SafeResynth: A New Technique for Physical Synthesis

Kai-hui Chang, Igor L. Markov and Valeria Bertacco

The University of Michigan, Department of EECS
2260 Hayward St., Ann Arbor, MI 48109-2121

January 9, 2008

Abstract

Physical synthesis is a relatively young field in Electronic Design Automation. Many published optimizations for physical synthesis end up hurting the quality of the final design, often because they neglect important physical aspects of the layout, such as long wires or routing congestion. Our work defines and explores the concept of physical safeness and evaluates empirically its impact on route length, via count and timing. In addition, we propose a new physically safe and logically sound optimization, called SafeResynth, which provides immediately-measurable improvements without altering the design's functionality. SafeResynth can enhance circuit timing without detrimental effects on route length and congestion. We achieve these improvements by performing a series of netlist transformations and re-placements that are individually evaluated for logical soundness (that is, they do not alter the logic functionality) and for physical safeness. When used alone, SafeResynth improves circuit delay of IWLS'05 benchmarks by 11% on average after routing, while increasing route length by less than 0.2%. Since transistors are not affected by SafeResynth, it can also be applied to post-silicon debugging, where only metal fixes are possible. [1]

Keywords: Circuit optimization, physical synthesis, post-silicon debugging, metal fix

1 Introduction

Circuit 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 has been placed, post-placement

[1] A preliminary version of this work has been presented at ASPDAC'07. New contributions in the current manuscript include: (1) a thorough discussion of the concept of physical safeness; (2) experiments to explore the factors that affect the safeness of a physical synthesis technique; and (3) a new application for post-silicon debugging and metal fix.

timing optimization has been studied extensively. Most techniques either modify the logic or change the physical aspects of the circuit [11]. Physical-level solutions include net buffering, gate sizing [18] and gate relocation [1]. Logic-level solutions include gate replication [14], rewiring [7, 9] and restructuring [6, 20, 23, 26]. Physical synthesis commonly encompasses those techniques which strive to improve circuit timing using placement or routing information.

A number of previous publications on physical synthesis do not actually provide an overall improvement because when optimizing one aspect of the design, they negatively impact other aspects. For instance, logic replication may increase area and route length, which, in turn, may generate critical-path nets longer than expected [14]. Indiscriminate buffering may also create many gate overlaps, leading to potentially detrimental effects on circuit timing when such overlaps are finally resolved [21]. A number of recent publications address precisely the issue of *stability* in physical synthesis optimizations. For example, Li *et al.* [21] propose an incremental placement algorithm which maintains the stability of placement for gate sizing and buffer insertion, while Luo *et al.* [24] and Brenner *et al.* [4] address this same problem by designing legalizer tools that seek to preserve performance metrics. However, these approaches tend to break down in later stages of physical design, when circuits are heavily optimized. In these stages, layout transformations are more likely to cause unexpected effects and destabilize the previously performed optimizations.

In our work we address this stability problem using a novel approach: instead of applying changes that may destabilize a layout and then correcting routing or overlap problems later, we seek and pursue only transformations that preserve the original layout as much as possible. To this end, we define and explore what we call *physically safe* netlist transformations — those that do not create cell overlaps and thus provide immediately-measurable improvements at each step. As indicated by our empirical results, these transformations produce more predictable improvements and no detrimental effects on other circuit parameters. In addition, they do not conflict with any other existing design flow and can be used before or after other transformations, including "unsafe" ones. In the past, safe transformations have been largely neglected because they offered very little improvement [7]. However, we found that the extent of the improvement depends entirely on the pool of transformations available. To find such transformations effectively, we propose a technique, called SafeResynth, which is based on simulation and iterative equivalence checking. By broadening the set of transformations and applying them

in a safe way, we show that we can improve circuit timing with very little risk of destabilizing an existing design flow or hampering timing closure, a common problem of traditional physical synthesis techniques. As an illustrative example, Figure 1 shows two possible transformations which SafeResynth would propose. 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 $g6$, leading to shorter connecting wires and hence better timing delay for $g8$. In Figure 1(b), the long connection from $g6$ to $g8$ is removed and $g8$ is now driven by the new gate as before, however in this example $g6$ still drives gate $g1$. Note that none of the new gates overlap with old gates, and their placement is located on previously unused locations. Empirical results, reported in Table 3, show that our technique can improve delay by 11% on average while route length and via count increase by less than 0.2%. To further investigate a broad range of aspects that may affect the safeness of a physical synthesis technique, we conducted several experiments using SafeResynth with various layout configurations and delay models. Our results show that delay improvement after routing may be considerably different from the one estimated before routing. To alleviate this problem, safe techniques should be used. Alternatively, more accurate delay models can also improve the stability of a given optimization technique.

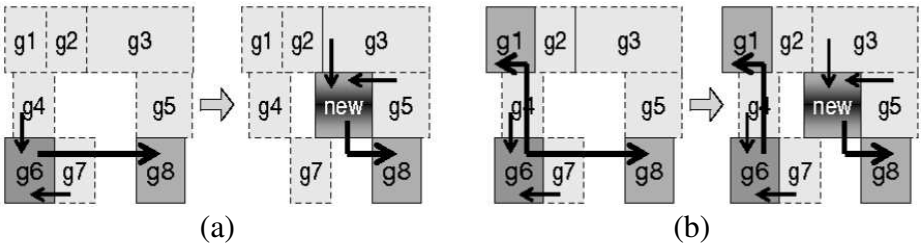


Figure 1: Example transformations for row-based standard-cell layout: (a) resynthesized gate *new* replaces $g6$ to drive $g8$, (b) gate cloning uses resynthesized gate *new* to drive $g8$, while the original driver $g6$ continues to drive $g1$.

Another important application of safe physical synthesis techniques is post-silicon debugging [16]. In post-silicon debugging, once the root cause of a bug has been identified, a Focused Ion Beam (FIB) edit can be performed to implement the fix on the metal layers of the chip. Being able to implement the fix before the next tape-out allows engineers to validate the correctness of the fix and can reduce the overall number of respins. However, FIB can only change metal layers of a chip and cannot create any new transistors. As a result, this technique is often called *metal fix*. Due to this limitation, *spare cells* (cells that are not used in the de-

sign) are usually pre-placed in a layout so that they can be connected through FIB to correct a bug. Existing physical synthesis techniques, however, rarely focus on this application. To this end, we observe that physically safe techniques are especially suitable for exploring metal fix opportunities because they produce minimal perturbation of the layout, and, in particular, SafeResynth can be applied to post-silicon debugging by using these spare cells.

The rest of this paper is organized as follows. In Section 2 we describe the concepts of physical safeness, and then review previous work on physical synthesis. We then propose a new powerful, safe and sound physical synthesis approach in Section 3. Experimental results are reported in Section 4, and Section 5 concludes this paper.

2 Safeness of Physical Synthesis Techniques

Existing techniques for post-placement timing optimization vary in strength and differ in how they affect gate locations [11]. We use the term "physical safeness" to describe their impact on placement. In this section, we first describe safeness in detail. After that, we introduce several physical synthesis techniques and analyze their optimization capabilities and safeness.

2.1 Physical Safeness

The concept of *physical safeness* is used to describe the impact of an optimization technique on the placement of a circuit. Physically safe techniques only allow legal changes to a given placement; therefore, accurate analysis such as timing and congestion can be performed. Such changes are safe because they can be rejected immediately if the layout is not improved. On the other hand, unsafe techniques allow changes that produce a temporarily illegal placement. As a result, their evaluation is delayed, and it is not possible to reliably decide if the change can be accepted or must be rejected until later. Therefore, the average quality of unsafe changes may be worse than that of accepted safe changes. In addition, other physical parameters, such as via count, may be impacted by unsafe transformations, as can be seen from Table 7.

2.2 Physically Safe Techniques

Symmetry-based rewiring is the only timing optimization technique that is physically safe in nature. It exploits symmetries in logic functions, looking for pin reconnections that improve

timing [7]. For example, the inputs to an AND gate can be swapped without changing its logic function. Since only wiring is changed in this technique, the placement is always preserved. An example of symmetry-based rewiring is given in Figure 2(a).

The advantage of physically safe techniques is that the effects of any change are immediately measurable, therefore the change can be accepted or rejected reliably. As a result, delay will not deteriorate after optimization and no timing convergence problem will occur. However, the improvement gained from these techniques is often limited because they cannot aggressively modify the logic or use larger-scale optimizations. For example, in [7] timing improvement measured before routing is typically less than 10%. To this end, our experimental results in Section 4 show that post-routing timing improvements may not match pre-routing results and must be evaluated directly.

2.3 Physically Unsafe Techniques

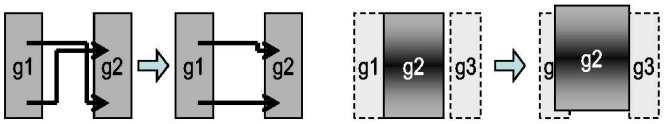
Traditional physical synthesis techniques are physically unsafe because they create cell overlaps and thus prevent immediate evaluation of changes. Although some of these techniques can be applied in a safe way, they may lose their strength. It follows that existing physical synthesis tools usually rely on unsafe techniques, planning to correct potentially illegal changes after the optimization phase is complete. A classification of these techniques and their impact on logic are discussed below. Methods to make these techniques safe are also described.

Gate sizing and buffer insertion are two important techniques that are extensively used, as shown in Figure 2(b) and Figure 2(d). Gate sizing chooses the sizes of the gates carefully so that signal delay in wires can be balanced with gate delay, and gates have enough strength to drive the wires. Buffer insertion adds buffers to drive long wires. The work by Kannan *et al.* [18] is based on these techniques. To make buffer insertion physically safe, one would allow inserting buffers only in overlap-free sites [3]. Similarly, gate sizing can be made physically safe if it is performed only when the resized gate does not overlap any other gate.

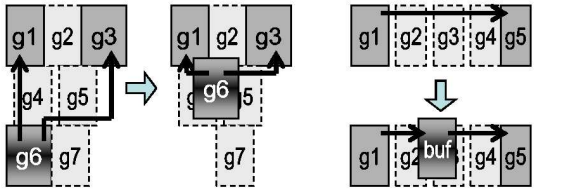
Gate relocation moves gates on critical paths to better locations in order to shorten wire-length and optimize timing. An example of gate relocation is given in Figure 2(c). Ajami *et al.* [1] utilize this technique by performing timing-driven placement with global routing information using the notion of movable Steiner points. They formulate the simultaneous placement and routing problem as a mathematical program. The program is then solved by Han-Powell

method. To make gate relocation physically safe, one must place the relocated gate on unused sites.

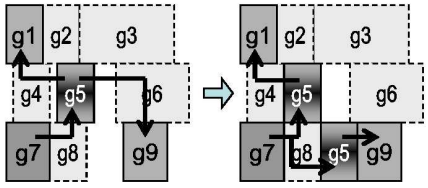
Gate replication is another technique that can improve circuit timing. Consider Figure 2(e) for example, by duplicating $g5$, the delay to $g1$ and $g9$ can be reduced. Hrkic *et al.* [14] propose a placement-coupled approach based on such technique. Given a placed circuit, they first extract replication trees from the critical paths after timing analysis, and then they perform embedding and post-unification to determine the gates that should be duplicated and their locations. Since duplicated gates may overlap with existing gates, at the end of the process, a phase of timing-driven legalization is applied. Although their approach improves timing by 1-36%, it also increases route length by 2-28%. Gate replication can be made physically safe if the duplicated gates are always placed on unoccupied sites.



(a) Symmetry-based rewiring. (b) Gate sizing.



(c) Gate relocation. (d) Buffer insertion.



(e) Gate duplication.

Figure 2: Several distinct physical synthesis techniques. Newly-introduced overlaps are removed by legalizers after the optimization phase has completed.

Traditional rewiring techniques based on addition or removal of redundant wires are not physically safe. The basic idea is to add one or more redundant wires to make a target wire redundant so that it becomes removable. Since gates must be modified to reflect the changes in

wires, cell overlaps may occur. To make such techniques physically safe, changes that create cell overlaps must be rejected. The work by Chang *et al.* utilizes this technique using an ATPG (Automatic Test Pattern Generation) reasoning approach [9].

Traditional restructuring focuses on directing the synthesis process using timing information obtained from a placed or routed circuit. It is more aggressive in that it may change the logic structure as well as the placement. Therefore, ensuring its physical safety is more difficult. For example, new cell locations cannot be evaluated reliably for technology-independent restructuring unless technology mapping is also performed. Moreover, restructuring techniques based on technology-independent (unmapped) netlists are likely to be unsafe because the performed optimizations may distort a given placed circuit. As a result, the effects of the changes are not immediately measurable. In other words, the delay after optimization may be worse than before. Although carefully designed techniques can be used to alleviate this problem [20, 21, 24], it is difficult to be eliminated altogether. The strength and safeness of these techniques are summarized in Table 1.

Techniques	Physical safeness	Optimization range
Symmetry-based rewiring	Safe	Local
SafeResynth (this work)	**Safe**	**Medium**
ATPG-based rewiring, buffer insertion, gate sizing, gate relocation	Unsafe*	Local
Gate replication	Unsafe*	Medium
Restructuring	Unsafe	Large-scale

Table 1: Comparison of physical synthesis techniques in terms of physical safeness and optimization range. *Note: some of these techniques could be made safe but popular implementations use them in an unsafe fashion, allowing gate overlap.

3 A New Powerful and Safe Physical Synthesis Approach

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 validated by equivalence checking [27]. The logical soundness of the optimizations is guaranteed by the equivalence checking validation step. In other words, SafeResynth will not produce netlist modifications that corrupt the circuit's functional correctness. Since our

goal is layout optimization, we can prune some of the opportunities based on their improvement potential before formally verifying them. 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 placement produced is always legal and the circuit improvement can be evaluated immediately. In this work, we discuss the application of SafeResynth for timing optimization, but our solution can also be used to optimize other circuit parameters, such as power or reliability. In addition, it may be applied to post-silicon debugging for timing-violation repair. A preliminary version of the SafeResynth solution discussed in this section was presented in [8].

3.1 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 Table 2).

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.

3.2 SafeResynth Framework

The SafeResynth framework is outlined in Figure 3, and an example is shown in Figure 4. 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 3.4. 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 soundness. 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 then call equivalence checking using the input cone of the new signal when we found certain changes that improve timing. 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.

Figure 3: The SafeResynth framework.

3.3 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

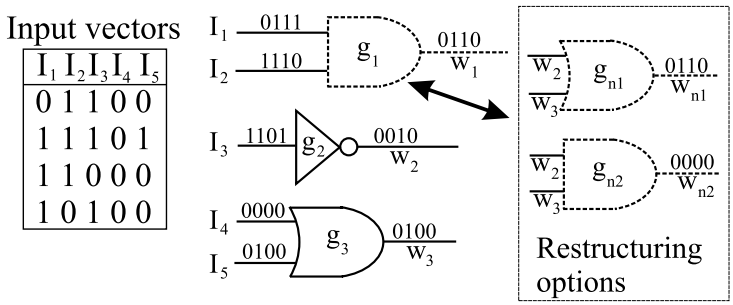


Figure 4: A restructuring example. Input vectors to the circuit are shown on the left. Each wire is annotated with its bit-signature computed by simulation on those test vectors. We seek to compute signal w_1 by a different gate, e.g., in terms of signals w_2 and w_3. Two such restructuring options (new 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.

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. Table 2 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

Table 2: 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;
```

Figure 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;
```

Figure 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.

3.4 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 [27] 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.

Despite our efforts to generate high-quality signatures, ill-behaved signatures still exist and may render our simulation-based techniques ineffective. For example, a wire with an all-1 signature can generate a target wire with an all-1 signature using any wire through an OR gate. The same happens to NOR, AND and NAND gates, but not to XOR and XNOR gates. This problem arises because the gate being tried is controlled by one of its inputs. When this happens, only equivalence checking can verify the correctness of resynthesis involving the ill-behaved wire. Needless to say, most such resynthesis opportunities are invalid, making the time spent to verify them worthless. Therefore in our implementation, we abandon resynthesis opportunities with the number of uncontrolled bits (bits in the signature with non-controlling value of the gate) smaller than 4, making sure that simulation-based techniques have enough chance to prune impossible combinations of wires.

3.5 Analysis of Our Approach

Several aspects of our approach are discussed in this subsection, 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 (see the largest benchmark in Table 3).

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 [13, 22] 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 [6], it is capable of exhaustively finding optimizations that may be long-range. This is important in later stages of physical synthesis because most short-range optimization opportunities may have been exploited. In addition, the exhaustive nature of SafeResynth also allows us to find optimizations that most resynthesis techniques do not consider. For example, the optimization shown in Figure 7 typically cannot be found by synthesis tools because it uses an additional level of logic to generate signal A that is already available; however, this opportunity can be exploited by SafeResynth because it exhaustively tries all possible combinations of wires and gates. Furthermore, since signatures implicitly encode controllability don't-cares, these don't-cares are automatically utilized in our techniques by construction, giving our technique more optimization power to find restructuring opportuni-

ties.

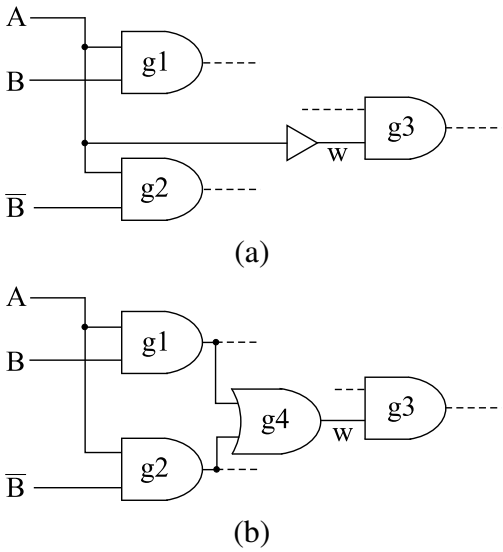


Figure 7: SafeResynth example. The original circuit is shown in (a), and w is on the critical path (a buffer is already inserted to improve its signal propagation). The optimized circuit is shown in (b), where a new gate $g4$ is inserted by SafeResynth. The delay is improved in (b) because wire A has significantly smaller load; therefore, signal propagation to $g1$ and $g2$ becomes faster, resulting in better timing for w even though new gate delays are introduced. This optimization typically cannot be found by traditional resynthesis techniques because an extra level of logic is used.

When we try to resynthesize a wire, we are either trying to remove a gate and drive all the relevant sinks by a new gate or to 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 physically 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 whitespace 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 physically safe. On the other hand, by subsuming and generalizing several existing techniques

they achieve considerable strength in practice.

As far as limitations go, we observed that 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.

4 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 improvements as wire delays become more significant in newer technologies. We use three net delay models. The *star model* from [25] applies Elmore formulas to a star topology with the star point placed at the center of gravity of all pins. Other models use the D2M formulas from [2], and we apply them to Rectilinear Steiner Minimal Trees (RSMTs) generated by the FLUTE package [10][2] or to actual net routes produced by an industry router. We perform our optimizations using STA engines based on the star model and RSMT, and we route the resynthesized layout to measure the final timing based on actual net routes. We observe that STA based on RSMTs provides much more accurate timing estimation than the star model, and the simpler star model has poor correlation with the routed timing. Therefore we report primary results based on RSMTs in this paper, and report results based on the star model only for comparison.

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 [5], the NanoRoute 4.1 router from Cadence, and the FLUTE RSMT package from GSRC Bookshelf [10]. Simulation patterns are generated by the ABC package from UC Berkeley [27], and all transformations are verified by an external equivalence checker based on the MiniSat SAT solver [12].

Our initial testcases are selected from IWLS 2005 benchmarks [28], where the design utilization is 70%, but for experiments in Table 6 we varied the amount of whitespace. These benchmarks belong to the following suites: OpenCores (SPI, DES_AREA, TV80, SYSTEM-

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

CAES, 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 0.18μm library. Our current implementation can only generate two-input NAND, NOR, AND, OR and XOR gates. 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 is performed.

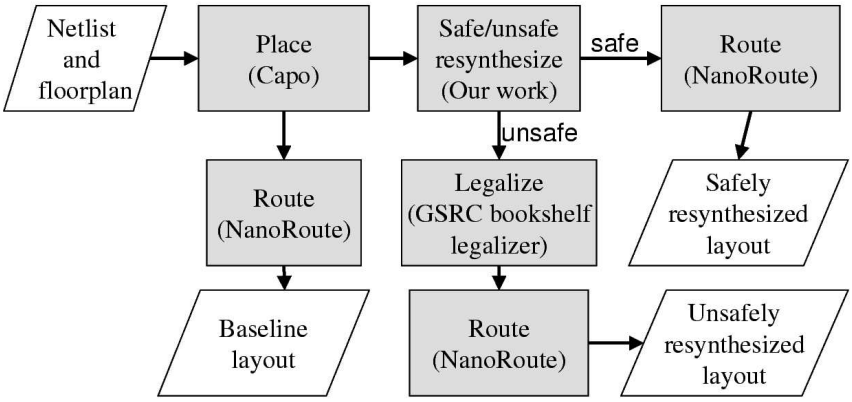


Figure 8: Flow chart of our resynthesis experiments.

The flow of our experiments is summarized in Figure 8. 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 of the benchmarks and our empirical results are summarized in Table 3, where the numbers are averages over three independent runs.

Evaluation of SafeResynth: from the results shown in Table 3, 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 [14] 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

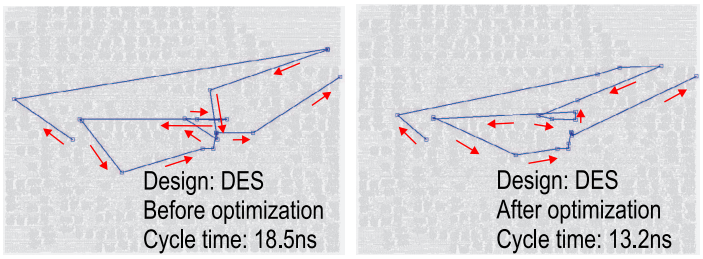
Benchmark	Cell count	Net count	Resynthesized				Runtime (min)
			Est. delay improv.	Routed delay improv.	Route length increase	Additional vias	
SPI	3227	3277	2.89%	2.84%	0.14%	0.14%	1.07
DES_AREA	4881	5122	1.24%	1.28%	0.19%	0.56%	0.66
TV80	7161	7179	12.23%	12.08%	0.25%	0.13%	1.77
SYSTEMCAES	7959	8220	2.94%	2.94%	0.04%	-0.12%	1.02
MEM_CTRL	11440	11560	6.42%	6.54%	0.12%	0.24%	44.71
AC97	11855	11948	2.67%	1.56%	0.04%	-0.14%	0.58
USB	12808	12968	5.21%	3.09%	0.06%	0.15%	1.36
PCI	16816	16990	5.99%	0.00%	0.09%	0.10%	1.68
AES	20795	21055	2.32%	2.25%	0.09%	-0.08%	2.63
WB_CONMAX	29034	30165	61.37%	**61.29%**	0.19%	-0.19%	7.6
Ethernet	46771	46891	85.66%	**85.61%**	0.04%	-0.14%	21.66
DES_PERF	89341	98576	1.98%	1.93%	0.02%	0.01%	5.58
DMA	19118	19809	3.33%	1.03%	0.01%	-0.03%	1.37
b14	8679	8716	3.66%	3.66%	0.04%	-0.03%	4.32
b15	12562	12605	3.71%	3.63%	0.03%	-0.15%	2.22
b17	37117	37167	5.26%	5.22%	0.00%	-0.07%	4.99
b18	92048	92214	17.54%	**17.41%**	-0.04%	-0.07%	23.05
b22	28317	28354	6.58%	6.46%	0.02%	-0.23%	7.75
s35932	7273	7599	9.11%	0.00%	0.05%	0.14%	0.31
s38417	8278	8309	2.38%	0.00%	0.06%	0.14%	0.94
Average			**12.12%**	**10.94%**	**0.07%**	**0.02%**	

Table 3: Improvement achieved by our techniques: relative delay improvements are shown, followed by changes in route lengths and via counts. "Est. delays" were delays estimated by the STA, while "routed delays" were measured using the D2M model from [2]. The last column shows the program runtime.

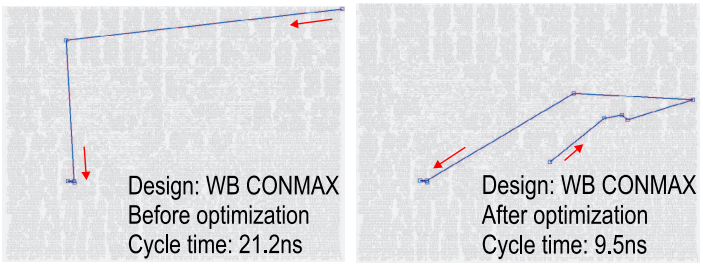
improved by 86% for the Ethernet benchmark, suggesting that our technique is effective when applied by itself. However, our technique still achieves up to 17% delay improvement when applied to already optimized benchmarks (DMA to s38417), indicating that it can augment traditional optimization techniques for further improvement.

The impact of our techniques is illustrated in Figure 9: (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.

Comparison between safe and unsafe optimizations: in order to compare safe and unsafe



(a)



(b)

Figure 9: 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.

optimizations, we apply our resynthesis technique in an unsafe way to compare the results with safe resynthesis. In particular, we allow gate overlap during resynthesis and rely on a legalizer to remove the 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 [29] 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. The results are summarized in Table 4, where the estimated and routed delay improvements are both shown. The route length and via count increase are summarized in Table 5.

The comparison of estimated delay improvement between safe and unsafe resynthesis in Table 4 shows that unsafe resynthesis provides more improvement before legalization because

Benchmark	Estimated delay improvement				Routed delay improvement		
	Safe resynth.	Unsafe resynthesis		Unsafe + safe resynth.	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.
		Before legal.	After legal.				
AC97	2.67%	3.67%	3.44%	3.67%	1.56%	1.31%	2.65%
USB	5.21%	5.29%	5.10%	5.29%	3.09%	6.69%	10.41%
PCI	5.99%	5.37%	4.58%	5.37%	0.00%	-1.90%	0.00%
AES	2.32%	5.06%	4.94%	5.06%	2.25%	3.61%	5.66%
WB_CONMAX	61.37%	61.54%	61.48%	61.54%	61.29%	61.30%	63.14%
Ethernet	85.66%	86.41%	85.89%	86.41%	85.61%	82.07%	86.60%
DES_PERF	1.98%	2.21%	2.12%	2.21%	1.93%	0.49%	2.44%
Average	**23.60%**	**24.22%**	**23.93%**	**24.22%**	**22.25%**	**21.94%**	**24.41%**

Table 4: A comparison of safe resynthesis, unsafe resynthesis, and unsafe followed by safe resynthesis. Relative improvements in delay are shown. Unsafe optimizations allow cell overlaps, and legalization is required to remove the overlaps. Since netlists that are only slightly modified cannot reflect the difference between unsafe and safe resynthesis, we choose seven large benchmarks from the OpenCores suite in this experiment, whose netlists are more significantly altered.

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 5, performing safe instead of unsafe resynthesis avoids the significant increase in via count. The results also suggest that to obtain the greatest improvement, the advantages of both safe and unsafe techniques should be leveraged. As Table 4 shows, this goal can be achieved by applying safe resynthesis after unsafe resynthesis.

The effects of timing-driven placement on optimization results: Capo supports a "boost" mode which optimizes timing during placement [19]. In order to study the effects of timing-driven placement on our technique, we perform safe resynthesis using the same benchmarks shown in Table 4, whose placements are produced by Capo-boost in this experiment. On average, pre-resynthesized routed delay improved by 10.34% due to timing-driven placement, while resynthesis provides an additional 20.08% improvement, resulting in an 30% overall improvement. Compared with the 22.25% improvement shown in Table 4, this result suggests that our optimization is mostly orthogonal to that provided by timing-driven placement, and can improve

Bench-mark	Route length increase			Via count increase		
	Safe	Unsafe resynth.	Unsafe + safe resynth.	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.
AC97	0.04%	0.12%	0.06%	-0.14%	2.60%	2.19%
USB	0.06%	0.00%	0.00%	0.15%	1.56%	1.35%
PCI	0.09%	0.44%	0.48%	0.10%	5.88%	5.47%
AES	0.09%	0.08%	0.11%	-0.08%	4.00%	3.99%
WB_CONMAX	0.19%	-0.16%	0.00%	-0.19%	-0.07%	-0.59%
Ethernet	0.04%	0.00%	0.02%	-0.14%	0.16%	0.14%
DES_PERF	0.02%	-0.12%	-0.11%	0.01%	0.08%	0.08%
Average	**0.08%**	**0.05%**	**0.08%**	**-0.04%**	**2.03%**	**1.80%**

Table 5: A comparison of safe resynthesis, unsafe resynthesis, and unsafe followed by safe resynthesis. Relative increases in route length and via count are shown.

Perc. of white-space	Estimated delay improvement				Routed delay improvement		
	Safe resynth.	Unsafe resynthesis		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%
10%	23.59%	24.12%	23.64%	24.01%	**23.52%**	**23.56%**	23.98%
3%	20.33%	20.78%	20.34%	21.63%	**20.22%**	**20.23%**	21.38%

Table 6: A comparison of delay improvement for layouts with different percentage of white-space.

upon it.

Effects of whitespace on optimization results: in order to study the impact of available whitespace on the success of optimization results, we repeat the same experiments with varying whitespace. The results are summarized in Table 6 and Table 7. We can observe from the tables that delay improvement tends to decrease with the reduction in whitespace because of diminishing flexibility in layouts. However, the difference is small, showing that our SafeResynth technique is effective even when whitespace is limited. In addition, the safeness property is not affected by the amount of whitespace. As seen from Table 7, safe resynthesis essentially preserves via counts, while unsafe resynthesis significantly increases via counts.

Effects of delay model on physical safeness: in order to explore other factors that may affect the stability of physical synthesis techniques, we reran the same experiments using the STA engine based on the star model [25] to study the relation between STA accuracy and physical

Perc. of white-space	Route length increase			Via count increase		
	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.	Safe resynth.	Unsafe resynth.	Unsafe + safe resynth.
30%	0.08%	0.05%	0.08%	**-0.04%**	**2.03%**	1.80%
10%	0.05%	0.09%	0.07%	**-0.01%**	**2.29%**	1.87%
3%	0.04%	0.05%	0.05%	**0.15%**	**1.68%**	1.62%

Table 7: A comparison of route length and via count for layouts with different percentage of whitespace.

Perc. of white-space	Estimated delay improvement			Routed delay improvement		
	Safe resyn-thesis	Unsafe resyn-thesis	Unsafe + safe resynth.	Safe resyn-thesis	Unsafe resyn-thesis	Unsafe + safe resynth.
30%	34.20%	34.51%	36.53%	19.73%	20.58%	20.97%
10%	26.27%	25.62%	28.11%	15.62%	15.12%	17.78%
3%	21.64%	21.57%	24.10%	16.45%	16.10%	19.75%

Table 8: A comparison of delay improvement using STA based on the star model.

stability. The results are summarized in Table 8, which should be compared with Table 6 where timing analysis based on RSMT topology is used. These results show that the star model often overestimates delay, and therefore may also overestimate delay improvement. As can be seen from Table 8, the routed delay improvement is much smaller than the estimated improvement. In addition, the comparison with Table 6 shows that the star model provides smaller routed delay improvement, suggesting that timing analysis without route topology is inaccurate. The results also show that inaccurate timing analysis may make unsafe techniques unsafer. As can be observed from Table 8, safe resynthesis performs better than unsafe resynthesis at 3% and 10% whitespace, suggesting that legalization worsens the final timing more significantly. In contrast, Table 6 shows similar performance between unsafe and safe resynthesis. To further investigate this phenomenon, we provide detailed results after resynthesizing seven large OpenCores benchmarks at 3% whitespace in Table 9. The results show that when the star model is used, the optimized delay may be worse than that of the unoptimized layout. However, we do not observe such a phenomenon when RSMT topology is used. This observation suggests that inaccurate timing analysis worsens physical stability. On the other hand, Table 9 also shows that safe resynthesis is less likely to worsen the final timing when the star model is used, sug-

gesting that physical stability can be improved by applying the optimizations in a safe way. This result indicates that the stability of existing physical synthesis techniques may be improved by performing safe instead of unsafe layout modifications.

Bench-mark	Routed delay improvement					
	Star model			RSMT topology		
	Safe resyn-thesis	Unsafe resyn-. thesis	Unsafe + safe resynth.	Safe resyn-thesis	Unsafe resyn-thesis	Unsafe + safe resynth.
AC97	0.00%	**-1.62%**	**-1.04%**	2.43%	3.14%	4.58%
USB	**-4.04%**	**-5.56%**	3.15%	6.81%	6.47%	8.19%
PCI	2.02%	1.44%	1.46%	3.80%	4.25%	5.29%
AES	1.58%	3.82%	4.33%	2.49%	2.26%	3.39%
WB_CONMAX	63.90%	63.82%	63.97%	60.47%	60.35%	60.32%
Ethernet	51.81%	51.89%	60.44%	63.67%	62.62%	65.64%
DES_PERF	**-0.14%**	**-1.11%**	5.94%	1.83%	2.49%	2.22%

Table 9: Routed delay improvement of OpenCores benchmarks at 3% whitespace resynthesized using STAs based on the star model and the RSMT topology.

5 Conclusions

In this paper we proposed and evaluated the concept of physical safeness to analyze circuit transformations in physical synthesis: physically safe techniques only modify the circuit in such a way that the effect is immediately and reliably measurable. Safe techniques are preferable in later physical synthesis stages because the optimizations are more stable, more reliable and more predictable. In addition, such techniques are especially suitable for post-silicon debugging because cell locations are preserved, allowing changes to be implemented via metal fix. On the other hand, most safe techniques known before our work are limited in their optimization power because of the small number of transformations allowed.

To overcome the limitations of traditional physically safe techniques, we proposed a new resynthesis algorithm called SafeResynth that is safe and sound. It is physically safe because no cell overlap is created during resynthesis. It is logically sound in that equivalence checking is used during resynthesis, thus the functional correctness of the resynthesized netlist is guaranteed. SafeResynth utilizes simulation to generate a signature for each wire, and the wires on the critical path are resynthesized using new gates with their inputs selected from compatible wires.

On-line equivalence checking is then applied to validate the proposed logic transformations. Since we allow the insertion of additional gates only when unused space is available, original gate locations are preserved. In addition, the ability to search for candidate wires globally 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. Furthermore, it is orthogonal to timing-driven placement and can provide additional improvements. Our technique can be applied to practically any design flow without hampering its timing closure. Finally, our comparison between safe and unsafe optimizations highlights the importance of developing more powerful physically safe techniques, or methods to apply intrinsically unsafe transformations in a safe way with minimal loss in optimization potential.

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] C. J. Alpert, G. Gandham, M. Hrkic, J. Hu, S. T. Quay, C. N. Sze, "Porosity-aware Buffered Steiner Tree Construction", *IEEE Trans. on CAD*, Apr. 2004, pp. 517-526.

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

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

[6] 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.

[7] C.-W. Chang, M.-F. Hsiao, B. Hu, K. Wang, M. Marek-Sadowska, C.-K. Cheng, S.-J. Chen, "Fast Postplacement Optimization Using Functional Symmetries", *IEEE Trans. on CAD*, Jan. 2004, pp. 102-118.

[8] K.-H. Chang, I. L. Markov and V. Bertacco, "Safe Delay Optimization for Physical Synthesis", *ASPDAC*, 2007, pp. 628-633.

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

[10] 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`

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

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

[13] 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.

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

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

[16] D. Josephson, "The Good, the Bad, and the Ugly of Silicon Debug", *DAC'06*, pp. 3-6.

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

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

[19] A. B. Kahng, I. L. Markov and S. Reda, "Boosting: Min-Cut Placement with Improved Signal Delay", *DATE'04*, pp. 1098-1103.

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

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

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

[23] 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.

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

[25] B. M. Riess and G. G. Ettelt, "Speed: Fast and Efficient Timing Driven Placement", *ISCAS'95*, pp. 377-380.

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

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

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

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