# An Approach to Placement-Coupled Logic Replication

Miloš Hrkić
University of Illinois at Chicago
Dept. of Computer Science
Chicago, IL 60607
mhrkic@cs.uic.edu

John Lillis
University of Illinois at Chicago
Dept. of Computer Science
Chicago, IL 60607
jlillis@cs.uic.edu

Giancarlo Beraudo
University of Illinois at Chicago
ECE Department
Chicago, IL 60607
gberaudo@ece.uic.edu

## ABSTRACT

We present a set of techniques for placement-coupled, timing-driven logic replication. Two components are at the core of the approach. First is an algorithm for optimal timing-driven fanin tree embedding; the algorithm is very general in that it can easily incorporate complex objective functions (e.g., placement costs) and can perform embedding on any graph-based target. Second we introduce the *Replication Tree* which allows us to induce large fanin trees from a given circuit which can then be optimized by the embedder. We have built an optimization engine around these two ideas and report promising results for the FPGA domain including clock period reductions of up to 36% compared with a timing-driven placement from VPR [12] and almost double the average improvement of local replication from [1]. These results are achieved with modest area and runtime overhead.

## Categories and Subject Descriptors

B.7.2 [**Hardware**]: Integrated Circuits—*Design Aids*

## General Terms

Algorithms, Performance

## Keywords

Timing Optimization, Logic Replication, Placement, Programmable Logic

## 1. INTRODUCTION AND BACKGROUND

The idea of *logic replication* is to duplicate certain cells in a design so as to enable more effective optimization of one or more design objectives. The idea has been applied in several different contexts including min-cut partitioning (e.g., [10] [11]) and fanout tree optimization (e.g., [9] [13]).

Recently a few papers including [1], [2] and [3] have explored the idea of using replication to effectively deal with interconnect-dominated delay at the physical level. In these papers it is observed that, because replication effectively separates multiple signal paths it becomes easier to, at the physical design level, "straighten" input-to-output (flip-flop to flip-flop) paths which might otherwise have been very circuitous (and therefore high delay).

This work was supported in part by NSF CAREER Award CCR-9875945.

*DAC'04,* June 7–11, 2004, San Diego, California, USA.


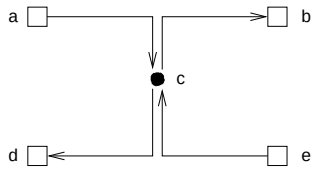


**Figure 1: Example with forced non-monotone paths.**

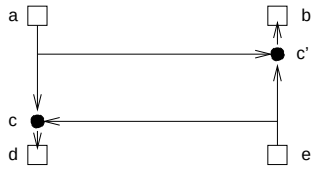


**Figure 2: Example illustrating path straightening by replication of cell $c$.**

A simple example from [1] reproduced in Figures 1 and 2 illustrates the idea. Suppose that the terminals at $a$, $b$, $d$ and $e$ are fixed. There are 4 distinct input-to-output paths; any movement of the central cell $c$ from the shown location will degrade the delay of at least one of these paths (assume for the moment a linear delay model). Thus in Figure 1 we have no choice but to tolerate non-monotone input-to-output paths. Now suppose that we replicate cell $c$ as shown in Figure 2 to form $c'$ computing the same function, but feeding only output $b$ while $c$ drives only $d$. If we produce such a logically equivalent netlist all input-to-output paths become virtually monotone.

The work of [1] exploits this phenomenon and is the most closely related to this paper, so we briefly review its contributions. First, [1] made a compelling case for the potential of replication by observing that not only do typical placements contain critical paths which are highly non-monotone, but also that the number of cells which have near-critical paths flowing through them is relatively small (thus, one may conjecture that a small amount of replication may be sufficient). Then an incremental replication procedure was proposed and evaluated experimentally with promising results. Roughly speaking the algorithm examined the current critical path and looked for cells to replicate; for such cells, it placed the duplicate, performed fanout partitioning and then legalized the placement. The criteria for selecting a cell was based on the goal of inducing *local monotonicity.* Local monotonicity was defined by a sequence of 3 cells on

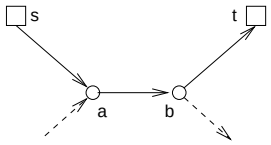


**Figure 3: Limitation of local monotonicity. Cells $a$ and $b$ are locally monotone yet $s$-to-$t$ path is not.**

a path $v_1, v_2, v_3$. Let $d(u, v)$ be the rectilinear distance between cells $u$ and $v$; then we say that the path from $v_1$ to $v_3$ is non-monotone if $d(v_1, v_3) < d(v_1, v_2) + d(v_2, d_3)$ (i.e., traveling to $v_2$ creates a detour). In such a case, $v_2$ is a good candidate for replication so as to straighten this path without disturbing other paths passing through $v_2$.

While this strategy proved effective in reducing clock period, we now observe that a technique based on local monotonicity has limitations. Figure 3 demonstrates this limitation. In the figure we see a critical path $(s, a, b, t)$ (dashed lines indicate other signal paths which may be near critical). Clearly, this path is non-monotone and yet, all sub-paths (of length 3) are locally monotone. In this case (which is not unusual), the approach is unable to improve the delay.

With this in mind, we have developed a more robust and general replication strategy. There are two key elements around which the approach has been built.

First we study the optimal *fanin tree*[1] embedding problem. In this problem we are given the root of a fanin tree (e.g., a flip-flop), a tree circuit which produces its inputs and arrival times at the inputs (leaves) of the tree. Our goal then is to embed the tree so as to obtain a tradeoff between the cost of the embedding (which can be quite general as we will see) and the arrival time at the root (sink) of the tree. Our solution has evolved from the closely related problem of embedding a fanout tree in buffer tree synthesis [5].

While this is an interesting result in its own right, unfortunately, most circuits, because of reconvergence, do not contain large sub-circuits which are fanin trees. This brings us to the second item at the core of the approach – the *Replication Tree*. The replication tree gives us a systematic way of taking a set of edges in a circuit forming a directed tree (e.g., with the root being the input of a flip-flop), and, using replication, induce a genuine fanin tree which can, in turn, be optimized by the fanin tree embedder. For timing optimization as in our case, a natural selection for such a tree is a *slowest paths tree* derived from static timing analysis. At this point, the embedder's ability to handle general cost functions becomes important – in particular, we are able to naturally encode the cost/benefit of replicating a cell in the "placement cost" component of the cost function.

Around these two main ideas – fanin tree embedding and the replication tree – we have built an optimization engine for the FPGA domain. Additional components of interest include a timing-driven legalizer and a set of post-processing enhancement techniques.

The paper is organized as follows. Section 2 describes a solution to the timing-driven fanin tree embedding problem. We introduce the *Replication tree* in section 3. Timing-driven legalizer is described in section 4. Section 5 gives a top-level view of our approach. Section 6 reveals some of the implementation details. Experimental results are presented in section 7 and we conclude in section 8.

[1]Fanin trees are referred to by some authors as Fan-out-Free Circuits or Leaf-DAG Circuits [4]; we can handle either such structure.

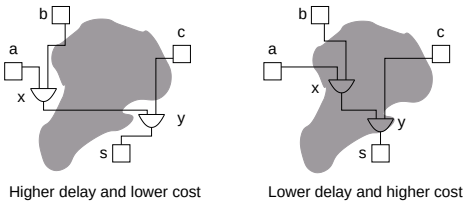


**Figure 4: Example of fanin tree embedding.**

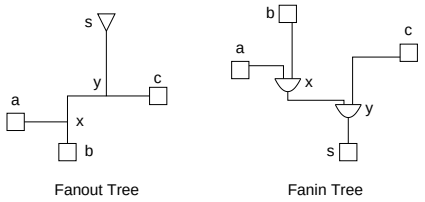


**Figure 5: Similarity between fanout and fanin tree.**

## 2. FANIN TREE EMBEDDING

In the fanin tree embedding problem we are given a fanin tree, placement of leaves (inputs) and root (sink), arrival times at the inputs and a target placement region (in our case this is encoded in an embedding graph). The goal is to place the internal tree nodes (gates) minimizing cost subject to an arrival time constraint at the root (typically there is a tradeoff between cost and arrival time).

In the general case, the cost function is extremely flexible and may include, in addition to wire-length cost, "placement cost" in which a cost $p_{ij}$ is incurred when cell $i$ is placed at slot $j$. This is extremely useful in our application since it allows us to give a cost "discount" if a cell is placed "on-top" of a logically equivalent cell (and thus these two cells can be unified). Thus, the solutions to the embedding problem naturally capture replication overhead.[2]

Figure 4 illustrates two embeddings of the same fanin tree. Given that the shaded region in the middle has high placement cost, we can have a solution with smaller cost but larger delay (left part of the figure), or we can have a solution with better delay but larger cost (on the right).

We have observed that the problems of embedding fanin and fanout (see [5, 6]) trees are very similar. A simple example is given in Figure 5. On the left we have a fanout tree with source $s$ and sinks $a$, $b$ and $c$ (signal flow is from top to bottom). In fanout tree embedding we place Steiner nodes $x$ and $y$. In fanin tree case, on the right, we have sink $s$, inputs $a$, $b$ and $c$, and in fanin tree embedding we place gates $x$ and $y$. For this reason, we have been able to adapt the Dynamic Programming (DP) embedding algorithm of the S-Tree algorithm [5] to the fanin tree problem.

The DP approach for fanout tree embedding starts from sinks and propagates *required-arrival time* and cost toward the source. In the case of a fanin tree we start from inputs and propagate *arrival time*, and cost toward the sink.

[2] As an aside, a simple linear program as in [8] can solve special cases of the embedding problem but seem incapable of solving it in the generality we have described here.

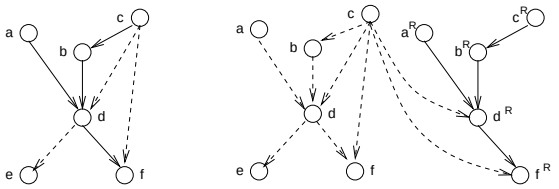


**Figure 6: Replication Tree example.**

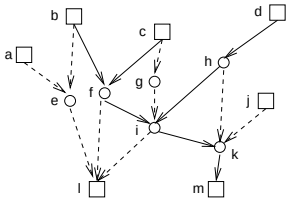


**Figure 7: Example of $\epsilon$-slowest paths tree.**

In the resulting DP approach for fanin tree embedding, a candidate solution (embedding) for a subtree rooted at node $i$ in the tree with node $i$ placed at vertex $j$ in the embedding graph is represented by its *signature* $(c, t)$, indicating that this subsolution incurs cost $c$ and has latest arrival time $t$ at $i$. Solutions at leaves are initialized to have zero cost and arrival times as specified by the problem instance (which is zero for PIs and FFs and latest arrival time computed by static timing analysis for other leaves).

In the bottom-up DP procedure we must combine candidate solutions from subtrees to form new candidate solutions. At internal node $i$ in the tree and vertex $j$ in the graph we *join* sub-tree solutions as follows:

$$c = p_{i,j} + c_1 + c_2 + ... + c_k$$
$$t = max(t_1, t_2, ..., t_k)$$

where $k$ in the number of inputs for gate at $i$, and $p_{i,j}$ is placement cost. For each pair $(i, j)$ instead of single best solution we keep a list of *non-dominated* solutions. One solution dominates the other if it is superior in both dimensions (i.e., both cheaper and faster). After computing joined solutions, they are propagated through the embedding graph using generalized version of Dijkstra's shortest path algorithm, as in [5]. At the root we obtain a set of solutions with cost vs. delay trade-off. From the trade-off curve we pick a fastest solution that is not faster than the precomputed lower-bound on best possible worst delay of the circuit (which is in general limited by distance between PIs and POs and number of logic blocks in between).

## 3. THE REPLICATION TREE

Since most circuits do not have large fanin trees due to reconvergence, we have devised the *Replication Tree* which enables us to induce large fanin trees in a logically equivalent circuit. The idea is best illustrated by an example. In the left part of Figure 6 we have a portion of a circuit with a set of edges in bold. These edges form a tree with all edges pointing toward the root $(f)$. Note that in the left figure, this tree does not form a valid fanin tree due to reconvergence. To induce a fanin tree we (temporarily) make a copy of each node in the tree $(f, d, a, b, c)$. If the original cell is $v$ and the copy is $v^R$, we assign connections as follows: let $u_1, ..., u_k$ be the inputs to $v$. If $(u_i, v)$ is a tree edge, then $v^R$ receives its $i$'th input from $u_i^R$; otherwise, it receives its $i$'th input from $u_i$.

This construction is applied to the circuit in the figure and results in the circuit on the right and yields a fanin tree sub-circuit formed by the replicated cells. Notice that cells $d^R$ and $f^R$ connect to $c$ rather than $c^R$ – otherwise the replicated cells would not form a proper fanin tree (technically speaking it is a Leaf-DAG because, for example "leaf" node $c$ connects to two cells in the tree, however, since the timing properties of $c$ are fixed and known, this does not complicate the embedding process). Thus from the construction we have two claims. First, if we modify the circuit in this way (again, temporarily), the result is functionally equivalent; this is clear from the construction. Second, the set of replicated nodes form the internal vertices of a legitimate fanin tree which can be embedded.

The temporary nature of the replication can now be tied to the placement cost which we have incorporated into the embedding formulation and this point is crucial. We mentioned that placing a node coincidentally with a logically equivalent node receives a "discount." In the context of the replication, this should now become clear – if the embedder places $v^R$ at the same location as $v$, there is no replication and thus, we implicitly only replicate the cells that yield the most significant improvement. A special case may occur if node $v$ has fanout of one. Then we still replicate but all placement locations receive a discounted cost, since no actual replication will ever occur.

Furthermore, over the course of multiple optimizations, we may have more than two copies of a cell. Placement cost is assigned accordingly in such situations (i.e., placement with *any* logically equivalent cell receives a discounted cost, not only with the immediate source of the replication).

Clearly there are many trees in a timing graph which we may use to generate a replication tree. For timing optimization, it is natural to focus on trees with slow paths. The *slowest paths tree (SPT)* can be thought of as the result of finding a longest paths tree from the critical sink in the timing graph *with the edges reversed* (equivalently, finding the shortest paths tree in the reversed graph with the delay values negated). Finding this tree is trivial once the static timing analysis has completed.

Similarly, an *$\epsilon$-SPT* is a subset of the slowest paths tree which includes only cells with paths within $\epsilon$ of the current critical path delay. This allows us to focus on the most critical portions of the fanin cone of the critical sink. An example of *$\epsilon$-slowest paths tree* is given in Figure 7. Circuit inputs are $a$, $b$, $c$, $d$ and $j$. Outputs are $l$ and $m$. Sink $m$ has been identified as critical. Edges of the $\epsilon$-SPT are shown with solid lines and dashed edges represent circuit connectivity. Note that $g$ and $j$ are not contained in the $\epsilon$-SPT.

## 4. TIMING-DRIVEN LEGALIZATION

After the preceding phases, it is possible that some cells overlap in the placement (actually it is very likely). The purpose of the legalizer is to resolve those overlaps and move cells from congested to empty locations. We observe that by moving cells that are on the critical path one may degrade circuit performance. In order to minimize perturbations to the placement and preserve timing achieved in the embedding phase (as much as possible), we have adopted *ripple-*

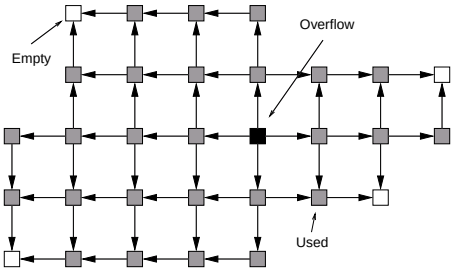


**Figure 8: Gain graph in legalizer.**

*move* strategy from [7]. We have also modified this strategy to incorporate timing as well as wiring information.

The legalizer is invoked after each embedding phase. During embedding it is possible that we replicate and/or move multiple cells, so we may have more than one violation in the placement. If an overlap-free placement is achievable (i.e. there are enough free slots), the legalizer will resolve one overlap at a time until the entire placement is legal.

In the procedure we first identify an overlap location. If we have more than one overlap, we pick the first one we encounter while we scan placement for overlaps. We then identify up to four closest free slots (one slot in each quadrant, if they exist, assuming that the center is at the congested slot). Next we identify which of those free slots will be used for legalization. To do this, we construct a gain graph (Figure 8), which has monotone paths from congested slot to free slots. Each edge is labeled by the gain value that we get by moving a cell from that slot to the neighboring slot (in the direction toward the target free slot).

Gain is computed as the difference of costs of having a cell at the current and the neighboring slot. This cost has wire and timing component. Wire cost is the sum of the estimated wire lengths of the net for which the current cell is a root and those nets for which current cell is a sink. As wire length estimation we use half-perimeter metric augmented by a net size coefficient from [12].

Timing cost is computed as the squared delay of the slowest path through the current cell if such delay approaches the critical delay (above 60% in our experiments) and zero otherwise (in this way, moves that are likely to make a near critical path worse are discouraged). The cost of a cell at particular location is a composite of timing and wire cost:

$$C = \alpha C_T + (1 - \alpha) C_W.$$

Gain of moving cell from current to new location is:

$$Gain = C_{curr} - C_{new}.$$

Once we have constructed the gain graph, we find the max-gain path in the graph and use target slot with the highest gain for ripple-move legalization. Note that to minimize perturbations of the placement we move cells at most one slot during a ripple move. Another motivation for this is that embedder has a much stronger algorithm for optimizing cell locations, so we want to keep cells as close to those locations as possible. Note that best gain value could still be negative (i.e. we may lose some quality/performance).

During ripple-moves it is possible that a cell may be moved to a slot which contains one of its logically equivalent cells. In that case we *unify* them and stop the current pass of a single overlap legalization.

```
Algorithm: RT-Embedding(C,P)
     C: Circuit; P: Placement
a1   while(improvement)
a2     Static_Timing_Analysis(C, P)
a3     T ← Extract_Replication_Tree(C, P)
a4     Embed_Tree(T)
a5     Post_Unify(C, P, T)
a6     Legalize(C, P)
a7   endwhile
```

**Figure 9: Replication Tree Embedding.**

# 5. OVERALL STRATEGY

Having introduced in previous sections the core components of our approach to replication, we now give an overview of how these components can be put together into a complete optimization flow. As in [1], we begin from a valid timing-driven placement produced by VPR [12].

Here is the top-level view of how our optimizer relates to VPR: VPR is invoked to give an initial placement; we optimize the placement by replication; we then give the result to the VPR detailed router to accurately assess the results.

Thus, our approach is not currently intended to replace any existing optimization steps in the flow but rather to complement them. The core replication procedure is focused on highly timing-critical sub-circuits and thus, while the embedding algorithm is nontrivial, the runtime penalty for using such a sophisticated algorithm is very small in the scope of the entire flow (this has been verified experimentally).

Figure 9 illustrates the pseudo-code of our main optimization loop. In each iteration we start with static timing analysis to identify the most critical sink. From an $\epsilon$-SPT, we extract a *Replication Tree* with the critical sink at its root. Then we pass the tree to embedder which will produce a family of solutions that trade off cost and delay. After the embedding phase we analyze circuit for possible post-process unifications (see section 6.2) since it is possible to have equivalent cells that are not exactly on top of each other, but close enough that unifying them would not harm timing. As a final step we invoke placement legalizer to resolve any possible cell overlaps that may have occurred.

# 6. DETAILS AND ENHANCEMENTS

## 6.1 Delay Model

In our experimental setup we use essentially the same placement-level delay estimator as used by VPR [12] and [1]. For the target FPGA architecture under consideration, all the switches are buffered and interconnect resources are uniform. As a result, RC effect are localized and thus the interconnect delay is reasonably approximated by a linear function of the Manhattan length of the interconnect. As an aside, we note that in principle, the embedding algorithm can use more general delay models.

## 6.2 Post-process Unification

Once we have an embedding of a replication tree, if we seek a timing superior solution, some cells may be placed close to logically equivalent cells but not quite on top of them. In this case implicit cell unification will not occur. However, it is possible that some of the equivalent cells lie on non-critical paths and that their child cells could pick up signal from the newly replicated cell without degrading their arrival time (sometimes delay can even improve).

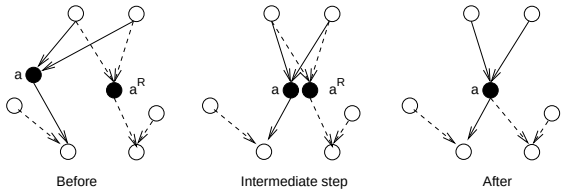


**Figure 10: Example of cell unification.**

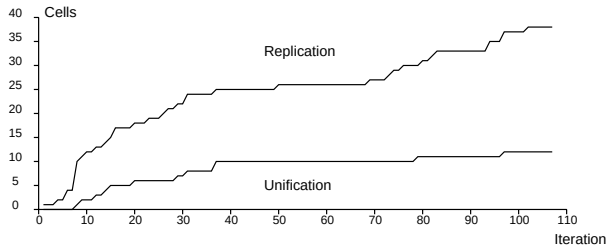


**Figure 11: Replication statistics for circuit ex1010.**

As a post-process, for each newly replicated cell we examine all their logically equivalent cells. If any fanout cell of those equivalent cells can improve its arrival time by taking the corresponding input from a newly replicated cell, we reassign it to the new replica. In this way we can improve delay on paths that were not explicitly captured by the replication tree. It is possible that in this process some of the equivalent cells remain without fanout (i.e., no cell is using their output). In this case such cell is deleted as redundant. Once we delete a cell we have to examine child count of its parents since deleted cell could have been the only child of its parent cell and then parent itself becomes redundant. This test is applied recursively up the path.

An example of this scenario in practice is when we have a non-tree structure (DAG) on one side of the FPGA. Then in each iteration a part of the DAG is extracted as a replication/fanin tree, optimized and placed further away so that replication must occur. In consecutive iterations the other parts of the DAG slowly migrate to the other side. Finally, the entire DAG can migrate to the other side, in which case replications, although necessary for intermediate solution, are now completely redundant. *Unification* naturally handles this anomaly. Figure 10 shows an example of unification. Before optimization we had cell $a$ and its replica $a^R$. Cell $a$ gets relocated to proximity of cell $a^R$. Timing analysis reveals that children of $a^R$ can get signal from $a$ without degrading worst delay through it so we perform unification.

Figure 11 shows relation between replicated and unified cells for circuit `ex1010`. The optimization took 106 loop iterations and during that time 38 cells were replicated but 12 were unified giving total of 26 replications at the end.

## 6.3 $\epsilon$-Slowest Paths Tree

As discussed previously, we use the $\epsilon$-SPT to guide replication tree construction. The value of $\epsilon$ is initially set to zero and is dynamically updated in the main loop of optimization flow. Since our approach has no randomized components, when no improvement is found for a tree rooted at particular critical sink, we would not be able to improve any further in subsequent iterations since the same sink will still be critical and the same tree will be selected. We address this problem by dynamically increasing the value of $\epsilon$ when non-improvement occurs. The intuition is that extracted tree will become larger and larger solution space gives more freedom in tree embedding optimization.

## 6.4 Flip-Flop relocation

If the circuit that we are optimizing contains FFs, it is possible that placement of those FFs is the limiting factor for further optimization. To address this issue we use a feature of the S-Tree algorithm [5] which performs simultaneous driver placement at no extra run-time overhead; in our case, this translates to simultaneous sink placement. Due to the deterministic nature of the approach, if we are not able to improve timing of particular critical sink it will become selected again as critical in next iteration of main optimization loop. If this occurs and the critical sink is a FF, then we allow it to move. We extract the trade-off curve which is composed of solutions at all possible placement locations for critical sink (not just the initial location) and choose the solution minimizing the arrival time without introducing large delay penalty on other paths that touch the FF that are not captured by the optimized tree (for example nets for which this critical FF is a source). However we do allow some degradation of solutions, and in the main loop we save the best solution seen until this point, so that we can always report the best solution encountered instead of the last one which could saturate in a local optimum.

# 7. EXPERIMENTS

We have implemented the Replication Tree Embedding algorithm and performed some initial experiments to evaluate its effectiveness. The experiments were conducted in a LINUX environment on a PC with an Intel PentiumM 1.3GHz CPU and 256MB of RAM. The main criteria of interest are the maximum delay through the circuit (i.e., clock period), wire length and number of logic blocks. All such statistics are reported by the VPR timing-driven router. We compared our approach with Timing Driven VPR [12] and with the local replication algorithm from [1]. Table 1 shows the experimental results for 20 MCNC benchmark circuits. As mentioned earlier, we used timing driven VPR to place the circuits. In the first data set we did not perform any additional optimizations. In the second data set we optimized placement by local replication[3] algorithm, and in the third data set we optimized placement using our approach (RT-Embedding). All placements were routed using VPR in timing driven mode. Since the local replication algorithm is randomized, we ran it three times and took the best result. The circuits were placed on the minimum square FPGA able to contain the circuit. As in [12] we use definition of low-stress routing as routing where FPGA has about 20% more routing resources available then the minimum required to successfully route the circuit. Also from [12], infinite-resource routing is when FPGA has unbounded routing resources. It is argued in [12] that the former represents the situation how FPGA will be routed in practice and the latter is a good placement evaluation metrics. For post-place-and-route experiments we present both low-stress ($W_{ls}$) and infinite-resource ($W_\infty$) critical path delay numbers. Results for local replication and RT-Embedding are normalized to VPR results.

[3]We have verified with the authors of [1] that results reported in [1] were based on wire-length-driven VPR (by mistake) while it was reported that comparisons were based on timing-driven VPR.

**Table 1: Comparison between Timing-Driven VPR, local replication and RT-Embedding**

| Circuit | Timing Driven VPR | | | | local replication normalized to VPR | | | | RT-Embedding normalized to VPR | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | crit path [ns] | | wire | | crit path | | wire | | crit path | | wire | |
| | $W_\infty$ | $W_{ls}$ | length | blk | $W_\infty$ | $W_{ls}$ | length | blk | $W_\infty$ | $W_{ls}$ | length | blk |
| ex5p | 80.59 | 81.99 | 20020 | 1135 | 0.792 | 0.806 | 1.027 | 1.004 | 0.764 | 0.774 | 1.090 | 1.011 |
| tseng | 50.54 | 53.65 | 10495 | 1221 | 0.987 | 0.955 | 1.012 | 1.004 | 0.987 | 0.978 | 1.060 | 1.002 |
| apex4 | 72.12 | 75.41 | 22332 | 1290 | 0.912 | 0.913 | 1.042 | 1.012 | 0.888 | 0.913 | 1.107 | 1.011 |
| misex3 | 64.44 | 65.87 | 21784 | 1425 | 0.914 | 0.937 | 1.013 | 1.007 | 0.852 | 0.891 | 1.148 | 1.010 |
| alu4 | 77.20 | 81.07 | 20796 | 1544 | 0.987 | 0.963 | 1.004 | 1.000 | 0.922 | 0.925 | 1.053 | 1.002 |
| diffeq | 55.29 | 57.49 | 15560 | 1600 | 1.004 | 1.000 | 1.002 | 1.003 | 0.989 | 0.969 | 1.026 | 1.001 |
| dsip | 65.38 | 67.21 | 17237 | 1796 | 0.924 | 0.938 | 1.024 | 1.001 | 0.793 | 0.804 | 1.277 | 1.001 |
| seq | 76.93 | 77.82 | 28493 | 1826 | 0.939 | 0.969 | 1.011 | 1.002 | 0.870 | 0.885 | 1.048 | 1.003 |
| apex2 | 94.61 | 95.47 | 30998 | 1919 | 1.000 | 1.000 | 1.000 | 1.000 | 0.811 | 0.838 | 1.120 | 1.010 |
| s298 | 124.20 | 127.35 | 22762 | 1941 | 0.937 | 0.937 | 1.029 | 1.003 | 0.915 | 0.903 | 1.034 | 1.001 |
| des | 90.44 | 91.31 | 27415 | 2092 | 0.898 | 0.895 | 1.044 | 1.003 | 0.876 | 0.876 | 1.039 | 1.001 |
| bigkey | 59.69 | 60.65 | 21074 | 2133 | 1.000 | 1.000 | 1.000 | 1.000 | 0.855 | 0.892 | 1.190 | 1.000 |
| frisc | 119.02 | 124.61 | 61109 | 3692 | 1.007 | 0.997 | 1.007 | 1.001 | 0.999 | 0.983 | 1.018 | 1.001 |
| spla | 111.03 | 113.57 | 68308 | 3752 | 0.874 | 0.889 | 1.035 | 1.005 | 0.812 | 0.824 | 1.108 | 1.008 |
| elliptic | 105.96 | 108.50 | 47456 | 3849 | 0.926 | 0.934 | 1.040 | 1.003 | 0.853 | 0.838 | 1.030 | 1.001 |
| ex1010 | 184.84 | 185.56 | 70300 | 4618 | 0.861 | 0.882 | 1.044 | 1.003 | 0.818 | 0.847 | 1.148 | 1.006 |
| pdc | 167.81 | 169.33 | 105073 | 4631 | 0.707 | 0.728 | 1.031 | 1.003 | 0.641 | 0.707 | 1.072 | 1.005 |
| s38417 | 97.20 | 100.61 | 64490 | 6541 | 0.974 | 0.961 | 1.004 | 1.000 | 0.930 | 0.944 | 1.017 | 1.000 |
| s38584.1 | 99.74 | 102.10 | 58869 | 6789 | 0.919 | 0.927 | 1.002 | 1.000 | 0.842 | 0.839 | 1.048 | 1.001 |
| clma | 211.78 | 217.24 | 145551 | 8527 | 0.926 | 0.915 | 1.021 | 1.003 | 0.746 | 0.745 | 1.053 | 1.005 |
| **average** | | | | | **0.925** | **0.927** | **1.020** | **1.003** | **0.858** | **0.869** | **1.084** | **1.004** |

We are able to improve critical path delay over VPR for all circuits in the test suite. The best delay reduction of 36% was achieved for circuit pdc. Average delay reduction is 14.2%, which almost doubles the average delay improvement of the local replication algorithm. The largest improvement over local replication is almost 19% for circuit apex2, for which local replication was not able to improve critical path delay at all. Observe that wire-length degradation of our approach is 8.4% on average, and average number of newly introduced cells by replication is only 0.4% of the total number of cells. One may argue that the increase in wire length is not negligible. However, perhaps more important than wire length is routability and our designs were always successfully routed (this is most relevant in the case of $W_{ls}$).

Runtime overhead of our approach is very modest – under 5% of the time of VPR flow (place and route). Note that low-stress routing critical path delay is slightly worse that the case with infinite routing resources. Degradation is consistent for all circuits in the test suites and also correlates with low-stress routing behavior conclusions from [12].

# 8. CONCLUSIONS

We have presented a general and robust approach to timing-driven, placement-coupled replication. Two items form the core of the approach. First we presented an efficient algorithm for optimal fanin tree embedding under a general cost model. Second we proposed the *replication tree* for inducing large sub-circuits which can be optimized by the embedder. The approach has a number of interesting properties including implicit unification of logically equivalent cells.

Around these core ideas we have built an optimization engine for the FPGA domain and demonstrated very promising preliminary experimental results.

Finally, we argue that the general ideas in this paper hold great promise beyond the context studied here. We suggest that the techniques can provide useful bridges between placement, routing and logic (re-)synthesis. Further, the graph-based modeling of the placement target would seem ideally suited to many practical problems (e.g., placement in the context of heterogeneous FPGA routing architectures).

# 9. REFERENCES


[1] G. Beraudo, J. Lillis, "Timing Optimization of FPGA Placements by Logic Replication," DAC, 2003.

[2] W. Gosti, A. Narayan, R.K. Brayton, A.L. Sangiovanni-Vincentelli, "Wireplanning in logic Synthesis," ICCAD, 1998.

[3] W. Gosti, S.P. Khatri, A.L. Sangiovanni-Vincentelli, "Addressing The Timing Closure Problem By Integrating Logic Optimization and Placement," ICCAD, 2001.

[4] S. Devadas, A. Ghosh, K. Keutzer, "Logic Synthesis," McGraw-Hill, 1994.

[5] M. Hrkić, J. Lillis, "S-Tree: A Technique for Buffered Routing Tree Synthesis," DAC, 2002.

[6] M. Hrkić, J. Lillis, "Buffer Tree Synthesis With Consideration of Temporal Locality, Sink Polarity Requirements, Solution Cost, Congestion and Blockages," IEEE Transactions on CAD, 2003.

[7] S.W. Hur, J. Lillis, "Mongrel: Hybrid Techniques for Standard Cell Placement," ICCAD, 2000.

[8] M. Jackson, E. Kuh, "Performance-driven Placement of Cell Based IC's," DAC, 1989.

[9] J. Lillis, C.K. Cheng, T.T.Y. Lin, "Algorithms for Optimal Introduction of Redundant Logic for Timing and Area Optimization," Proc. IEEE International Symposium on Circuits and Systems, 1996.

[10] L.T. Liu, M.T. Kuo, C.K. Cheng, T.C. Hu, "A Replication Cut for Two-Way Partitioning," IEEE Transactions on CAD, 1995.

[11] W.K. Mak, D.F. Wong, "Minimum Replication Min-Cut Partitioning," IEEE Transactions on CAD, October 1997.

[12] A. Marquardt, V. Betz, J. Rose, "Timing-Driven Placement for FPGAs," International Symposium on FPGAs, 2000.

[13] A. Srivastava, R. Kastner, M. Sarrafzadeh, "Timing Driven Gate Duplication: Complexity Issues and Algorithms," ICCAD, 2000.