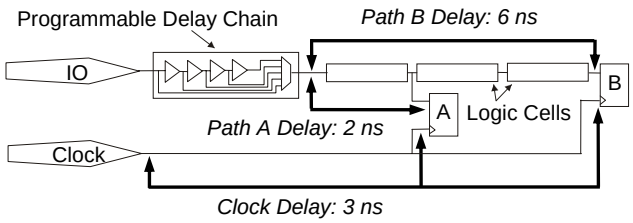

Fig. 1. Example of programmable delay chain use.

B. ASIC Techniques for Short-Path Optimization

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. Shenoy et al [4] 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 delay that should be added to connections to satisfy short-path timing without creating long-path violations. In FPGAs, the equivalent technique inserts logic cells configured as buffers on some connections; this is highly inefficient since it can consume a significant portion of the device logic. Consequently, a new approach is needed for FPGA short-path optimization.

C. Long-Path Timing Optimization

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

1. Slack Allocation

Timing constraints are specified on circuit paths. 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, a path can be any series of connections in a circuit. The number of possible paths in a circuit is exponential in connection count. Explicitly monitoring all these paths during optimization would be highly inefficient in both memory and run-time. Long-path slack allocation is a well-known technique that produces a

[1] IO T_{HOLD} and T_{SETUP} refer to timing relationships between the data and clock at the periphery of the FPGA. IO T_{SETUP} specifies the minimum amount of time the data should arrive at the FPGA before the clock. This datasheet-style specification is independent of clock frequency. In the example, the designer is promising the data will arrive ≥ 3 ns before the clock and hence is asking that the FPGA be configured so IO $T_{SETUP} \leq 3$ ns. This can be achieved by controlling the clock and data path delays within the FPGA. IO T_{HOLD} specifies the minimum time the data should be held at the FPGA after the clock. The designer is promising to hold the data until the clock and so is asking that the FPGA be configured to achieve IO $T_{HOLD} \leq 0$.

maximum delay budget for each circuit connection. If the design can be implemented so that each connection has a delay less than its maximum delay budget, all long-path constraints will be satisfied.

Various techniques have been discussed for long-path slack allocation. All these techniques rely on long-path timing analysis to compute 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 [5]. 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. In 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. Runtime is quadratic in the number of connections.

The Iterative-Minimax-PERT algorithm [6] improves on ZSA by introducing a faster allocation algorithm. This algorithm defines weights that control slack distribution – 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. Of the total path slack, *slack(c)*, the portion 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 needed to converge.

The Limit-Bumping Algorithm [7] 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 slacks are initially negative.

Techniques for optimally distributing slack using linear programming techniques and dual min-cost flow have recently been presented [8][9]. These techniques support weights like Minimax-PERT, in addition to lower and upper bounds on budgets, and balanced budget distribution. These more powerful budgeting frameworks can be more computationally intensive than Minimax-PERT.

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 constraints [7][10][11][12][14].

Frankle, in [7], 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 [11], 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 Versatile Place-and-Route (VPR) router [12]. The academic FPGA routers with the lowest wiring requirements on a set of standard benchmarks [12][13][15] 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, the router iteratively re-routes nets encountering congestion. The router inner-loop uses a routing-resource cost to "score" the use of resources during a directed search from the source to the sink of a connection, through a graph representing the routing fabric. A 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 minimize the delays of critical connections, and makes the router timing-driven. More specifically, from [12], 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 criticality, $CRIT_{LONG\text{-}PATH}$, indicates the desire that a connection be routed with small delay [12]:

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

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 *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 ones – critical connections penalize delay and ignore congestion to a greater extent.

D. Long-Path and Short-Path Timing Optimization

In [16], we described an earlier version of the Routing Cost Valleys (RCV) algorithm. In this paper, we describe the RCV algorithm in more detail, and present significant algorithm enhancements, most notably path-level guardbanding (in Section IV.A.4) and a revised routing delay cost formulation (in Section IV.B.1). These changes reduce CPU time, improve timing closure on difficult cases, and reduce the additional routing wire needed to repair short-path violations by a factor of 3 to 7 on average (depending on the types of timing constraints) versus that described in [16]. The experimental results have been completely re-done with a more recent version of the Quartus II CAD system, and are more extensive, incorporating both additional experiments and results for the more recent Stratix II FPGAs.

III. 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 250 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*:

$$T_{SLOW}(sreg,dreg) + T_{SLOW}(clk,sreg) - T_{FAST}(clk,dreg) \le 4\ ns, \quad \forall\ sreg,dreg \in clk\ clock\ domain \quad (6)$$

$T_{SLOW}(sreg,dreg)$ 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, there are three paths that affect long-path timing – 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 the general routing fabric is used to construct the clock tree, most CAD tools attempt to control the skew on the clock paths, and optimization of the register-to-register path delay is sufficient to satisfy most long-path timing constraints. Consequently, in this work, we adjust only the data-path delay and leave clock-path delays constant. Re-arranging (6) to reflect this:

$$T_{SLOW}(sreg,dreg) \le MAX_T(sreg,dreg), \quad \forall\ sreg,dreg \in clk\ clock\ domain, \quad where\ MAX_T(sreg,dreg) = T_{FAST}(clk,dreg) - T_{SLOW}(clk,sreg) + 4\ ns \quad (7)$$

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

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

where T_{SLOW} *(clk(dst_reg), dst_reg)* is the longest delay between the clock source and the destination register. Assuming that we optimize only the IO-cell-to-register path delay, while the clock-path delay is constant, (8) becomes:

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

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 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 between the source and destination based on the designer's long-path 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).

IV. 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.

A. Short-Path and Long-Path Slack Allocation

The new slack allocation algorithm extends [5], [6], and [7] to consider short-path constraints as well as long-path timing. It introduces minimum delay budgets in addition to maximum delay budgets and respects both lower and upper delay bounds. These minimum and maximum budgets can be used to guide an optimization algorithm to satisfy all short-path and long-path constraints. While implementing all connections to satisfy their minimum and maximum budgets is a sufficient condition for meeting all timing constraints, it is not a necessary one. All short-path and long-path constraints can be satisfied with some connections violating their 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 computed and satisfy the following condition:

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

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 help create achievable delay budgets and avoid "waste" of slack. For example, if the minimum budget exceeds the upper bound for a connection, the connection will not be able to meet its budget and this can lead to a short-path failure. Similarly, long-path failures can occur if maximum budgets are less than lower bounds. In terms of "wasting" slack, if the maximum budget exceeds the upper bound for a connection, the connection will meet its budget; however, the long-path slack allocated above the upper bound is "wasted" in the sense that the connection can not be implemented with that delay. A similar waste of short-path slack occurs when minimum budgets are less than lower bounds. It would have been better to allocate slack to other connections because the larger the separation between minimum and maximum delay budgets, the more flexibility an optimization algorithm has to satisfy timing.

1. Basic Algorithm

Fig. 2 summarizes the 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.

```
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}
```

Fig. 2. Basic short-path and long-path slack allocation.

The algorithm starts with "temporary delays", D_{TEMP}, equal to the lower delay bounds. The maximum delay budget iterations allocate positive long-path slack according to the Minimax-PERT algorithm of [6] to increase D_{TEMP}. 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.

Two weighting schemes were tested. The first was a unit weighting scheme. The second was a weighting scheme, similar to that used in [7], 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 Fig. 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 7 and the number of minimum-delay-budget iterations can be limited to 3 without affecting result quality. Second, the largest D_{TEMP} change in any connection is measured each iteration. When it drops below 800 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% versus using the first stopping condition alone, without affecting result quality [16]. Consequently, slack allocation averages less than 6% of the placement-and-routing time, even when the CAD tool is optimizing challenging setup-and-hold constraints (as described in Section V.A.3), which is the most CPU-intensive mode.

The techniques of [8] and [9] could be used instead of the Minimax-PERT algorithm to compute better delay budgets. However, the insensitivity to the weighting scheme used and the relatively aggressive stopping criterion imply that this application is unlikely to require this, and so the more computationally intensive approaches may not be appropriate.

2. Delay Preprocessing

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 constraints, even if a solution exists.

Fig. 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 (1.8 ns - 0.7 ns) = 1.1 ns to satisfy the short-path constraint, $T_{HOLD} \leq 0$. The logic cell and Register A are connected via a constant delay resource with negligible delay; the upper and lower delay bounds for this resource will be set to 0, which will correctly prevent slack from being allocated to this connection. The connection from the IO to the logic cell, *c'*, is the only connection to which delay can be added. This connection has only (1.8 ns + 3 ns - 0.7 ns - 2.1 ns) = 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 to $D_{BUDGET\_MAX}(c')$ 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 unlikely that sufficient slack will be allocated to *c'*. In fact, if the algorithm of Fig. 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.

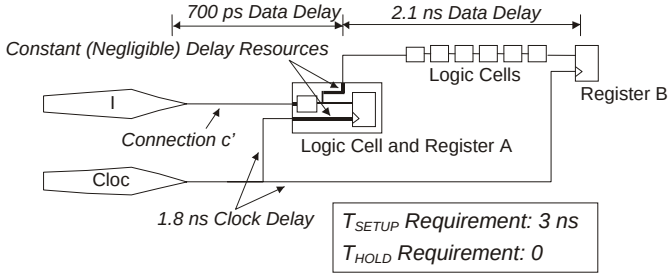

Fig. 3. Example illustrating failure of the basic algorithm.

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 Fig. 4 replaces the $D_{TEMP}\{C\} = D_{BOUND\_LOWER}\{C\}$ line in Fig. 2:

```
/* start of basic algorithm */
DTEMP{C} = DBOUND_LOWER{C}
iterate until stopping condition met {
  perform short-path STA using DTEMP{C}
  allocate negative short-path slack using Minimax-
     PERT and update DTEMP{C}
  DTEMP{C} = min (DTEMP{C}, DBOUND_UPPER{C})

  perform long-path STA using DTEMP{C}
  allocate negative long-path slack using Minimax-
     PERT and update DTEMP{C}
  DTEMP{C} = max (DTEMP{C}, DBOUND_LOWER{C})
}
/* continue basic algorithm */
```

Fig. 4. D_{TEMP} pre-processing algorithm.

By iterating between allocating short- 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 Fig. 2). Notice there is only one iteration loop in Fig. 4; that is, short-path negative slack may not be fully allocated before long-path negative slack allocation is performed. It is unnecessary to fully allocate short-path negative slack before proceeding because only an adjustment of the delay starting point represented by $D_{TEMP}\{C\}$ is needed each iteration, not perfect convergence. This is especially true since both slack allocation (Section IV.A.4) and the router try to achieve margin. In practice, the single loop in Fig. 4 is enough to lead to good delay budgets for the routing algorithm (Section IV.B) and saves run-time. The stopping criteria for Fig. 4 is similar to that in Fig. 2. The stopping condition is satisfied when either 7 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. 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 Fig. 3 example, with this D_{TEMP} pre-processing step, the worst slacks achieved are 817 ps (T_{SETUP}) and 179 ps (T_{HOLD}) – both satisfied.

3. *Post-Basic Algorithm Processing*

As mentioned previously, at the end of the basic algorithm, D_{BUDGET_MIN} will be greater than or equal to D_{BOUND_LOWER}. Enforcing this during short-path slack allocation in the basic algorithm is advantageous because it ensures that short-path slack is never "wasted" by allocating it to connections that can not be implemented with lower delay. However, this restriction is disadvantageous because ultimately the delay budgets will be used to guide routing. Since the routing algorithm can not explore every routing possibility, and, practically, the routing algorithm will try to meet the delay budgets with some margin, the router may end up routing all connections with slightly-greater-than-minimum delay – even those not in danger of failing short-path timing. To address this issue, another enhancement performs additional minimum-delay-budget iterations allocating short-path slack, after the basic algorithm is complete.

```
/* end of basic algorithm */

iterate until stopping condition met {
  perform short-path STA using DTEMP{C}
  allocate positive short-path slacks using Minimax-
     PERT and update DTEMP{C}
  DTEMP{C} = max (DTEMP{C}, -1.0 ns)
}
DBUDGET_MIN{C} = DTEMP{C}
```

Fig. 5. Post-basic algorithm short-path slack allocation.

Fig. 5 allows D_{TEMP} to go below D_{BOUND_LOWER} to ensure the minimum budgets more accurately reflect the absolute minimum delays necessary to meet short-path timing, so there is little danger of wasting routing resources. D_{TEMP} is kept above -1.0 ns to prevent "wasting" short-path slack on connections that are already guaranteed to be routed in

minimum delay, while not allocating enough to others.[2] The stopping criteria for Fig. 5 is similar to that of Fig. 2; it is satisfied when either 3 iterations have been performed or the maximum D_{TEMP} change of all connections is less than 800 ps in some iteration. When this technique is applied, the amount of routing wire used when solving register-to-register internal T_{HOLD} violations (Section V.A.3) is reduced by 21% and the placement-and-routing time is reduced by 2.6%. Even though the extra iterations take some time to perform, the easier routing problem more than compensates.

4. Path-level Guardbands

Once the connection-level delay budgets are computed, they can be used to guide a connection-based optimization algorithm, such as a negotiated congestion router as described in Section IV.B. Connection-based algorithms can try to achieve timing margin by not accepting solutions where connection delays are close to the delay budgets. For example, the router can attempt to achieve 1 ns margin above each connection's minimum delay budget. This is important because optimization algorithms often use approximate delay models to keep run-time reasonable.

A problem with connection-level guardbands is that some connections may have very tight delay budget windows (where margin can not be achieved), and other connections may have very wide delay budget windows. A connection-based algorithm cannot readily know the additional margin it should attempt to achieve on connections with wide delay budget windows to compensate for narrow windows elsewhere. As well, significantly guardbanding the minimum delay budgets for all connections with wide delay budget windows will waste routing resources to achieve timing margin that may not be required. Guardbanding is a path-based problem that is better solved during slack allocation. To achieve this, the appropriate guardbands are applied when computing short-path and long-path timing slacks for slack allocation. Both absolute and fraction-of-timing-requirement guardbands are applied according to the accuracy of the delay estimates for the FPGA being targeted. In Stratix II, for example, the short-path guardband consists of 250 ps plus 10% of the timing constraint. By applying these guardbands, the slack allocation algorithm can determine a set of delay budgets that achieve the necessary margin on a path-level. Sections V.A.4 and V.B show that this path-level guardbanding technique is very effective at helping RCV to meet difficult constraints.

B. Using Delay Budgets to Guide Routing

With a few exceptions (described in Section IV.C), we found that effective optimization to meet short-path timing constraints can be achieved by modifying the routing algorithm alone, leaving synthesis and placement only aware of long-path constraints. That is, even though earlier phases make decisions that the router can not reverse, the router can almost always find a way to add delay to solve short-path violations. The router benefits from the fact that most other phases of optimization are complete, so it can model delays more 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 (if routing congestion is not a problem) by selecting spirals of resources. Representing delay chains in the routing graph allows them to be selected and configured to help "slow down" connections as well.

We use a negotiated-congestion router with a modified delay cost and look-ahead function.

1. Delay Portion of the Routing Cost

The delay budgets produced by the slack allocation algorithm described in Section IV.A are used to augment the delay portion of the partial routing path cost. To generate these budgets, the slack allocation algorithm is run once before the core routing algorithm begins. The slack allocation, as discussed previously, requires lower and upper delay bounds. An initial minimum-delay routing of all connections, ignoring congestion, provides the lower-bound delays needed. The upper delay bounds for connections forced to use dedicated resources are set to the dedicated resource delays. The upper delay bounds, for other connections, are set to a large delay (100 ns).

The delay portion of the routing cost is illustrated in Fig. 6. 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).

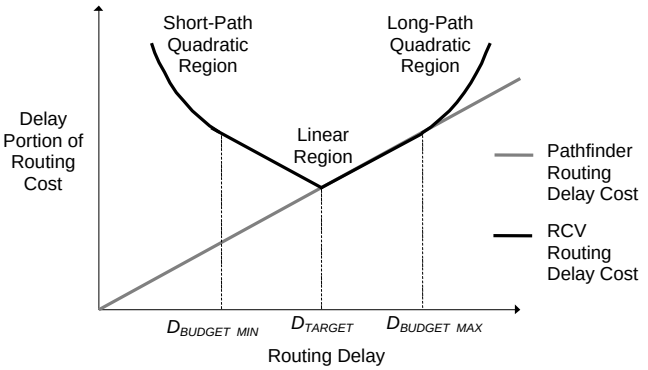

Fig. 6. RCV's delay cost compared to Pathfinder's delay cost.

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

$$D_{TARGET}(c) = \min\left(0.5 \cdot \left[D_{BUDGET\_MIN}(c) + D_{BUDGET\_MAX}(c)\right], D_{BUDGET\_MIN}(c) + 0.1\,ns\right) \quad (12)$$

[2] The RCV algorithm is not very sensitive to the precise value of this bound because the basic algorithm generally distributes most of the slack with little "waste" between the connections. We choose -1 ns because it is slightly less than any routing delay we can achieve.

The target delay is highly skewed towards the minimum delay budget – D_{TARGET} will be at most 0.1 ns above D_{BUDGET_MIN}. Our earlier approach in [16] was less aggressive, and only ensured D_{TARGET} would be within 1 ns of D_{BUDGET_MIN}. This change is possible because the minimum delay budget already reflects a path-timing guardband (Section IV.A.4), so minimal additional margin is needed. The advantage of aiming for a delay close to the minimum budget is that routing utilization is minimized and long-path timing margin is maximized. This improves the likelihood of satisfying the often more challenging long-path timing problem. Convergence speed is also improved by limiting the scope of the graph search and the amount of congestion from excessive wire use.[3] Limiting routing resource usage also avoids unnecessary power consumption. The algorithm still aims for slightly above the minimum delay budget to improve the likelihood the minimum delay budget is satisfied. Since the router does not explore every solution and delay is quantized in an FPGA, including some margin in D_{TARGET} (here 100 ps above D_{BUDGET_MIN}) is still desirable.

When the anticipated total 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 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 criticality:

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

$CRIT_{SHORT\text{-}PATH}$ 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 increase focus 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 budget violations, the 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:

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

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 in the delay cost formulation just described, the short-path linear and quadratic costs are not applicable for connections that have lower-bound delays that are larger than their target delays. In those cases, the minimum budget is trivially satisfied, and the short-path linear and quadratic costs are removed to avoid any runtime penalty during routing exploration and cost computation.

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 runtime but facilitates the search for the best routing path because the router is trying to minimize delay [12]. 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 should 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 in anticipation of 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 considerable routing near 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). Since the routing fabrics in recent FPGAs are quite regular, minimum delay routes can be accurately predicted. For long-path critical connections, as mentioned earlier, this "optimistic" look-ahead function facilitates the search for the best routing path. For short-path critical connections, the

[3] The long-path criticality is also restricted to be ≥ 0.1 so that, even for connections with easy-to-meet long-path targets, the router is still encouraged by the delay cost to explore and find minimum delay (resource) solutions.

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 obtain the additional resources opportunistically before it reaches the sink, which minimizes the need for backtracking to find additional resources.

If there is significant congestion on the fastest routing near the sink our optimistic look-ahead function will result in backtracking to find faster routes from the source to the sink vicinity, increasing CPU time. We have not seen excessive CPU times due to this phenomenon, but in FPGAs where the fastest routing can become saturated this effect could be significant and would motivate research into more sophisticated look-ahead functions that anticipate where to best obtain extra delay for short-path critical connections. The quadratic terms in the RCV delay cost (14) ensure that we will backtrack extensively before accepting a route that significantly violates our delay budgets, so the impact of look-ahead function errors is mostly a run-time penalty, rather than a result quality degradation.

3. *Minimum Delay Budget Relaxation*

Some designs which need very extensive short-path timing repair require the insertion of a large number of routing resources to obtain enough data-path delay to fix all violations. If this extra routing demand is large compared to the number of routing resources in the FPGA, it may not be possible to repair the violations and successfully route the design. By using a cost-based delay-budget formulation we ensure the router will violate some delay budgets to obtain a legal routing once the cost of congestion is high enough. However, letting negotiated congestion resolve these routability problems is slow, as it can require many routing iterations before the cost of congested resources overpowers the delay budgets.

We modified the FPGA routing algorithm to detect if it is converging very slowly by looking at the rate of decrease of the number of congested resources. If the average rate of convergence (using a geometric fit) is much slower than that profiled on typical designs, the router is having trouble. In that case, the routing algorithm first tries to reduce the minimum delay budgets of all connections that have been congested for the last 3 routing iterations and that satisfy the following criterion:

$$D_{BUDGET\_MIN}(c) - D_{BOUND\_LOWER}(c) \geq 1\,ns\ . \quad (15)$$

That is, connections which consistently have illegal routes and are demanding highly circuitous routes have their minimum budgets relaxed to aid convergence. If the routing algorithm still appears to be converging slowly, because highly excessive short-path repair is required, all minimum budgets are removed. In those extreme cases, there is usually a systematic problem with the design, and the designer should re-examine his or her clocking strategy and timing constraints to reduce or remove the short-path problems.

C. *Dedicated Resource Avoidance*

Some synthesis and placement decisions can force connections to be routed via fixed-delay dedicated resources. Examples of such dedicated resources are the carry chain circuitry and the dedicated look-up table to register routing within the Stratix FPGA logic cell [17]. When synthesis or placement forces the use of such dedicated resources, the router has no ability to insert delay and, hence, no ability to fix short-path violations using the respective connections.

We modified the 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 a part of paths that could have irreparable short-path violations; (b) could tolerate additional delay, without violating a long-path constraint; and (c) might be forced to use dedicated resources in some placements. Placements, in which dedicated routing must be used for these connections, are forbidden.

V. Experimental Results

The experimental results from two sets of designs will be presented. The first set consists of 200 representative FPGA designs gathered from Altera customers, with all user constraints (timing, placement, and routing) removed to avoid ambiguity in what is being measured. 100 of these designs have 6,663 to 87,377 logic cells (median of 17,979 logic cells) and target Altera Stratix devices [17]. The other 100 of these designs have 3,004 to 90,854 logic cells (median of 16,028 logic cells) and target Altera Stratix II devices [17]. The second set consists of 157 master-target 66-MHz PCI cores compiled into a range of Altera devices, packages, and speed grades [17]. All these cores are timing constrained according to the PCI specification. Table 1 summarizes the cores tested. PCI cores are measured because they are representative of typical FPGA customer designs with challenging IO timing.

Table 1
Summary of PCI Cores Tested

Device Family	Interface Width	Number of Logic Cells	Number of Cores
Stratix	32-bit	1108	38
Stratix	64-bit	1521	30
Cyclone	32-bit	1150	18
Cyclone	64-bit	1564	10
Stratix GX	64-bit	1521	8
Stratix II	64-bit	1492	17
Cyclone II	32-bit	1103	22
Cyclone II	64-bit	1505	14

All the experiments were run with version 6.0 of Altera's Quartus II Software [2] 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; it only addresses short-path constraints by setting the delay chains in the IO cells appropriately; however, as described in Section II.A, this technique is not very powerful.

With RCV, both long- and short-path timing are simultaneously optimized during routing and the remainder of the CAD flow is unchanged, so placement is only aware of long-path constraints and IO delay chain setting is still performed.

No routing failures were observed in any of the experiments, 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 solution.

A. Customer Design Benchmarks

1. Maximum Clock Frequency (F_{MAX})

This experiment measures the improvement in long-path results that can be achieved by replacing the traditional delay cost of a negotiated-congestion routing algorithm with that of RCV. We make our comparisons using the Quartus II router, which is based on the Pathfinder negotiated-congestion routing algorithm [11]. The Quartus II router achieves high-quality results, and even without the RCV enhancements outperforms the widely-used VPR router [12] in terms of long-path circuit timing by 2%.

For this experiment, the Quartus II Software was instructed to optimize only clock frequency, F_{MAX}. In circuits with multiple clocks, we measure the geometric average of the achieved frequency of all the clocks. Fig. 7 shows that RCV consistently improves F_{MAX} for both Stratix and Stratix II devices.[4] On average, RCV improves F_{MAX} by 3.2% at a cost of 43.4% extra router time. The increase in the total placement-and-routing runtime is only 5.3%, including the time needed to compute delay budgets. RCV increases wire use by 0.7%; however, since no routing failures were observed, it is clear that the router is leveraging only the wire available to achieve better timing.

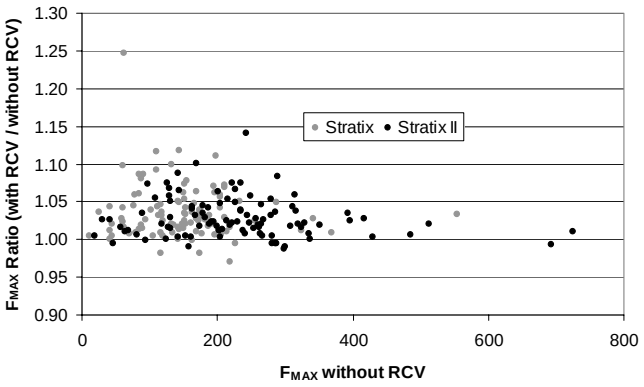

Fig. 7. F_{MAX} improvement with RCV.

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 [7]). 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.

2. IO T_{SETUP} and T_{HOLD}

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

TABLE 2
EFFECT OF RCV ON 200 DESIGNS WITH F_{MAX} AND SHORT-PATH/LONG-PATH IO TIMING CONSTRAINTS

	Stratix		Stratix II	
	Without RCV	With RCV	Without RCV	With RCV
Geometric Average F_{MAX} (MHz)	127.7	131.9	197.3	202.4
Arithmetic Average Worst T_{SETUP} Slack (ns)	0.125	0.346	1.027	1.089
Arithmetic Average Worst T_{CO} Slack (ns)	-2.356	-2.262	-0.158	-0.113
Arithmetic Average Worst T_{HOLD} Slack (ns)	-1.446	0.138	-0.633	0.636
Geometric Average Place-and-Route Time (minutes)	21.1	23.1	18.3	19.6

Table 2 presents the results. RCV improves performance on all four types of timing constraints, at the cost of 8.5% higher placement-and-routing time and 2.8% additional wire.

[4] Comparable improvements are seen with the Stratix III architecture.

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 200 designs. For this experiment, the Quartus II Software optimized: (i) clock frequency (F_{MAX}) and (ii) internal T_{HOLD} timing (between registers).

Of the 200 customer designs, 48 had internal T_{HOLD} violations without RCV. All these designs had complex clocking, such as gated clocks. With RCV, 17 of the designs had internal T_{HOLD} violations. RCV managed to achieve a 3.0% F_{MAX} improvement despite also focusing on short-path timing, but there was a place-and-route time increase of 12.7% and a 3.5% increase in wire. The wire increase was highly design dependent. For designs that did not have internal T_{HOLD} violations without RCV, there was a 1.1% wire increase as the router tried to improve the short-path margin of those designs. For designs that did have internal T_{HOLD} violations without RCV, there was a 9.2% wire increase as the router attempted to repair violations. Again, since there were no routing failures, the router used "available wire" to improve timing.

TABLE 3
INTERNAL T_{HOLD} VIOLATION REPAIR WITH RCV
(MAGNITUDE OF WORST T_{HOLD} VIOLATION)

Without RCV (ns)	With RCV (ns)	Failure Reason *	Without RCV (ns)	With RCV (ns)	Failure Reason *
27.96	29.18	RL	3.27	No Violation	
18.56	18.58	DR	3.24	No Violation	
12.61	12.82	RL	3.14	No Violation	
11.62	11.51	RL	3.13	No Violation	
10.45	9.86	DR	3.06	No Violation	
10.24	9.58	RL	2.99	No Violation	
9.67	9.47	RL	2.78	No Violation	
7.92	7.64	RL	2.71	No Violation	
6.31	6.22	RL	2.40	No Violation	
6.10	6.08	RL	2.38	No Violation	
5.79	5.41	RL	2.34	No Violation	
5.15	5.20	DR	2.12	No Violation	
4.54	4.48	RL	1.91	No Violation	
4.51	4.81	RL	1.91	No Violation	
4.39	No Violation		1.85	No Violation	
4.29	No Violation		1.78	No Violation	
4.21	No Violation		1.61	No Violation	
4.11	No Violation		1.57	No Violation	
4.08	No Violation		1.49	No Violation	
4.06	2.85	DR	1.34	1.35	DR
4.03	No Violation		1.25	No Violation	
3.91	No Violation		0.99	No Violation	
3.73	No Violation		0.34	0.20	DR
3.29	No Violation		0.21	No Violation	

* RL: Routing Limited, DR: Dedicated Routing Connection

Table 3 summarizes the internal T_{HOLD} results. Most of the small and moderate violations are repaired by RCV – mainly severe violations remain. All the violations that remain are not repaired either because the router runs out of wire (routing-limited) or the hold violations occur on a dedicated routing path. In the routing-limited cases, since all the designs do route, the router either gracefully reduces short-path optimization effort to achieve a legal routing, or it completely gives up on short-path optimization on certain connections if the wiring demands are unrealistically large (Section IV.B.3). In the dedicated routing cases, synthesis or placement decisions are made so that short-path critical connections are forced to use dedicated routing and the router has no option to improve the timing of those paths. The techniques described in Section IV.C can address these cases, but as of yet, they have not been extended to address the rarer logic and routing topologies in some of these designs.

4. Meeting Aggressive Timing Constraints

This experiment measures the ability of a placement-and-routing algorithm to consistently meet aggressive long-path timing constraints. For this experiment, the Quartus II Software is run multiple times for each design. In the first run, it is instructed to optimize clock frequency, F_{MAX}, and the performance of each clock in every design is measured. In subsequent runs, constraints are applied to all the clocks of every design. For each clock, the constraint applied is a fraction of the operating frequency measured from the first run. The number of clock domains which pass timing are measured for each fraction. This whole process is repeated without RCV, with RCV without path-level guardbanding, and with RCV with path-level guardbanding.

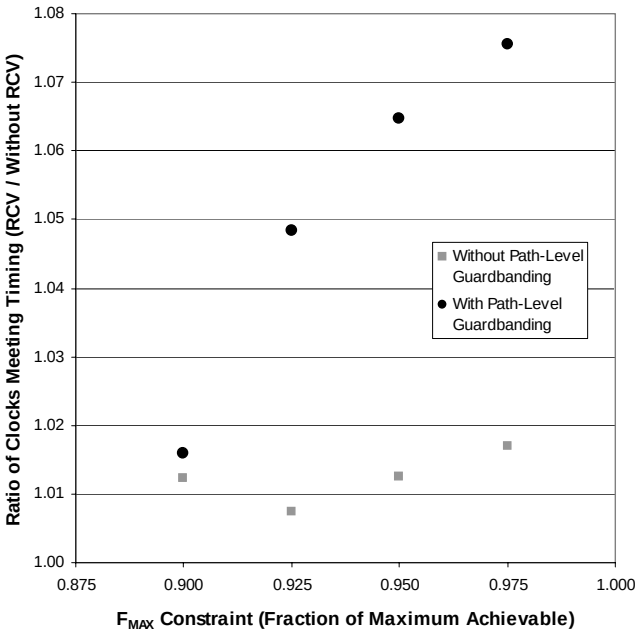

Fig. 8. Meeting aggressive timing constraints with RCV.

The results are shown in Fig. 8. It is important to note that since RCV improves F_{MAX} by 3.2%, the "with RCV" cases actually have more aggressive frequency targets to meet.[5] Nevertheless, even without path-level guardbanding, RCV is able to meet timing more consistently (about 1.2% improvement). The delay budgets help guide the connection-level router to make tradeoffs which do not compromise path-

[5] Both "with RCV" cases have comparable frequency targets to meet.

level timing performance; this helps the router meet path-level timing constraints more consistently. With path-level guardbanding, there is, on average, a 5.1% improvement in the number of clocks which meet timing, and there is a trend towards more improvement with more aggressive constraints. This trend makes sense since delay modeling inaccuracies are more likely to make highly timing marginal clocks fail timing and path-level guardbanding helps the router avoid this by strategically achieving more margin where it is beneficial. The results clearly show that RCV not only increases the maximum speed at which circuits can operate, but also improves the consistency with which the router closes timing under difficult constraints.

B. 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 (IO T_{SETUP} and T_{HOLD} constraints). Fig. 9 shows that without RCV, the Quartus II software meets the short-path (T_{HOLD}) constraints on only 40 of the 157 PCI cores tested, and meets the long-path (T_{SETUP}) constraints on 79 of the 157 cores. Fig. 10 shows the comparable results with RCV enabled. All of the 157 PCI cores meet their short-path (T_{HOLD}) and long-path (T_{SETUP}) constraints – a vast improvement.

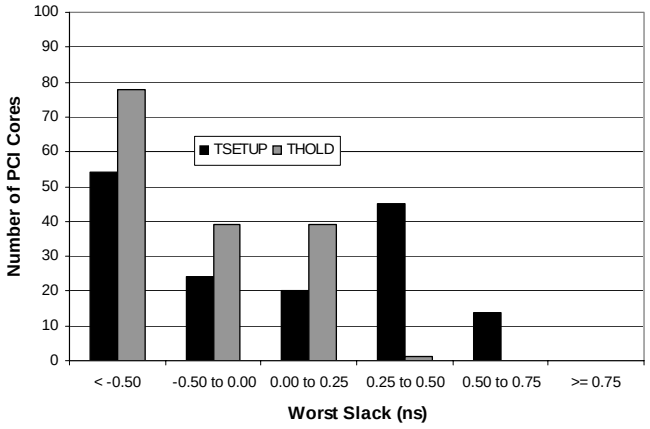

Fig. 9. PCI IO timing without RCV.

Another result of interest is illustrated in Fig. 11. It shows the performance of RCV on these PCI cores without the path-level guardbanding technique described in Section IV.A.4. For these results, only connection-level delay budget margin is used to guide the router, as described in [16]. Even though that approach aims for balanced long-path and short-path margin on a connection-basis, with up to 1 ns of short-path margin per connection, 62% of the designs have less than 500 ps of short-path margin and 39% of the designs have less than 500 ps of long-path margin. With path-level guardbanding, these percentages are 6% and 20%, respectively. This illustrates the effectiveness of path-level guardbanding on the combined short- and long-path timing optimization problem.

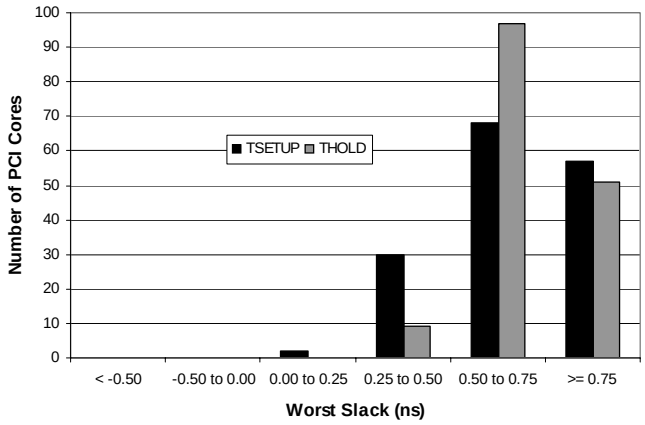

Fig. 10. PCI IO timing with RCV.

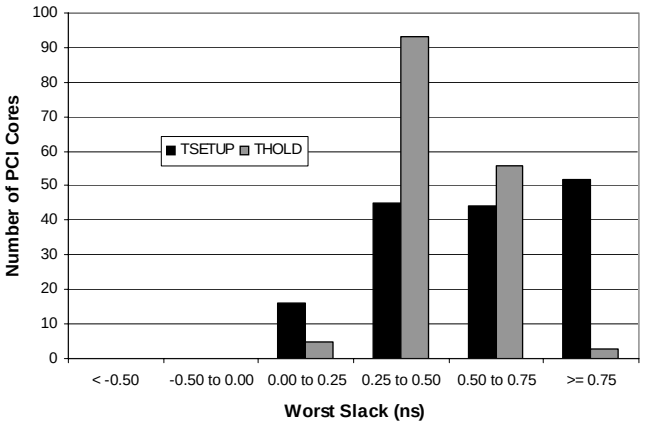

Fig. 11. PCI IO timing with RCV, without path-level guardbanding.

VI. CONCLUSION

This paper introduced RCV, the first published algorithm to simultaneously optimize considering short- 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. This algorithm also employs guardbands to account for delay estimation errors during routing. The router uses 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 try to solve short-path violations resulted in timing failures in 75% of 157 PCI cores tested, while RCV met the constraints on all of the cores. On a set of 200 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 1.43 ns and 0.14 ns, respectively. On a set of 200 benchmark circuits, RCV achieved 3.2% 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. Finally, the runtime impact of RCV is moderate, as it increased the total place-and-route time by only 5% to 10% when satisfying typical timing constraints.

REFERENCES

[1] J. Anderson, S. Nag, K. Chaudhary, S. Kalman, C. Madabhushi and P. Cheng, "Run-Time Conscious Automatic Timing-Driven FPGA Layout Synthesis", Int'l Conf. on Field-Programmable Logic and Applications, 2004, pp. 168-178.

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

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

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

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

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

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

[8] E. Bozorgzadeh, S. Ghiasi, A. Takahashi, and M. Sarrafzadeh, "Optimal Integer Delay Budget Assignment on Directed Acyclic Graphs", IEEE Trans. on CAD, August 2004, pp. 1184-1199.

[9] S. Ghiasi, E. Bozorgzadeh, P. Huang, R. Jafari, M. Sarrafzadeh, "A Unified Theory of Timing Budget Management", IEEE Trans. on CAD, November 2006, pp. 2364-2375.

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

[11] 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.

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

[13] K. So, "Solving Hard Instances of FPGA Routing with a Congestion-Optimal Restrained-Norm Path Search Space", ISPD, 2007, pp. 151-158.

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

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

[16] R. Fung, V. Betz, and W. Chow, "Simultaneous Short-Path and Long-Path Timing Optimization for FPGAs", ICCAD, 2004, pp. 838-845.

[17] Device Family Data Sheets, www.altera.com.