Safe Delay Optimization for Physical Synthesis

Kai-hui Chang, Igor L. Markov, Valeria Bertacco
EECS Department, University of Michigan, Ann Arbor, MI 48109-2122
{changkh, imarkov, valeria}@umich.edu

Abstract — Physical synthesis is a relatively young field in Electronic Design Automation. Many published optimizations for physical synthesis end up hurting the final result, often by neglecting important physical aspects of the layout, such as long wires or routing congestion. In this work we propose SafeResynth, a safe resynthesis technique, which provides immediately-measurable delay improvement without altering the design's functionality. It can enhance circuit timing without detrimental effects on route length and congestion. When applied to IWLS'05 benchmarks, SafeResynth improves circuit delay by 11% on average after routing, while increasing route length and via count by less than 0.2%. Our resynthesis can also be used in an unsafe mode, akin to more traditional physical synthesis algorithms popular in commercial tools. Applied together, our safe and unsafe transformations achieve 24% average delay improvement for seven large benchmarks from the OpenCores suite. The relative contribution of safe and unsafe techniques varies depending on the amount of whitespace in the layout.

I. Introduction

Timing optimization of digital logic is gaining importance with each technology step, as interconnect contributes a larger fraction of critical-path delay due to its poor scaling. Since accurate timing information can only be obtained after the circuit is placed, post-placement timing optimization has been studied extensively. Most techniques either modify the logic or change the physical aspects of the circuit [9]. Physical solutions include net buffering, gate sizing [15] and gate relocation [1]. Logical solutions include gate replication [12], rewiring [6, 7] and restructuring [5, 16, 19, 22]. Techniques based on a placement or routing with the goal to improve timing are often called physical synthesis.

We observe that a number of previous works on physical synthesis do not provide an overall improvement because when optimizing one aspect of the design, they may damage other aspects. For example, uncontrollable logic cloning may increase area and route length, making critical nets longer than expected during placement and routing [12]. Indiscriminate buffering may also create many gate overlaps, leading to potentially detrimental effects on circuit timing when overlaps are resolved [17]. A number of related works try to solve this problem. For example, Li et al. [17] proposed an incremental placement algorithm which maintains the stability of a placement for gate sizing and buffer insertion, while Luo et al. [20] and Brenner et al. [3] addressed this problem by designing legalizers that seek to preserve performance metrics.

Timing-driven placement also suffers similar stability problems. While published papers typically report timing estimates before routing, critical nets often detour during routing, thus aggravating performance compared with traditional placement. As a result, place-and-route tools sometimes produce *better* results when the timing-driven mode is *disabled*. In practice, the best bet to improve timing is to try as many timing-driven and non-timing-driven tools as possible. As the empirical results in [13, 14] suggest, no placer – commercial or academic – dominates on all benchmarks.

In our work we propose *SafeResynth*, a safe and powerful physical synthesis technique, based on simulation and iterative equivalence checking. By broadening the set of transformations, SafeResynth possesses more optimization power than many existing techniques. In addition, it provides immediately-measurable delay improvement on every step without detrimental effects on other circuit parameters. As a result, it can improve circuit delay considerably with very little risk of destabilizing an existing design flow or hampering timing closure, a common problem with new ideas in physical synthesis. Figure 1 shows two examples of our optimizations. In Figure 1(a), the signal that drives $g8$ is resynthesized using gates located closer to it, and a new gate is added to replace the old one. In Figure 1(b), one gate ($g8$) originally driven by $g6$ uses gate *new* as its new source, while the other gate ($g1$) is still driven by $g6$. Since the new gates are placed on previously unused sites, they do not overlap with old gates. Empirical results show that our technique can improve delay by 11% while route length and via count increase by less than 0.2%.

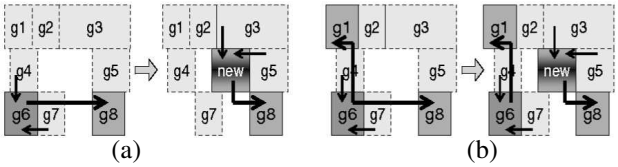

Fig. 1. Example transformations for row-based standard-cell layout: (a) resynthesized gate *new* replaces $g6$ to drive $g8$, (b) gate cloning uses resynthesized gate *new* to drive $g8$, while the original driver $g6$ continues to drive $g1$.

The rest of this paper is organized as follows. In Section II we propose a new powerful and safe timing optimization approach. Several aspects of our technique are analyzed in Section III. Experimental results are reported in Section IV, and Section V concludes this paper.

II. Our SafeResynth Technique

Our safe physical synthesis approach, SafeResynth, is discussed in detail in this section. It uses *signatures* produced by simulation to identify potential resynthesis opportunities, whose correctness is then verified by equivalence checking [23]. Since our goal is layout optimization, we can prune some

of the opportunities based on their promise before formally verifying them since verification is relatively slow. To this end, we propose pruning techniques based on physical constraints and logical *compatibility* among *signatures*. SafeResynth is powerful in that it does not restrict resynthesis to small geometric regions or small groups of adjacent wires. It is safe because the produced placement is always legal and the delay improvement can be evaluated immediately.

A. Terminology

We define a *signature* as a bit-vector of simulated values of a wire. Given the signature s_t of a wire w_t to be resynthesized, and a certain gate g_1, a wire w_1 with signature s_1 is said to be *compatible* with w_t if it is possible to generate s_t using g_1 with signature s_1 as one of its inputs. In other words, it is possible to generate w_t from w_1 using g_1. For example, if $s_1 = 1$, $s_t = 1$ and $g_1 = AND$, then w_1 is compatible with w_t using g_1 because it is possible to generate 1 on an AND's output if one of its inputs is 1. However, if $s_1 = 0$, then w_1 is not compatible with w_t using g_1 because it is impossible to obtain 1 on an AND's output if one of its inputs is 0 (see Figure 4).

A *controlling value* of a gate is the value that fully specifies the gate's output when applied to one input of the gate. For example, 0 is the controlling value for AND because when applied to the AND gate, its output is always 0 regardless of the value of other inputs. When two signatures are *incompatible*, that can often be traced to a *controlling value* in some bits of one of the signatures.

B. SafeResynth Framework

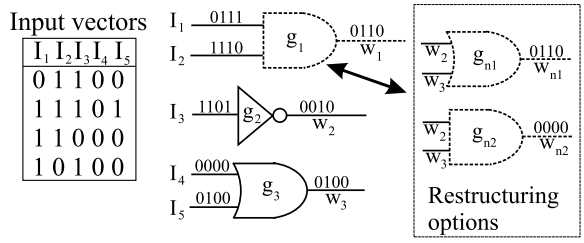

Fig. 2. A restructuring example. Input vectors to the circuit are shown on the left. Each wire is annotated with its bit-signature computed by simulation on those test vectors. We seek to compute signal w_1 by a different gate, e.g., in terms of signals w_2 and w_3. Two such restructuring options (new gates) are shown as g_{n1} and g_{n2}. Since g_{n1} produces the required signature, equivalence checking is performed between w_{n1} and w_1 to verify the correctness of this restructuring. Another option, g_{n2}, is abandoned because it fails our compatibility test.

The SafeResynth framework is given in Figure 3, and an example is given in Figure 2. Initially, *library* contains all the gates to be used for resynthesis. We first generate a signature for each wire by simulating certain input patterns, whose selection will be discussed in detail in Section II-D. In order to optimize timing, $wire_t$ in line 2 will be selected from wires on the critical paths in the circuit. Line 3 restricts our search of potential resynthesis opportunities according to certain physical constraints, and lines 4-5 further prune our search space based on logical correctness. After a valid resynthesis option is found, we try placing the gate on various overlap-free sites close to a desired location in line 6 and check their timing improvements. In this process, more than one gate may be added if there are multiple sinks for $wire_t$, and the original driver of $wire_t$ may be replaced. We only call equivalence checking when we found certain changes that improve timing because formal verification is time-consuming. In line 10 we remove redundant gates and wires that may appear because $wire_t's$ original driver may no longer drive any wire, which often initiates a chain of further simplifications.

1. Simulate patterns and generate a signature for each wire.
2. Determine $wire_t$ to be resynthesized and retrieve $wires_c$ from the circuit.
3. Prune $wires_c$ according to physical constraints.
4. Foreach $gate \in library$ with inputs selected from combinations of compatible wires $\in wires_c$.
5. Check if $wire_t$'s signature can be generated using $gate$ with its inputs' signatures. If not, try next combination.
6. If so, do restructuring using $gate$ by placing it on overlap-free sites close to the desired location.
7. If timing is improved, check equivalency. If not equivalent, try next combination of wires.
8. If equivalent, a valid restructuring is found.
9. Use the restructuring with maximum delay improvement for resynthesis.
10. Identify and remove gates and wires made redundant by resynthesis.

Fig. 3. The SafeResynth framework.

C. Search-Space Pruning Techniques

In order to resynthesize a target wire ($wire_t$) using an n-input gate in a circuit containing m wires, the brute force technique needs to check $\binom{m}{n}$ combinations of possible inputs, which can be very time-consuming for $n > 2$. Therefore it is important to prune the number of wires to try.

When the objective is to optimize timing, the following physical constraints can be used in line 3 of the framework: (1) wires with arrival time later than that of $wire_t$ are discarded because resynthesis using them will only increase delay, and (2) wires that are too far away from the sinks of $wire_t$ are abandoned because the wire delay will be too large to be beneficial. We set this distance threshold to twice the HPWL (Half-Perimeter Wirelength) of $wire_t$.

In line 4 logical compatibility is used to prune the wires that need to be tried. Wires not compatible with $wire_t$ using $gate$ are excluded from our search. Figure 4 summarizes how compatibilities are determined given a gate type, the signatures of $wire_t$ and the wire to be tested ($wire_1$).

Gate type	$wire_t$	$wire_1$	Result
NAND	0	0	Incompatible
NOR	1	1	Incompatible
AND	1	0	Incompatible
OR	0	1	Incompatible
XOR/XNOR	Any	Any	Compatible

Fig. 4. Conditions to determine compatibility: $wire_t$ is the target wire, and $wire_1$ is the potential new input of the resynthesized gate.

To accelerate compatibility testing, we use the "one-count", i.e., the number of 1s in the signature, to filter out unpromising candidates. For example, if $gate$==OR and the one-count of $wire_t$ is smaller than that of $wire_1$, then these two wires are

incompatible because OR will only increase one-count in the target wire. This technique can be applied before bit-by-bit compatibility test to detect incompatibility faster.

Our *pruned_search* algorithm that implements lines 4-5 of the framework is outlined in Figure 5. The algorithm is specifically optimized for two-input gates but can be extended to gates with more than two inputs. $Wire_t$ is the target wire to be resynthesized, $wires_c$ are wires selected according to physical constraints, and *library* contains gates used for resynthesis. All wires in the fanout cone of $wire_t$ are excluded in the algorithm to avoid formation of combinational loops.

```
Function pruned_search(wire_t, wires_c, library)
1  foreach gate ∈ library
2   wires_g = compatible(wire_t, wires_c, gate);
3   foreach wire_1 ∈ wires_g
4    wires_d = get_potential_wires(wire_t, wire_1, wires_g, gate);
5    foreach wire_2 ∈ wires_d
6     Restructure using gate, wire_1 and wire_2;
```

Fig. 5. The *pruned_search* algorithm.

In Figure 5, function *compatible* returns wires in $wires_g$ that are compatible with $wire_t$ given *gate*. Function *get_potential_wires* returns wires in $wires_d$ that are capable of generating the signature of $wire_t$ using *gate* and $wire_1$, and its algorithm is outlined in Figure 6. For XOR/XNOR, the signature of the other input can be calculated directly, and wires with signatures identical to that signature are returned using the signature hash table. For other gate types, signatures are calculated iteratively for each wire (denoted as $wire_2$) using $wire_1$ as the other input, and the wires that produce signatures which match $wire_t'$*s* are returned.

```
Function get_potential_wires(wire_t, wire_1, wires_g, gate)
1  if gate == XOR/XNOR
2   wires_d= sig_hash[wire_t.signature XOR/XNOR
      wire_1.signature];
3  else
4   foreach wire_2 ∈ wires_g
5    if wire_t.signature ==
       gate.evaluate(wire_1.signature, wire_2.signature)
6     wires_d ← wires_d ∪ wire_2;
7  return wires_d;
```

Fig. 6. Algorithm for function *get_potential_wires*. XOR/XNOR is considered separately because the required signature can be calculated uniquely from $wire_t$ and $wire_1$.

The effectiveness of our search-space pruning techniques is supported by our empirical results. For example, in the worst case (MEM_CTRL) 7,560 equivalence checking steps are performed during resynthesis. However, it is far smaller than the number of resynthesis options in the search space (about 1 billion), indicating that our techniques are effective in pruning unpromising resynthesis opportunities.

D. Implementation Insights

In our implementation, we select desired locations for placing the restructured gates with the following criterion: the first 200 overlap-free slots closest to the Center Of Gravity (COG) of the new gate's input and output wires' COG. Although better initial guesses may exist for desired locations than the COG, they are not necessary because a fairly large number of valid locations will be evaluated rigorously. As a result, having an extremely accurate initial guess is not necessary to find the actual best location.

The performance of our algorithm is greatly influenced by the quality of the signatures generated by simulation. Poor signatures cannot distinguish many different wires and require additional calls to equivalence-checking. On the other hand, potentially resynthesizable wires can usually be distinguished from those not resynthesizable if their signatures are different. In light of this, we enhanced the FRAIG package in ABC [23] to dump its patterns and use them for our initial simulation. The purpose of the patterns in ABC is to distinguish different nodes in the AIG (And-Inverter Graphs) netlist built from the circuit, therefore they are also suitable for generating signatures that can distinguish different wires. In particular, if the FRAIG package is run with infinite backtrack limit, at least one simulation vector will exist to distinguish every two nodes. Currently, FRAIGs first simulate 2048 random patterns. Next, they append the counterexamples returned during equivalence checking and their variants as additional simulation patterns.

III. Analysis of Our Approach

Several aspects of our approach are discussed in this section, including its scalability, optimization power, safeness, advantages and limitations.

Scalability: Suppose that there are m wires in the circuit and g n-input gates are used for resynthesis, then the worst case time complexity of our resynthesis algorithm is on the order of $g \times m^n$ if $n \leq m/2$. However, by using physical constraints and logical pruning techniques, as well as several other heuristics, the time complexity is reduced significantly in practice. From our experimental results, we observe that the runtime is somewhere between linear and quadratic for $n = 2$. For example, a netlist with almost 100K nets can be resynthesized in 24 minutes (the largest benchmark in Table I). We have also developed pruning techniques for multi-input gates, but omit them due to space limitations.

Aside from runtime, the use of signatures instead of other logic representations, such as BDDs, makes our approach more scalable in terms of memory usage. For example, comparable methods to find resynthesis opportunities in [11, 18] are evaluated for at most 5K gates at a time, whereas our techniques typically handle 100K-gate circuits in minutes. Commercial tools often use BDDs but achieve scalability by means of (i) netlist partitioning, and (ii) restricting logic optimization to small windows. To this end, our main contribution is a relatively simple framework that is fast and naturally scales to large designs without netlist partitioning or windowing.

Optimization Power and Safeness: Our resynthesis technique tries to reproduce a signal using gates in the library with new inputs selected from the whole circuit, therefore it is essentially a form of technology mapping. Since the selection is not limited by small windows like in previous restructuring techniques [5], it is capable of finding optimizations that are long-range. Furthermore, complete controllability don't-cares

are automatically utilized in our techniques by construction, while no explicit don't-care computations [21] are required. These don't-cares also give our technique more optimization power to find restructuring opportunities.

When we try to resynthesize a wire, we either remove a gate and drive all the relevant sinks by a new gate or speed up the propagation of the signal to the sinks of the wire. The former case subsumes simple gate relocation, gate relocation that simultaneously changes gate type, and also several types of traditional restructuring. The latter case subsumes single-gate logic replication, including the possibility of gate relocation and changing the gate type immediately after cloning.

All our transformations are safe in that no gates will be overlapped by our optimization. They also have limited effect on congestion because gates may be removed after each transformation, making white-space almost equal or even better after resynthesis. Furthermore, it is easy to veto transformations that violate designer-specified constraints or worsen designer-specified quality metrics, e.g., involve wires crossing obstacles, increase gate area or aggravate routing congestion. By making sure that every transformation improves major quality metrics without introducing new violations, we ensure that our resynthesis techniques are safe. On the other hand, by subsuming and generalizing several existing techniques they achieve considerable strength in practice.

Advantages and Limitations: In summary, the advantages of the resynthesis approach proposed include:

- The number of physical violations does not increase, and major physical parameters are not worsened.
- Long-range optimizations can be considered, and complete controllability don't-cares are exploited.
- Gate relocation and replication techniques can be subsumed whenever beneficial.
- A form of technology mapping is included.
- Our experimental results show a 11% delay improvement as measured after routing with less than 0.2% increase in route length and via count on average, which confirms that our technique is both powerful and safe.

Our technique does not improve standard arithmetic circuits because they are already heavily optimized. Nonetheless, our technique can be very helpful for large netlists automatically synthesized from HDL descriptions.

IV. Experimental Results

We implemented our techniques in C++ including a simple incremental Static Timing Analysis (STA) engine for our experiments. In our benchmarks, gate delays range from 0.025ns to 0.15ns, the unit capacitance is $131.53pF/m$ and unit resistance is $337K\Omega/m$. The driver resistance ranges from $2.5K\Omega$ to $10K\Omega$, and the port capacitance is $0.0149pF$. These parameters are based on a 0.18μm technology library, and we expect greater delay improvements as wire delays become more significant in newer technologies. Our delay model is based on the D2M formulas from [2], and we apply those formulas to Rectilinear Steiner Minimal Trees (RSMTs) generated by the FLUTE package [8][1] or to actual net routes produced by an industry router. We perform our optimizations using the STA engine based on RSMT, and we route the resynthesized layout to measure the final timing based on actual net routes.

Our hardware platform is an AMD Opteron 880 workstation running Fedora 4 Linux. Our experiments use the min-cut placer Capo 10 from the University of Michigan [4], the QPlace 5.2 placer from Cadence Design Systems, the NanoRoute 4.1 router also from Cadence and the FLUTE RSMT package from GSRC Bookshelf [8]. Simulation patterns are generated by the ABC package from UC Berkeley [23], and all transformations are verified by an external equivalence checker based on the MiniSat SAT solver [10]. While our organization licenses a broad range of EDA software, we do not currently have access to relevant physical synthesis tools. However, we do not believe that our techniques are used by EDA vendors, and would be willing to perform empirical comparisons when appropriate tools become available to us. We expect that our tools will achieve comparable improvements, but run considerably faster.

Our initial testcases are selected from IWLS2005 benchmarks [24], where the design utilization is 70%, but for experiments in Table II we varied the amount of whitespace. They belong to the following suites: OpenCores (SPI, DES_AREA, TV80, SYSTEMCAES, MEM_CTRL, AC97, USB, PCI, AES, WB_CONMAX, Ethernet and DES_PERF), Faraday (DMA), ITC99 (b14, b15, b17, b18 and b22) and ISCAS89 (s35932 and s38417). The benchmarks in the OpenCores suite are produced by a quick synthesis run of Cadence RTL Compiler, and all the benchmarks are mapped to a $0.18\mu m$ library. Our current implementation can only generate two-input NAND, NOR, AND, OR and XOR gates, as well as their variants where one of the inputs is inverted. In particular, if a three-input gate can be replaced by a two-input gate, our technique will find this restructuring opportunity. Although the netlists used in our experiments have multi-input cells, such as AOI, we do not need to break them down into smaller cells. Multiple gate cloning is not yet supported in the current implementation. As a result, area utilization remains roughly the same after resynthesis.

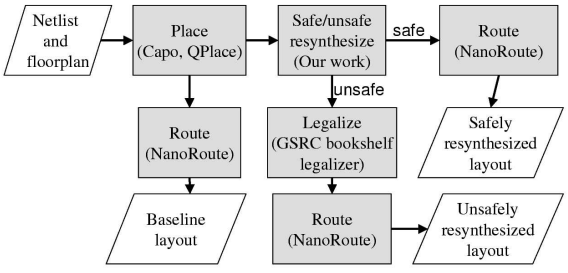

Fig. 7. Flow chart of our resynthesis experiments.

Our experimental flow is summarized in Figure 7. Three iterations of the resynthesis are carried out for each run, and the maximum number of resynthesis attempts for each wire is limited to 1,000 to further reduce runtime. Characteristics

[1]Minimal Steiner tress sometimes provide unnecessarily large source-to-sink delays, and our framework can use a drop-in replacement for timing-driven Steiner trees.

of the benchmarks and our empirical results are summarized in Table I, where the numbers are averages over three independent runs. Although we performed experiments using both Capo and QPlace, we only report the results produced by Capo due to space limitations. However, results from QPlace show similar trends. In addition to benchmarks in the table, our technique has shown similar performance on other netlists.

From the results, we observe that our approach is effective in reducing the delay for most of the benchmarks with minor increase in total route length, and sometimes it even results in route length reduction. The average delay improvement is 12% before routing and 11% after routing, while the route length and via counts increase by less than 0.2% on average. This is remarkable, compared to the results for logic cloning in [12] where route length increases by 2-28%. The results also show that our SafeResynth approach works most effectively for the OpenCores benchmarks (SPI to DES_PERF), because they are generated by quick synthesis without optimization. For example, the delay improved by 86% for the Ethernet benchmark, suggesting that our technique is effective when applied by itself. However, our technique still achieves up to 17% delay improvement when applied to already optimized benchmarks (DMA to s38417), indicating that it can augment traditional optimization techniques for further improvement.

The impact of our techniques is illustrated in Figure 8: (a) the detour of the critical path is reduced, which also reduces the maximum delay; and (b) our resynthesis technique found another source to generate the same signal. Although the new path is longer, the delay is actually reduced.

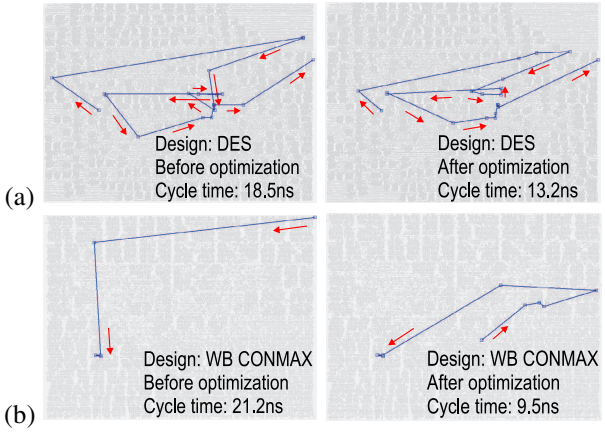

Fig. 8. Two optimization examples, one critical path per plot. Delay calculations are at the 0.18μm technology node. In (a) the critical path is shortened. In (b) an alternative source to generate the same signal is found. Although the new path is longer, the delay is actually reduced.

Traditional physical synthesis techniques are unsafe in that they allow cell overlaps during optimization and rely on a legalizer to remove the overlaps. Since accurate analysis cannot be performed immediately, the executed optimizations may worsen other physical parameters after legalization and routing. In order to compare safe and unsafe optimizations, we apply our resynthesis technique in an unsafe way by allowing cell overlaps. In our unsafe resynthesis, the location to place the resynthesized gate is determined by trying 400 sites near the desired coordinate regardless whether these sites are overlap-free or not. We used the legalizer provided by GSRC Bookshelf [25] in our experiments, and noticed that its runtime is typically short. In addition to performing safe and unsafe resynthesis separately, we combined both techniques by performing safe resynthesis after unsafe resynthesis in the hope of leveraging both their advantages. While this experiment does not cover all possible safe and unsafe techniques, we believe that it is representative. Because benchmarks that are only slightly modified cannot reflect the difference between safe and unsafe resynthesis, we use seven large benchmarks from OpenCores in this experiment, whose netlists are more significantly altered. In order to study the effects of available whitespace on the optimization results, we conducted the same experiments on the same designs with different percentages of whitespace. The results are summarized in Table II.

The comparison of estimated delay improvement between safe and unsafe resynthesis in Table II shows that unsafe resynthesis provides more improvement before legalization because the resynthesized gate is placed at the best location. However, the improvement reduces after legalization and becomes close to the improvement achieved by safe resynthesis. This shows that performing our resynthesis technique in a safe way, instead of the traditional unsafe way, does not result in any loss in its optimization strength. In addition, performing safe optimizations avoids the detrimental effects that worsen other physical parameters. As can be observed from Table II, performing safe instead of unsafe resynthesis avoids the significant increase in via count. Furthermore, the comparison among different percentage of whitespace shows that the improvement is only slightly affected by the amount of whitespace. These observations suggest that our technique is both powerful and safe. To obtain the greatest improvement, the advantages of both safe and unsafe techniques should be leveraged. As suggested by the results in Table II, this goal can be achieved by applying safe resynthesis after unsafe resynthesis.

V. Conclusions

In this paper we proposed an algorithm for resynthesis, called SafeResynth, which provides powerful improvements in physical synthesis, while applying safe modifications. It utilizes simulation to generate a signature for each wire, and wires on the critical path are resynthesized using new gates with their inputs selected from compatible wires. On-line equivalence checking is then carried out to verify the correctness of logic transformations. Since we allow inserting additional gates only when unused space is available, the effects of the change can be evaluated immediately, and the detrimental effects on other physical parameters can be avoided. At the same time, the global search for candidate wires gives our technique the power to find long-range optimizations. Experimental results show that our technique can improve timing considerably without deteriorating other circuit parameters, such as route length and via count. As a result, our technique can be applied to practically any design flow without hampering its timing closure. In addition, these results also suggest that the stability of existing physical synthesis techniques may be improved by performing layout modifications in a safe way.

TABLE I

IMPROVEMENT ACHIEVED BY OUR TECHNIQUES: DELAYS, ROUTE LENGTHS AND VIA COUNTS FOR UNOPTIMIZED LAYOUTS ARE SHOWN, FOLLOWED BY RELATIVE DELAY IMPROVEMENT DUE TO RESYNTHESIS. SAFERESYNTH RUNTIME IS SHOWN IN THE LAST COLUMN.

Benchmark	Cell count	Net count	Original				Resynthesized				Runtime (min)
			Estimated delay (ps)	Routed delay (ps)	Route length (μm)	Via count	Estimated delay improv.	Routed delay improv.	Route length increase	Additional vias	
SPI	3227	3277	2922	2918	238056	22661	2.89%	2.84%	0.14%	0.14%	1.07
DES_AREA	4881	5122	4451	4440	285701	30269	1.24%	1.28%	0.19%	0.56%	0.66
TV80	7161	7179	5519	5507	506243	50951	12.23%	12.08%	0.25%	0.13%	1.77
SYSTEMCAES	7959	8220	4688	4687	783394	61878	2.94%	2.94%	0.04%	-0.12%	1.02
MEM_CTRL	11440	11560	5118	5093	1108789	90876	6.42%	6.54%	0.12%	0.24%	44.71
AC97	11855	11948	2307	2750	871881	85041	2.67%	1.56%	0.04%	-0.14%	0.58
USB	12808	12968	3173	3257	1000834	87536	5.21%	3.09%	0.06%	0.15%	1.36
PCI	16816	16990	3777	4430	1390256	122375	5.99%	0.00%	0.09%	0.10%	1.68
AES	20795	21055	4417	4391	1358891	131246	2.32%	2.25%	0.09%	-0.08%	2.63
WB_CONMAX	29034	30165	19367	19402	2750881	257579	61.37%	**61.29%**	0.19%	-0.19%	7.6
Ethernet	46771	46891	70789	70762	7686013	427475	85.66%	**85.61%**	0.04%	-0.14%	21.66
DES_PERF	89341	98576	11564	11542	7643762	595196	1.98%	1.93%	0.02%	0.01%	5.58
DMA	19118	19809	5016	5064	2055086	153406	3.33%	1.03%	0.01%	-0.03%	1.37
b14	8679	8716	6338	6306	703240	59178	3.66%	3.66%	0.04%	-0.03%	4.32
b15	12562	12605	4809	4793	1029036	94015	3.71%	3.63%	0.03%	-0.15%	2.22
b17	37117	37167	5761	5757	3192066	280868	5.26%	5.22%	0.00%	-0.07%	4.99
b18	92048	92214	10843	10820	7423151	686753	17.54%	**17.41%**	-0.04%	-0.07%	23.05
b22	28317	28354	7261	7240	2292342	192405	6.58%	6.46%	0.02%	-0.23%	7.75
s35932	7273	7599	4081	3942	769292	59307	9.11%	0.00%	0.05%	0.14%	0.31
s38417	8278	8309	2796	3195	562242	58937	2.38%	0.00%	0.06%	0.14%	0.94
Average							**12.12%**	**10.94%**	**0.07%**	**0.02%**	

TABLE II

A COMPARISON OF DELAY, ROUTE LENGTH AND VIA COUNT FOR LAYOUTS WITH DIFFERENT PERCENTAGE OF WHITESPACE USING SAFE, UNSAFE AND UNSAFE+SAFE RESYNTHESIS. UNSAFE OPTIMIZATIONS ALLOW CELL OVERLAPS, AND LEGALIZATION IS REQUIRED TO REMOVE THE OVERLAPS. THIS TABLE SHOWS THE AVERAGE RESULTS OF SEVEN LARGE BENCHMARKS FROM OPENCORES.

Percentage of whitespace	Estimated delay improvement				Routed delay improvement			Route length increase			Via count increase		
	Safe resynth.	Unsafe resynthesis		Unsafe + safe resynth.	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.
		Before legal.	After legal.										
30%	23.60%	24.22%	23.93%	24.22%	**22.25%**	**21.94%**	24.41%	0.08%	0.05%	0.08%	**-0.04%**	**2.03%**	1.80%
10%	23.59%	24.12%	23.64%	24.01%	**23.52%**	**23.56%**	23.98%	0.05%	0.09%	0.07%	**-0.01%**	**2.29%**	1.87%
3%	20.33%	20.78%	20.34%	21.63%	**20.22%**	**20.23%**	21.38%	0.04%	0.05%	0.05%	**0.15%**	**1.68%**	1.62%

REFERENCES

[1] A. H. Ajami and M. Pedram, "Post-Layout Timing-Driven Cell Placement Using an Accurate Net Length Model with Movable Steiner Points", *DAC'01*, pp. 595-600.

[2] C. J. Alpert, A. Devgan and C. Kashyap, "A two moment RC delay metric for performance optimization", *ISPD'00*, pp. 69-74.

[3] U. Brenner, A. Pauli and J. Vygen, "Almost Optimum Placement Legalization by Minimum Cost Flow and Dynamic Programming", *ISPD'04*, pp. 2-9.

[4] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?", *DAC'00*, pp. 693-698.

[5] C. Changfan, Y. C. Hsu and F. S. Tsai, "Timing Optimization on Routed Designs with Incremental Placement and Routing Characterization", *IEEE Trans. on CAD*, Feb. 2000, pp. 188-196.

[6] C. W. Chang et al., "Fast Postplacement Optimization Using Functional Symmetries", *IEEE Trans. on CAD*, Jan. 2004, pp. 102-118.

[7] S. C. Chang, L. Van Ginneken and M. Marek-Sadowska, "Circuit Optimization by Rewiring", *IEEE Trans. on Comp.*, Sep. 1999, pp. 962-969.

[8] C. Chu and Y.-C. Wong, "Fast and Accurate Rectilinear Steiner Minimal Tree Algorithm for VLSI Design", *ISPD'05*, pp. 28-35. `http://class.ee.iastate.edu/cnchu/flute.html`

[9] W. Donath et al., 'Transformational Placement and Synthesis", *DATE'00*, pp. 194-201.

[10] N. Eén and N. Sörensson, "An Extensible SAT-solver", *Theory and Applications of Satisfiability Testing, SAT*, 2003, pp. 502-518.

[11] S.-Y. Huang, K.-C. Chen and K.-T. Cheng, "AutoFix: A Hybrid Tool for Automatic Logic Rectification", *IEEE Trans. on CAD*, Sep. 1999, pp. 1376-1384.

[12] M. Hrkic, J. Lillis and G. Beraudo, "An Approach to Placement-Coupled Logic Replication", *DAC'04*, pp. 711-716.

[13] C. Hwang and M. Pedram, "Timing-Driven Placement Based on Monotone Cell Ordering Constraints", *ASPDAC'06*, pp. 201-206.

[14] A. B. Kahng and Q. Wang, "Implementation and Extensibility of an Analytic Placer", *IEEE Trans. on CAD*, May 2005, pp. 734-747.

[15] L. N. Kannan, P. R. Suaris and H. G. Fang, "A Methodology and Algorithms for Post-Placement Delay Optimization", *DAC'94*, pp. 327-332.

[16] V. N. Kravets and P. Kudva, "Implicit Enumeration of Structural Changes in Circuit Optimization", *DAC'04*, pp. 438-441.

[17] C. Li, C-K. Koh and P. H. Madden, "Floorplan Management: Incremental Placement for Gate Sizing and Buffer Insertion", *ASPDAC'05*, pp. 349-354.

[18] C.-C. Lin, K.-C. Chen and M. Marek-Sadowska, "Logic Synthesis for Engineering Change", *IEEE Trans. on CAD*, Mar. 1999, pp.282-202.

[19] A. Lu, H. Eisenmann, G. Stenz and F. M. Johannes, "Combining Technology Mapping with Post-Placement Resynthesis for Performance Optimization", *ICCD'98*, pp. 616-621.

[20] T. Luo, H. Ren, C. J. Alpert and D. Pan, "Computational Geometry Based Placement Migration", *ICCAD'05*, pp. 41-47.

[21] A. Mischenko and R. K. Brayton, "SAT-Based Complete Don't-Care Computation for Network Optimization", *DATE'05*, pp. 412-417.

[22] H. Vaishnav, C. K. Lee and M. Pedram, "Post-Layout Circuit Speed-up by Event Elimination", *ICCD'97*, pp. 211-216.

[23] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 51205. `http://www-cad.eecs.berkeley.edu/~alanmi/abc/`

[24] `http://iwls.org/iwls2005/benchmarks.html`

[25] UMICH Physical Design Tools, `http://vlsicad.eecs.umich.edu/BK/PDtools/`