Two-Stage Physical Synthesis for FPGAs

(Invited Paper)

Deshanand P Singh, Valavan Manohararajah and Stephen D Brown
Altera Toronto, 151 Bloor St. West, Suite 200, Toronto, Ontario M5S1S4
Email: {dsingh,vmanohar,sbrown}@altera.com

***Abstract*— This paper presents an overview of an industrial physical synthesis CAD flow for FPGAs. The flow provides a performance speedup of 10%–15% for most circuits, and a significant number of circuits show a speedup of 20%–180%. We describe the algorithms used to achieve this result including: incremental retiming, BDD-based resynthesis, local rewiring, and logic replication. The effectiveness of these operations depends on the ability to accurately determine which portions of logic are timing critical at a stage of the CAD flow where there is still freedom to perform logic restructuring. We show how this problem can be effectively solved by inserting prediction and restrurcturing operations at multiple points of the FPGA CAD flow.**

I. Introduction

Recent research effort in *physical synthesis* has strived to eliminate the artificial separation that exists between the various steps in CAD. Most of the existing work is applicable to ASIC CAD flows [16], [5], [12], [24], [25], [6], but a few of the more recent efforts explore FPGA CAD flows [13], [22], [18]. Some have concentrated on making the synthesis step more aware of what happens during placement and routing [16], [13], while others have explored the use of synthesis type algorithms during placement and routing [5], [12], [24], [25], [6], [22], [18]. Our work builds upon both of these general approaches to develop an effective suite of optimizations that is able to improve the performance of circuits implemented in FPGAs.

Most of the delays in an FPGA circuit are due to the programmable routing network [19]. These delays cannot be determined with great certainty until the routing step completes. The traditional logic synthesis step of the FPGA CAD flow is responsible for creating a circuit implementation that will realize the functionality of a designer's hardware specification. At this early stage of the CAD flow it is difficult to predict the delays of the routed connections. It is for this reason that traditional logic synthesis may create circuit structures that are suboptimal in terms of critical path performance. With perfect knowledge of the routed delays, the task of optimizing the circuit structure to improve performance is well studied. These optimizations include *delay-driven resynthesis* and *sequential retiming*. Thus, the great challenge of FPGA Physical Synthesis is the prediction of delays for use with these timing driven optimizations.

We use a two-stage approach to integrate delay prediction and timing driven netlist optimizations. First, we use **coarse physical synthesis** at an early stage in the FPGA CAD flow that occurs immediately after logic synthesis and technology mapping. A coarse prediction technique is used to approximate delays for the routed connections, and coarse-grained restructuring operations are used to make relatively large scale circuit changes. Secondly, we use **fine physical synthesis** at a late stage in the CAD flow that occurs immediately after placement and just before routing. Of course, it would be advantageous to perform local optimizations once routing has completed and accurate routing delays are available. However, making changes to the circuit during the routing step is extremely complicated. Instead we choose to perform local timing-driven optimizations at the placement step, which is sufficiently close to the routing step that reasonably accurate delays are known. Furthermore, small changes to the circuit can still be made by using a novel incremental placement technique [21].

The remainder of this paper is organized as follows. First, we describe the target FPGA architectures used in this paper. Next, we provide an overview of the coarse and fine physical synthesis techniques, and describe their integration with timing analysis and incremental placement. We then describe four timing-driven circuit optimization techniques. Finally, we present experimental results, and conclude with remarks that are directions for future research.

II. Target FPGA Architectures

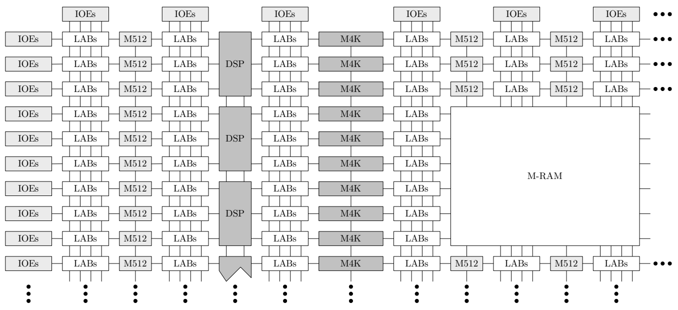
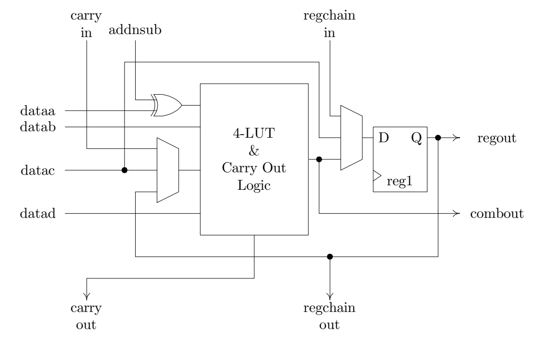
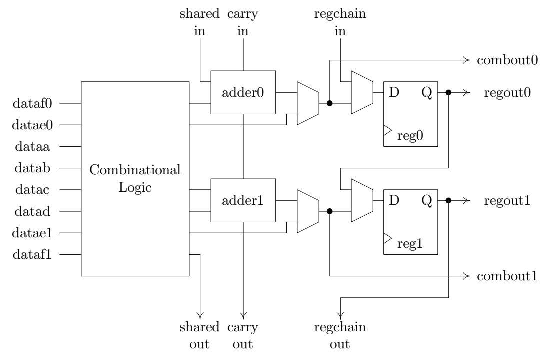


Fig. 1. Structure of the Stratix and Stratix II FPGA architectures.

Altera's Stratix [10] and Stratix II [11] chips were used as the target for the experiments described in this paper. As illustrated in Figure 1, the high level structure of both chips is similar. Both chips are comprised of I/O elements (IOEs), logic array blocks (LABs), digital signal processing blocks (DSPs) and memory elements (M512, M4K and M-RAM). While DSPs and memory elements perform very specific roles in the FPGA, the LABS can be configured to perform arbitrary logic functions. The LABs are also the source of significant differences between the two architectures. A LAB in a Stratix device contains 10 logic elements (LEs) while a LAB in a Stratix II device contains 8 adaptive logic modules (ALMs). The Stratix LE, illustrated in Figure 2(a), contains a four-input lookup table (4-LUT), a register and some logic that facilitates the creation of arithmetic circuits. Figure 2(b) illustrates the Stratix II ALM. It contains two registers, two sets of addition circuitry and a combinational logic module that can implement two functions of varying complexity. The combinational logic module can be configured to implement a single 6-LUT, or two LUTs with five or fewer inputs. If the module is configured to implement two 5-LUTs, the LUTs must share at least two of their inputs as there are only 8 inputs connected to the module.



(a) Stratix.



(b) Stratix II.

Fig. 2. Logic Elements.

III. Two-Stage Constrained Physical Synthesis: An Overview

Figure 3 illustrates the CAD flow used in our work. In the first step, design entry, the design is described in terms of a hardware description language such as VHDL or Verilog. Logic synthesis optimizes the circuit obtained from design entry. During logic synthesis the netlist is represented in terms of a generic gate library. The technology mapping stage converts the netlist to use the logic elements available in the target FPGA architecture. Following technology mapping, a coarse physical synthesis step is performed. Here timing driven optimizations work in concert with timing analysis to identify and restructure the logic on the critical path. Next, a clustering step is used to group the technology mapped circuit into a set of clusters. In both the Stratix and Stratix II architectures, the clustering step creates a set of LABs. Following clustering, placement determines a position for each cluster in the circuit. Once placement is completed, a fine physical synthesis step is performed. Here timing driven optimizations use both timing analysis and incremental placement to restructure the logic on the critical path. Incremental placement is needed to integrate the modifications made by the timing driven optimizations into the existing placement. The final step in the CAD flow, routing, determines the wires that will be used to connect the elements that make up the circuit.

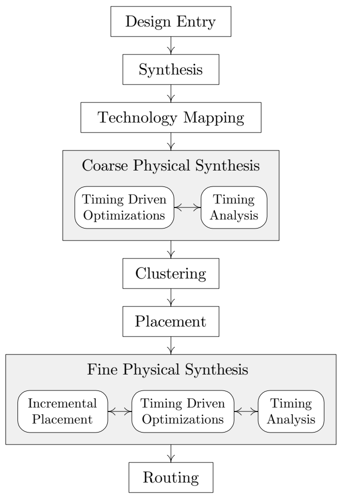


Fig. 3. The two-stage physical synthesis CAD flow.

A. Timing Analysis

Timing Analysis provides the information needed so that timing driven optimizations can identify which paths of a circuit are performance critical. Although the delays within circuit components (logic delays) are known, the delays between circuit components (routing delays) are not known and must be predicted. During coarse physical synthesis, we estimate the delay of a connection between components to be the average delay for a connection of that *type* observed on a large number of place and route experiments. A connection's type is identified by a four-tuple: driving component type, driving port type, driven component type, and driven port type. During fine physical synthesis, placement information is available, and we estimate the delay of a connection between two components to be the delay of the fastest route between the two components. Although the assumption of the fastest route is inaccurate for non-critical signals, it is accurate for critical and near-critical connections which tend to use fast routes when possible.

Once the inter-component delays have been predicted, timing analysis determines the *slack* [4] of every connection. The slack of a connection is defined to be the amount of delay that can be added to the connection before it becomes *critical*. A connection is critical if the length of a path it belongs to exceeds the path-length constraint set by the user. Timing analysis also determines a *slack ratio* for each connection.

The slack ratio is a value between 0 and 1 which indicates the relative importance of each connection to overall circuit timing. Connections that have a significant effect on circuit timing have slack ratios closer to 0 while connections that have negligible effect on circuit timing have slack ratios closer to 1. A precise definition of slack ratios is beyond the scope of this paper. However, from an optimization prespective, slack ratios provide the most accurate information as the formulation accounts for multi-cycle clock constraints, inverted clocks and skew.

B. Incremental Placement

The timing-driven optimizations that take place during fine physical synthesis may create an invalid placement. For example, a BDD-based resynthesis algorithm may create new wires that violate the constraint on the number of wires entering a LAB. A logic replication algorithm may create new LEs or ALMs which would then require placement. Incremental placement (ICP) is used to integrate the modifications made by the timing-driven optimizations into the existing placement while perturbing the existing placement as little as possible. A brief overview of ICP is presented here. The interested reader is referred to [21] for further details.

The primary goal of ICP is to resolve the architectural violations created when the circuit modifications are integrated into the existing placement. Nearly all architectural constraints in modern FPGAs [1] [27] are found in the clustered logic blocks (LABs in our case). Some common constraints include:

- A limit on the number of logic elements within the cluster.
- A limit on the number of distinct inputs to the cluster.
- A limit on the number of distinct control signals (e.g. clock, reset) that can be used within the cluster.

The ICP algorithm uses an iterative improvement strategy where logic elements are moved according to a cost function. This cost function consists of three components:

- **Cluster Legality Cost** - Each cluster is penalized if it contains any architectural violations. The cost is proportional to the total number of constraints violated.
- **Timing Cost** - The timing cost is used to ensure that critical logic elements are not moved into locations that would significantly increase the critical path delay.
- **Wirelength Cost** - Wirelength estimation is used to ensure that the circuit is easily routable after the logic element moves.

The total cost is a weighted sum of these components. Cost-lowering moves are made until no further illegalities exist in the placement. A novel hillclimbing strategy is used to ensure that the iterative improvement algorithm does not get stuck in a local minima where none of the proposed moves seem to improve the cost even though there is remaining illegality.

C. Physical Synthesis Constraints

The timing-driven optimizations that are applied to restructure the circuit are subject to several constraints. These constraints ensure that the resulting circuit is functionally equivalent to the original, can be realized in the target architecture, and meets user requirements. These constraints include:

- *User defined timing constraints*.
- *User defined "dont't touch" constraints*. A user may specify that portions of the logic are not to be touched regardless of potential benefits.
- *Area constraints*. This includes a global constraint on the maximum area increase allowed. It may also include constraints to ensure that registers are created evenly across the design during retiming.
- *Architectural constraints*. These constraints define rules for handling specialized structures in the target FPGA architecture. For example, carry chains provide high speed implementations of arithmetic logic, but signals propagating along the chain must be strictly combinational and must have a fanout of one. Thus, combinational resynthesis algorithms must avoid modifying carry chains unless they are capable of producing the highly specialized circuit topology required by the chain. Furthermore, a retiming algorithm must ensure that a register does not end up between the elements of a chain.
- *Implicit constraints*. These constraints are automatically generated to ensure that the circuit functions correctly after the application of the timing-driven optimizations. An example of this constraint is shown in Figure 4. In this example a register feeds the asynchronous reset signal of several other registers in the design. Retiming theory allows us to move the source register backwards. However, doing so may introduce a glitch on the signal that feeds the asynchronous lines. This situation could potentially cause disasterous malfunctions. These types of implict legality contraints are described in detail in [26].

With the exception of user defined timing constraints, the other constraints are handled as the optimizations take place. If a restructuring operation violates any of the constraints, it is undone and the optimization process continues on another part of the circuit. User defined timing constraints are handled by the timing analysis engine. The slack ratios produced by timing analysis are computed with respect to the user defined timing constraints. Thus, as long as the optimization algorithms use slack ratios to identify the parts of the circuit to be restructured, user timing constraints will be adhered to.

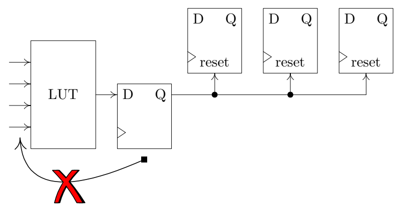


Fig. 4. An example of an asynchronous constraint.

```
1   $bestt \leftarrow$ TimingAnalysis()
2   $bestc \leftarrow Circuit$
3   **for** $i \leftarrow 1$ **upto** $MaxIterations$
4       $S \leftarrow$ CriticalLogic()
5       **for** $s \in S$
6           $Circuit \leftarrow$ Optimize($s$)
7       **end for**
8       IncrementalPlacement() (if necessary)
9       $t \leftarrow$ TimingAnalysis()
10      **if** $t \geq bestt$
11          $bestt \leftarrow t$
12          $bestc \leftarrow Circuit$
13      **end if**
14  **end for**
15  $Circuit \leftarrow bestc$
```

Fig. 5. A framework for timing-driven optimization circuit optimization.

IV. Timing-Driven Circuit Optimizations

We now describe four timing-driven optimization algorithms. The first two, incremental retiming and BDD-based resynthesis, are used during both coarse and fine physical synthesis, and the last two, local rewiring and logic replication, are used during fine physical synthesis only. There are slight differences between the versions of the algorithms used during the two physical synthesis steps. The versions used during coarse physical synthesis target a larger portion of the circuit and have a greater freedom in restructuring the circuit than the versions used during fine physical synthesis.

All four timing-driven optimizations use the algorithmic layout illustrated in Figure 5. A call to timing analysis is the first major task performed by each algorithm. In addition to helping determine the slacks and slack ratios for each connection in the circuit, timing analysis returns the minimum slack in the circuit. We use the minimum slack to judge the quality of a circuit during the optimizations. Each iteration begins by identifying the critical and near critical logic in the circuit (line 4). We then perform optimizations on the selected logic (line 6). This is the step where each of the four optimization techniques perform different operations. If any of the physical synthesis constraints are violated during an optimization, the optimization is undone and an unmodified circuit is returned. Following the optimizations, incremental placement is used to integrate the changes made to the circuit into the existing placement. This call is used only during the fine physical synthesis step. At the end of each iteration (lines 10–13), the modified circuit is saved if timing has been improved. Once all iterations have completed, the best circuit discovered during the iterations replaces the initial circuit.

A. Incremental Retiming

Sequential retiming is a powerful logic optimization technique for synchronous circuits which uses the property that flip flops can be taken from the outputs of gates and moved to their inputs, or vice versa. Using these moves in combination, one can attempt to maximize circuit speed and minimize area. This technique was first introduced in the early 1980's by Leiserson and Saxe [8] [9] where an optimal solution to the retiming problem is presented. The solution involves solving a system of difference equations. For an FPGA circuit netlist containing n elements, with at most k inputs, the worst-case complexity of this optimal retiming algorithm is $O(n^2 log(n))$.

In contrast, we have implemented an incremental retiming algorithm that is linear in complexity and produces results that are very close to optimal [20]. The basic idea behind our retiming scheme is to perform a series of backward and forward retiming iterations. During a backward retiming iteration, we identify registers whose inputs come from a critical or a near critical path. These registers are then pushed backwards across the logic driving it as illustrated in Figure 6a. During a foward retiming iteration, we identify registers whose outputs are connected to a critical or a near critical path. These registers are then pushed forwards across the logic being driven as illustrated in Figure 6b. During both backward and forward pushes we have to ensure that the functionality of the circuit is unchanged during powerup and reset conditions. Registers in Stratix and Stratix II are set to zero on powerup. Reset signals also set the register to zero. In Figure 6, following the backward and forward pushes, the functionality of LUTs f, g, and h is changed so as to preserve the powerup and reset functionality expected of the subcircuits illustrated.

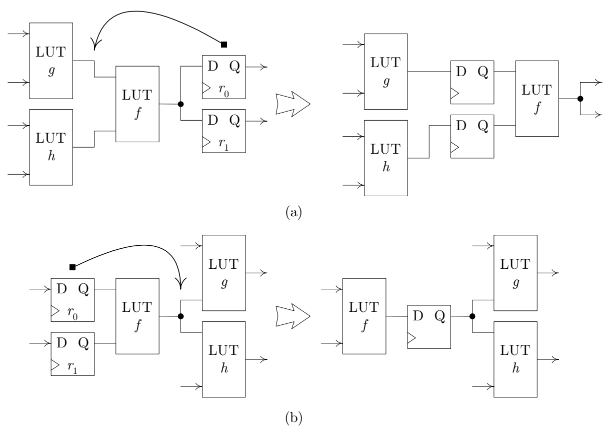


Fig. 6. An example of backward and forward retiming pushes.

The worst case complexity of our retiming algorithm is $O(Kn)$, where K is the number of retiming iterations and n is the number of nodes in the circuit. Given that K is a constant, the algorithm has linear time complexity $O(n)$.

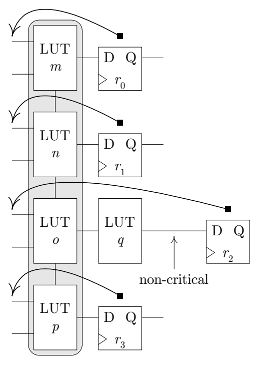


Fig. 7. Backward Push across a Carry Chain.

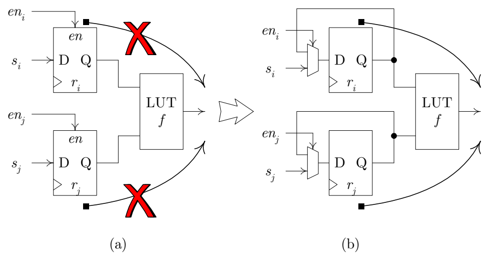


Fig. 8. Incompatible Secondary Signals.

One of the complexities of implementing this incremental retiming algorithm for FPGAs lies in handling the complexities of carry chains. These chains provide high-speed implementations of arithmetic logic and our internal experiments indicate that they are often involved in performance-critical regions of logic. The most important constraint that we must observe is that the carry chain cannot provide registered versions of the carry signal propagating between elements. These carry connections must be strictly combinational. This restriction prevents our algorithm from pushing registers across individual elements of any particular chain. Instead, we perform "group pushes" across entire chains. This ensures that the chain remains in a legal configuration after a register push. Figure 7 illustrates this situation. Suppose that the register r_0 has a critical input. We can then perform a backwards group push across the entire chain where registers $r_0 \ldots r_3$ are moved backwards to the inputs of the chain. Notice that register r_2 is not fed directly by the chain and is fed by a non-critical connection. Normally, this register would never be moved backward because it isn't directly involved with a timing-critical path; however, we actively search for these types of situations to enable a legal group push.

Another complexity introduced by our FPGA architectures is the diverse functionality provided by the configurable registers. These registers offer features such as clock enables, synchronous clears/loads and asychronous clears/loads. Registers can only be pushed across a logic element if they contain exactly the same set of these secondary control signals [3]. There are many situations where critical pushes may involve registers that have incompatible control signals. Consider the case shown in Figure 8(a). Suppose that the register r_i has a critical output, but the clock enable signals en_i and en_j are distinct. Retiming theory would disallow a seemingly beneficial forward push; however, we attack this problem by decomposing the registers r_i and r_j into simpler forms that contain explicit enable logic as shown in Figure 8(b). This decomposition now allows us to push the registers forward and improve the critical path.

B. BDD-Based Resynthesis

The BDD-based resynthesis algorithm finds alternative functional decompositions for the critical or near critical logic in the circuit. Given a function $f(X, Y)$ defined over two sets of variables X and Y, functional decomposition finds subfunctions

$$g_1(Y), g_2(Y), \ldots, g_k(Y)$$

such that f can be re-expressed in terms of the gs:

$$f(X, g_1(Y), g_2(Y), \ldots, g_k(Y))$$

The set of variables X is referred to as the *free set* and the set of variables Y is referred to as the *bound set*. If there are no variables common to X and Y, the decomposition is said to be *disjoint*. Otherwise the decomposition is *non-disjoint*. We consider both disjoint and non-disjoint decompositions during resynthesis.

The LUTs in an FPGA are capable of implementing any function of k variables. Thus, functional decomposition (as opposed to algebraic decomposition) can be used to find subfunctions that fit naturally into LUTs. The structure of the decomposition may have a significant impact on timing. Our decomposition technique attempts to structure the circuit in such a way as to minimize the number of logic levels traversed by critical signals.

Figure 9 illustrates the process of resynthesis. First, a LUT f with critical inputs is identified. Next, a cone of logic rooted at f is grown. The cone is then collapsed into a single LUT and a BDD [2] representing the functionality of the cone is constructed. Functional decomposition is performed directly on the BDD [7]. At each step of the decomposition, a single LUT suitable for the target architecture is extracted from the BDD and the BDD is reexpressed in terms of the extracted LUT. This procedure is continued until the remaining BDD fits into a single LUT.

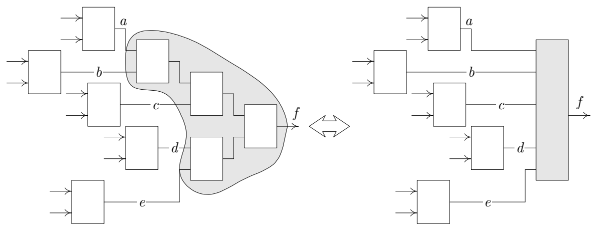


Fig. 9. The BDD-based resynthesis operation.

An example of BDD-based functional decomposition is given in Figure 10. The figure illustrates a reduced, ordered BDD for the function

$$f = (p + q + r)\bar{s} + \overline{(p + q + r)}(\bar{s}t + \bar{s}u + s\bar{t}\bar{u})$$

Here, we have chosen an alphabetical ordering of the variables for the BDD of f. However, during resynthesis, we use a sifting [17] procedure that helps move non-critical variables to the top of the BDD. A *cut* in the BDD establishes two sets of variables. The variables above the cut constitute the bound set and the variables below the cut constitute the free set. Figure 10 illustrates a cut in f that separates the bound set, $\{p, q, r\}$, from the free set, $\{s, t, u\}$. The portion of the BDD above the cut references two distinct functions, f_0 and f_1, below the cut. Thus, the portion of the BDD above the cut can be replaced by a single boolean variable g that determines whether f_0 or f_1 is to be selected. A separate BDD computes the value for g, and in the new BDD for f, f_0 is selected when $g = 0$ and f_1 is selected when $g = 1$. Note that this encoding is abitrary. We could have just as easily selected f_0 when $g = 1$ and f_1 when $g = 0$. The resulting decomposition can be expressed as

$$\begin{aligned} g &= p + q + r \\ f &= g\bar{s} + \bar{g}(\bar{s}t + \bar{s}u + s\bar{t}\bar{u}) \end{aligned}$$

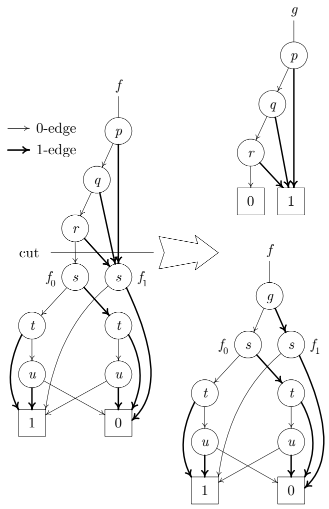


Fig. 10. An example of BDD-based decomposition.

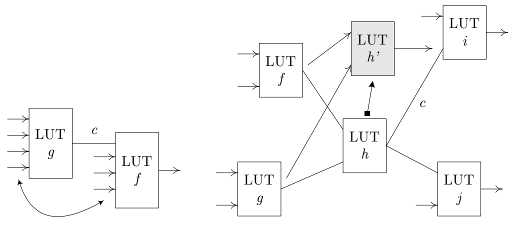
We refer the interested reader to [15] for a description of heuristics used for cone expansion and subfunction extraction.

C. Local Rewiring

Figure 11(a) illustrates the local rewiring optimization [14]. We identify a pair of LUTs f and g connected by a critical signal c. Using functional decomposition techniques, we determine if the overall timing of the two LUTs can be improved by swapping some of the non-critical signals attached to f with some of the critical signals attached to g. Although local rewiring and BDD-based resynthesis use functional decomposition techniques, they operate on two different scales. Local wiring considers two LUTs at a time while BDD-based resynthesis considers entire cones. The result is that the operations carried out during local retiming have much more predictable timing changes and introduce very little illegality into the existing circuit.

D. Logic Replication

Following placement, a LUT that drives a signal with several fanouts may be placed at a location that is not ideally suited for any of its fanouts. For example, in Figure 11(b), LUT h drives two LUTs i and j, and it has been placed at a location that balances its need to drive both LUTs at the same time. However, if connection c is critical, we can replicate h to produce a new LUT h' which can be placed closer



(a) Local Rewiring. (b) Logic Replication

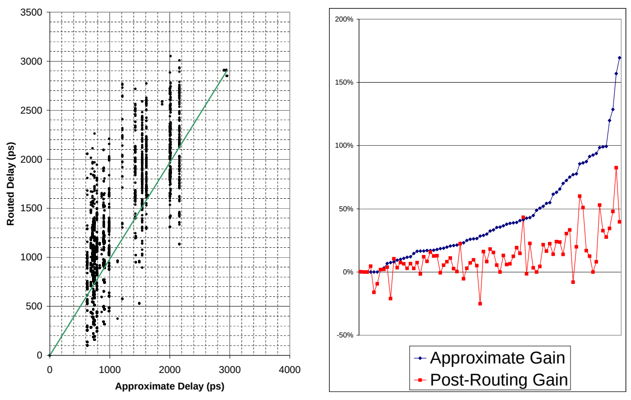
Fig. 11. Rewiring and Replication

to the target of c. The logic replication algorithm performs this transformation on critical signals driven by multi-fanout sources.

V. RESULTS

In this section, we provide experimental results illustrating the predictive power of both coarse and fine physical synthesis. In addition, we demonstrate the effeciveness of our two-stage approach for performance optimization of a large industrial benchmark suite.

A. Coarse Prediction



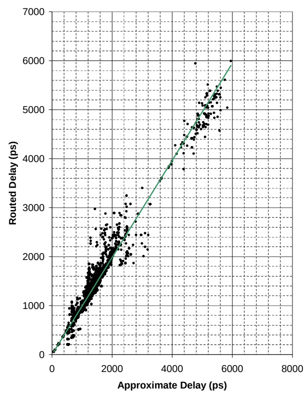
(a) Delay Prediction. (b) Correlation Results.

Fig. 12. Coarse Physical Synthesis.

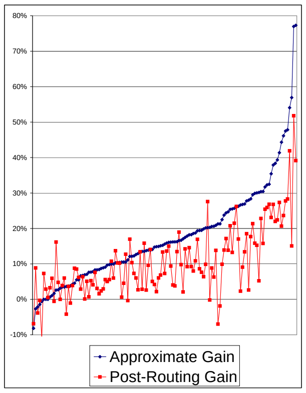
As discussed previously, the coarse prediction techniques assign a delay value to a connection based on characteristics of the source and sink. These delay values are determined by computing the average delay for each connection type over a large number of benchmark circuits and a number of place and route runs. Figure 12(a) demonstrates the effect of this approximation technique on a single benchmark circuit. We plot the final routed delay vs. the approximate delay for each connection in the netlist. This graph shows a number of vertical “delay bands.” The banding results from the fact that we assign a single delay value to each connection type; however, each of these connection types are likely to have different placements as well as routings.

The test of any approximation technique lies in its predictive power. Figure 12(b) illustrates the predictive power of the coarse physical synthesis approach. The graph shows the approximate performance gains predicted by using our timing-driven netlist optimizations in conjunction with the coarse prediction technique for approximately 100 industrial circuits. A second curve shows the actual gain that results post-routing. The general trend shows that a prediction of a large performance gain usually results in a significant performance gain after place and route. The correlation coefficient (R) is a statistical measure that expresses how closely two variables are linearly related. Predictive strength is computed by measuring the correlation coefficient between the approximate gains and post-routing gains. We find that $R_{coarse} = 0.5$. A value of 0 indicates no predictive power, while a value of 1 corresponds to perfect prediction.

B. Fine Prediction



(a) Delay Prediction.



(b) Correlation Results.

Fig. 13. Fine Physical Synthesis.

Once placement is completed, the timing-driven optimization techniques discussed previously are used to improve the circuit's critical path. Since placement is completed, the delay of every connection can be estimated by computing an approximate route for the connection. The approximation that we use is to assume that the fastest possible route is available from the source to the sink. This calculation is extremely fast since it can be efficiently cached and it is reasonably accurate. Critical connections tend to be assigned to the fastest possible route since the Quartus II router will attempt to optimize timing as much as possible. Our approximation is not accurate for non-critical signals since the router may use non-optimal paths to avoid congestion; however, our timing-driven optimizations generally target regions of the circuit involving the most timing critical logic. As before, Figure 13(a) depicts this effect for a single benchmark circuit. Clearly, our fine physical synthesis approximation is very close for the majority of connections.

Figure 13(b) demonstrates the predictive power of the fine physical synthesis techniques. We find that $R_{fine} = 0.8$. Clearly, our approximations are always optimistic. However, the general trend shows that we have greater fidelity between our prediction and the post-routing result than the coarse physical synthesis approach.

C. Results on Industrial Circuits

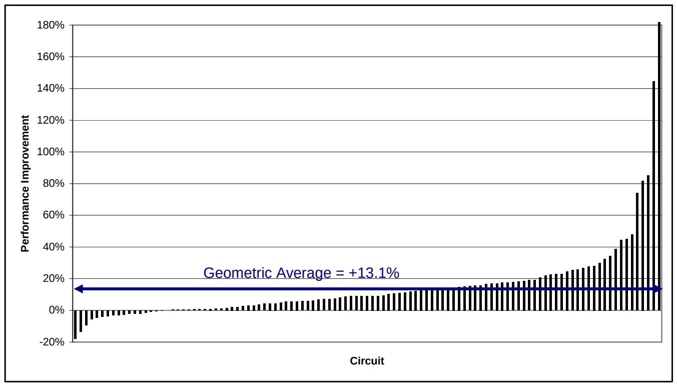


Fig. 14. Physical Synthesis Results.

Although, we have greater predictive power using our fine physical synthesis approach there is a limitation in the amount of restructuring that is possible since we must incrementally sift modifications into the existing placement. Therefore, our approach involves making large scale and aggressive circuit optimizations during coarse physical synthesis. During late physical synthesis, we attempt to use the greater predictive power to "fix" any paths that were missed during the coarse optimizations. Figure 14 demonstrates the performance gains that were obtained using our two-stage flow on a Stratix II industrial benchmark suite (design sizes range from 5000–110000+ LEs). We find that the average performance gain is approximately 13%. For this performance gain, the area penalty is a relatively small 3.6% increase in logic elements. This result is consistent across all Altera FPGA families and is similar regardless of the logic synthesis tool (Quartus Integrated Synthsis or leading third party solutions).

VI. Conclusions

We have presented a two-stage physical synthesis approach for FPGAs. These techniques typically provide a 10%–15% performance improvement for an "average" circuit and a significant fraction of the circuits have a gain between 20% and 180%.

Our future research will focus on two areas. The first is improved prediction in coarse physical synthesis by examining netlist structure or performing coarse and quick placement. All of these techniques will attempt to assign predicted delays to connections; however, we feel that path-based prediction may offer the greatest opportunities. Path-based schemes would attempt to predict the criticality of entire paths early in the CAD flow rather than attempting to accurately compute the delay of each connection.

The second area of research involves additional physical synthesis stages. For example, there are certain constrained optimizations [23] that can occur after routing and take advantage of the greatest possible timing predictability.

References

[1] Altera. *Altera Databook*.

[2] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions to Computers*, Vol. C-35, No. 8, Aug. 1986, pp. 677–691.

[3] K. Eckl, J.C. Madre, P. Zepter and C. Legl. A Practical Approach to Multiple-Class Retiming. *DAC*, 1999.

[4] R. Hitchcock, G. Smith and D. Cheng. Timing Analysis of Computer-Hardware. *IBM Journal of Research and Development*, Jan. 1983, pp. 100–105.

[5] Y. Jiang, A. Krstic, K. Cheng and M. Marek-Sadowska. Post-Layout Logic Restructuring for Performance Optimization. In *Proceedings of the Design Automation Conference*, Anaheim, CA, June, 1997, pp. 662–665.

[6] L. Kannan, P. Suaris and H. Fang. A Methodology and Algorithms for Post-Placement Delay Optimization. In *Proceedings of the Design Automation Conference*, San Diego, CA, June 1994, pp. 327–332.

[7] Y-T. Lai, K-R. Pan and M. Pedram. OBDD-Based Functional Decomposition: Algorithms and Implementation. *IEEE Trans. On Computer Aided Design*, Vol. 15, No. 8, 1996, pp. 977–990.

[8] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry. *Journal of VLSI and Computer Systems*, pages 41–67, 1983.

[9] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[10] D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, B. Pedersen, G. Powell, S. Reddy, C. Wysocki, R. Cliff, and J. Rose. The Stratix Routing and Logic Architecture In *FPGA '03, ACM. Symp. FPGAs*, pages 15–20, 2003.

[11] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M.Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff and J. Rose. The Stratix II Logic and Routing Architecture. In *FPGA '05, ACM Symposium on FPGAs*, pages 14–20, 2005.

[12] Y. Lian and Y. Lin. Layout-based Logic Decomposition for Timing Optimization. In *Proceedings of the Asia Pacific Design Automation Conference*, Hong Kong, Hong Kong, Jan. 1999.

[13] J. Y. Lin, A. Jagannathan and J. Cong. Placement-Driven Technology Mapping for LUT-Based FPGAs. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2003, pp. 121–126.

[14] V. Manohararajah, D. P. Singh, S. D. Brown and Z. G. Vranesic. Post-Placement Functional Decomposition for FPGAs. In *Proceedings of the International Workshop on Logic Synthesis*, Temecula, CA, June 2004, pp. 114–118.

[15] V. Manohararajah, D. Singh and S. Brown. Timing Driven Functional Decomposition for FPGAs. To appear in *IWLS'2005*.

[16] M. Pedram and N. Bhat. Layout Driven Logic Restructuring/Decomposition. In *Proceedings of the Int. Conf. on Computer-Aided Design*, San Jose, CA, Nov. 1991, pp. 134–137.

[17] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of the Int. Conf. on Computer Aided Design*, Santa Clara, CA, 1993, pp. 42–47.

[18] K. Schabas and S. D. Brown. Using Logic Duplication to Improve Performance in FPGAs. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2003, pp. 136–142.

[19] M. Sheng and J. Rose. Mixing Buffers and Pass Transistors in FPGA Routing Architectures. In *Proceedings of the ACM Int. Symposium on FPGAs*, Monterey, CA, Feb. 2001, pp. 75–84.

[20] D. Singh, V. Manohararajah, S. Brown. Incremental Retiming for FPGA Physical Synthesis. To appear in *DAC'2005*.

[21] D. P. Singh and S. D. Brown. Incremental Placement for Layout-Driven Optimizations on FPGAs. In *Proceedings of the Int. Conf. on Computer-Aided Design*, San Jose, CA, 2002, pp. 752–759.

[22] D. Singh and S. Brown. Integrated Retiming and Placement for Field Programmable Gate Arrays. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2002, pp. 67–76.

[23] D. Singh and S. Brown. Constrained Clock Shifting for Field Programmable Gate Arrays. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2002, pp. 121–126.

[24] G. Stenz, B. Riess, B. Rohfleisch and F. Johannes. Timing Driven Placement in Interaction with Netlist Transformations. In *International Symposium on Physical Design*, Napa Valley, CA, 1997, pp. 36–41.

[25] T. Tien, H. Su and Y. Tsay. Integrating Logic Retiming and Register Placement. In *Proceedings of the Int. Conf. on Computer-Aided Design*, San Jose, CA, 1998, pp. 136–139.

[26] B. van Antwerpen, M. Hutton, G. Baeckler and R. Yuan. A Safe and Complete Gate-Level Register Retiming Algorithm. In *IWLS 2003*, pages 140–147, 2003.

[27] Xilinx. *Xilinx Databook*.