CONTANGO: Integrated Optimization of SoC Clock Networks

Dongjin Lee, Igor L. Markov
University of Michigan, 2260 Hayward St., Ann Arbor, MI 48109
{ejdjsy, imarkov}@eecs.umich.edu

***Abstract*—On-chip clock networks are remarkable in their impact on the performance and power of synchronous circuits, in their susceptibility to adverse effects of semiconductor technology scaling, as well as in their strong potential for improvement through better CAD algorithms and tools.**

Our work offers new algorithms and a methodology for SPICE-accurate optimization of clock networks, coordinated to satisfy slew constraints and achieve best trade-offs between skew, insertion delay, power, as well as tolerance to variations. Our implementation, called Contango, is evaluated on 45nm benchmarks from IBM Research and Texas Instruments with up to 50K sinks.

I. INTRODUCTION

Accurate distribution of clock signals is a major limiting factor for high-performance integrated circuits when unintended clock skew narrows down the useful portion of the clock cycle. Semiconductor scaling in the 1990s made clock optimization more challenging. While transistors continued scaling, interconnect lagged in performance [6]. This phenomenon boosted demands for repeaters in clock networks, raised their power profile, and complicated their synthesis.

Clock networks were among the first circuits to suffer the impact of process, voltage and temperature variations. Systematic variations can affect paths to different sinks in different ways, making effective skew higher than nominal skew. Intra-die variations may be stronger on some paths than on others, which would further increase effective skew. These challenges have motivated research at the device, circuit and algorithm levels [9]. In general, smaller sink latencies and shorter tree paths decrease exposure to variations.

Our work focuses on clock-network synthesis for ASICs and SoCs, where clock frequencies are not as aggressive as in high-performance CPUs, but power is limited, especially for portable applications. In this context, tree topologies remain the most popular choice, potentially with further tuning and enhancements. The SoC context introduces another twist — layout obstacles. A typical SoC includes numerous predesigned blocks (CPUs, RAMs, DSPs, etc) and datapaths. While it may be possible to route wires over such obstacles, buffer insertion is typically not allowed. One can fathom the difficulty of such optimization through comparison to signal-net routing, where obstacle-avoiding Steiner trees currently remain an active area of research [10].

We make the following contributions.

- Notions of *slow-down & speed-up slack* for clock trees
- Tree optimizations driven by accurate delay models
- A simple and robust technique for obstacle avoidance in clock trees subject to slew constraints
- A provably-good sink-polarity correction algorithm
- A methodology for clock-tree optimizations that outperforms the best results at the ISPD‘09 contest *on every benchmark* by $2.15 - 3.99$ times, while reducing skew to $2.2 - 4.6ps$. On newer Texas Instruments benchmarks with up to 50K sinks, skew remains $< 11ps$.

Further optimization is possible by selecting best parameters for each benchmark, at the cost of increased runtime. But skew$< 20ps$ is considered negligible in industrial practice.

II. BACKGROUND AND PRIOR WORK

DME algorithms. Traditionally, clock trees have been constructed with respect to simple delay models — geometric pathlength or Elmore delay. In this context, the results in [1], [2], [4], [14] show how to build zero-skew trees (ZSTs) with minimal wirelength, improving upon H-trees and fishbones.

The Deferred Merge Embedding (DME) algorithm, using the concept of *merging segment* [1], [4] for constructing zero-skew tree, was extended to the bounded-skew tree (BST) problem. BST/DME algorithms [3], [7] generalize merging segments to merging regions. When BST/DME algorithms were introduced in the early 1990s, many chip designs included one large central buffer to drive clock signals through the entire chip. However, SPICE simulations indicate that traditional clock trees wouldn't satisfy slew constraints in modern designs because the maximal length of unbuffered interconnect decreased significantly due to technology scaling [6].

Obstacle-avoiding clock trees. The concept of merging regions in BST/DME was extended to obstacle-avoiding trees in [8], where (*i*) obstacles were assumed rectangular, (*ii*) no routing over obstacles was allowed, and (*iii*) buffering was not considered. The authors observed that obstacle processing slowed down their BST/DME algorithm and hinted at more sophisticated geometric data structures. In contrast to [8], the ISPD‘09 contest allowed *routing* but not *buffering* over obstacles, with modern SoCs in mind.

Fast buffer insertion. L. van Ginneken introduced an algorithm for buffering RC-trees [5], which minimizes Elmore delay and runs in $O(n^2)$ time, given n possible buffer locations. While not intended for clock trees, it minimizes worst delay rather than skew. The $O(n \log n)$-time variant of van Ginneken's algorithm proposed in [12] is more approriate for large trees. Futhermore, it spares buffers on fast paths and results in low skew if the initial tree was balanced.

The ISPD‘09 clock-network synthesis contest was organized by IBM Austin Research Laboratory and based on a 45nm technology. Sink latencies and clock skew were evaluated by SPICE. The main objective was the difference between the least sink latency @1.2V (supply) and the greatest sink latency @1V (supply). This *Clock Latency Range* (CLR) metric was intended to capture the impact of variations, but nominal skew was also recorded. The 10%-90% slew rate and total power were strictly limited.

III. Problem analysis

The design of a clock network offers a large amount of freedom in topology selection, spacing and sizing of inverters, as well as the sizing of individual wires. Traditionally, network topology is decided first. Trees offer unparalleled flexibility in optimization because latency from the root to each sink can be tuned individually, while large groups of sinks can be tuned by altering nodes and edges high up in the tree.

A. Optimization objectives & delay modeling

Accurate clock network design is complicated by the fact that the optimization objectives are not available in closed form and take significant CPU resources to evaluate. Skew optimization requires much higher accuracy than popular Elmore-like delay models. For example, a 5ps error represents only 1% of 500ps sink latency, but 50% of 10ps skew. Closed-form models do not capture resistive shielding in long wires, do not propagate slew with sufficient accuracy, and do not account for slew's impact on delay well. Newer, more sophisticated models are laborious to implement and only available in modern commercial tools. Our strategy is to use simple analytical models at the first steps of the proposed flow — (1) to construct zero-skew clock trees and (2) to perform initial fast buffer insertion, — but drive further optimizations by SPICE runs, Arnoldi approximation, or any other available timing analysis tool/model.

B. Nominal skew optimization

An initial buffered clock tree is constructed early in the design flow. Assuming no slew violations, the latency of each sink is known from SPICE simulations, at which point minimal and maximal latencies (T_{max} and T_{min}) can be found.[1] Since absolute sink latencies are not as important as skew ($T_{max} - T_{min}$), skew can be improved by either decreasing T_{max} (speeding up the slowest sinks) or increasing T_{min} (slowing down the fastest sinks).

Definition 1: Consider a clock tree and its sink s. The *slow-down slack* $Slack_s^{slow}$ (*speed-up slack* $Slack_s^{Fast}$) of s is the amount in *ps* by which the sink latency can be unilaterally increased (decreased) without increasing clock skew. In other words, $Slack_s^{slow} = T_{max} - T_s$ and $Slack_s^{Fast} = T_s - T_{min}$.

Slow sinks often cluster together, and so do fast sinks. Hence, clock skew can be improved by modifying a few nodes or edges high in the tree. To find desired delay change, we propagate slack information up the tree as follows.

Let $Sinks_e$ be the set of downstream sinks for edge e.

Definition 2: Consider a clock tree and its edge e. The *slow-down slack* $Slack_e^{slow}$ (*speed-up slack* $Slack_e^{Fast}$) of e is the amount in *ps* by which the edge delay can be unilaterally increased (decreased) without increasing clock skew.

Lemma 1: For any edge e in the tree

- $Slack_e^{slow} = \min_{s \in Sinks_e} Slack_s^{slow}$
- $Slack_e^{Fast} = \min_{s \in Sinks_e} Slack_s^{Fast}$

Given slacks on n sinks, all edge slacks can be computed in $O(n)$ time.

[1]Separately for rising and falling transitions, for each PVT corner.

Lemma 2: For any edge e and its parent in the tree, $Slack_e^{slow} \geq Slack_{parent(e)}^{slow}$ and $Slack_e^{Fast} \geq Slack_{parent(e)}^{Fast}$.

The flexibility of a tree edge is limited by each downstream sink. Therefore, for edges close to the root we often have $Slack_e^{slow} = Slack_e^{Fast} = 0$. It is important to note that the validity of slacks-related calculations does not depend on the use of specific delay models or SPICE simulations. When visualizing clock trees, we color their edges with a red-green gradient, indicating low slack with red and high slack with green, as shown in Figure 3.

Lemma 2 suggests that, instead of changing the delay of an edge, one can change the delay of its downstream edges by an equal amount, as long as only one delay change is applied on each root-to-sink path. When choosing between tree edges on the same path, we prefer (at early stages of optimization) to tune edges as high in the tree as possible, so as to minimize (i) the amount of change, (ii) the risk of introducing slew violations and (iii) power overhead. However, in a highly optimized tree, we tune bottom-level edges where we can better predict the impact on skew. The preference for high-level tree edges can be formalized as follows.

Proposition 1: For each edge e in the tree, define $\Delta_e^{slow} = Slack_e^{slow} - Slack_{parent(e)}^{slow}$. If every edge is slowed down exactly by Δ_e^{slow}, the tree's skew will become zero, and both slow-down and speed-up slacks will become zero.

Naturally $\Delta_e^{fast} = Slack_e^{fast} - Slack_{parent(e)}^{fast}$, and a mirror statement holds. For a tree edge e, it is possible that $\Delta_e^{fast} > 0$ and $\Delta_e^{slow} > 0$, facilitating conflicting optimizations. If optimizations are not coordinated well, some edges may be sped up and some slowed down, while the overall skew is unchanged. To avoid such conflicts, one can perform rounds of speed-up and rounds of slow-down, separated by SPICE-based analysis and slack update. In practice, it is usually much easier to slow down an edge (e.g., by wire snaking) than to speed it up. If any speed-up is possible, e.g., by using stronger buffers, it is performed first. Rounds of speed-up and slow-down are more conveniently performed top-down, so that when an edge cannot be tuned by the desired amount, the remainder is passed to its downstream edges.

We found that after nominal skew is sufficiently optimized, both rising and falling transitions can individually limit speed-up and slow-down slacks. We handle the two transitions separately and define edge slacks as the smaller of rise-slack and fall-slack. Furthermore, speed-up and slow-down slacks can be computed for each process corner given (two in the ISPD'09 contest). In order to improve the multicorner CLR objective, a tree edge can be sped up conservatively by the minimum of its speed-up slacks, and can be slowed down by the minimum of its slow-down slacks.

C. Coordinating multiple optimizations

We found that different clock-tree optimizations exhibit different strength/range and different accuracy (see Table III). Our strategy in coordinating clock-tree optimizations is to start with optimizations that offer the greatest range, and then transition to optimizations with greater accuracy.

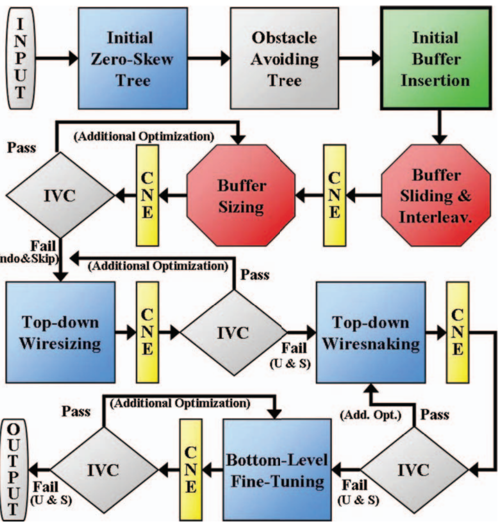

Fig. 1. Key steps of the Contango methodology. Blue boxes represent *skew reduction* techniques, red octilinear shapes show *CLR reductions*, and the green box with thick border reduces both objectives. An Improvement- & Violation-Checking (IVC) step follows each Clock-Network Evaluation (CNE) using circuit simulation tools, e.g., SPICE. "Fail" indicates no improvement or having slew violations, leading to a transition to the next optimization.

IV. Proposed SoC clock-synthesis methodology

Our proposed clock-network synthesis methodology and its major algorithmic steps are shown in Figure 1. Contango first builds an initial tree using a ZST/DME algorithm [3] and alters it to avoid obstacles. It then uses an $O(n \log n)$-time variant of van Ginneken's buffer insertion algorithm [12] to ensure small insertion delay and satisfy slew constraints. A series of novel clock-tree optimizations are applied next.

A. Obstacle-avoiding clock trees

As we pointed out in Section II, obstacle-avoiding clock trees can be built by repairing obstacle violations in ZSTs. This approach is attractive when large obstacles abut the chip's periphery because ZSTs naturally avoid areas without clock sinks. This approach is also attractive when obstacles are small or thin enough that a buffer inserted immediately before the obstacle can drive the wire over the obstacle, so that no rerouting is necessary. A third convenient case occurs when a wire can be rerouted around the obstacle without an increase in length. Most obstacles are rectangular in shape, but such rectangles may abut, creating rectilinear-shaped obstacles. When two obstacles abut, we cannot place a buffer between them, and therefore handle them as one compound obstacle. Contango detours wires using the following algorithm, illustrated in Figure 2 for a composite obstacles.

Step 1. Identify all wires that intersect obstacles. For each point-to-point connection, perform *shortest-path maze routing* around the obstacles. For subtrees that cross an obstacle, find L-shaped segments that link points inside and outside the obstacle. For each L-shape, choose one of the two possible configurations that minimizes overlap with the obstacle.

Step 2. When a wire crosses an obstacle, Contango captures an entire subtree enclosed by the obstacle (see Figure 2). The total capacitance of the subtree is then measured and compared to the capacitance that can be driven by a single buffer without risking slew violations (*slew-free capacitance*). Sub-trees that can be driven by one buffer do not require detours.

Step 3. For obstacles crossed by a subtree that cannot be safely driven by a single buffer, Contango establishes a detour along the contour of the obstacle. This is accomplished by first considering the entire contour as a detour, and then removing one segment between tree sinks adjacent along the contour, so as to ensure that the clock network remains a tree. If we were to minimize total capacitance, we would remove the longest segment of the contour between two adjacent tree sinks. However, *we minimize the longest detoured source-to-sink path*, and therefore *remove the segment furthest from the tree source* (counting distances along the contour). In other words, we first find the sink most distant from the source along the contour, and include in the detour the entire shortest path to the source. The other segment incident to the sink is removed, but the shortest path from its other end to the source is included (see Figure 2).

Detours may significantly increase skew, but electrical correction can compensate for that.

B. Composite inverter/buffer analysis

Most technology libraries support dedicated clock buffers or inverters that are larger and more reliable than those for signal nets. Parallel composition of buffers increases driver strength, helping with slew constraints and improving robustness to variations. Yet, buffer sizes must be moderated to satisfy total power limits. For a given buffer library, we consider many possible composite buffers. Using dynamic programming, we select several non-dominated configurations that can be further evaluated during buffer insertion. Algorithmic details are omited here because the ISPD'09 contest used only two inverter types — *large* and *small*. Table I shows that eight parallel *small* inverters exhibit smaller output resistance than one *large* inverter, and smaller input/output capacitance. Hence Contango used 8× *small* inverters instead of *large* inverters, in batches of 16×, 24×, etc. This benchmark-independent optimization, along with buffer sizing, plays an important role in our methodology.

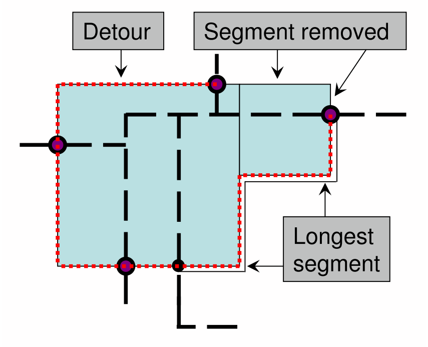

Fig. 2. An illustration of our detouring algorithm. Small solid circle indicates the source of detour, larger circles indicate sinks. The detour is shown with red dotted lines.

INVERTER TYPE	INPUT Cap., fF	OUTPUT Cap., fF	OUTPUT Res., Ω
1X Large	35	80	61.2
1X Small	4.2	6.1	440
2X Small	8.4	12.2	220
4X Small	16.8	24.4	110
8X Small	**33.6**	**48.8**	**55**

TABLE I
INVERTER ANALYSIS FOR ISPD'09 CNS BENCHMARKS.

C. Initial inverter insertion with sizing

Given a clock tree with buffers, it is easy to increase the latency of a given sink, but it is difficult to speed up a sink. Therefore, our strategy is to first make sinks as fast as possible, and then reduce skew with wiresnaking and wiresizing. When buffers are inserted into an Elmore-balanced tree, source-to-sink paths contain practically the same numbers of buffers.

We adapted the $O(n \log n)$-time variant of van Ginneken's algorithm from [12]. Due to its speed, it can be launched with different inverter configurations, effectively performing simultaneous optimization across multiple parameters. Our experiments indicate that driver strength is a major factor in moderating the impact of supply-voltage variations. Therefore Contango performs fast buffer insertion with different composite buffers until it finds the best-performing solution with strongest composite buffers within the 90% of the power limit. We reserve $\gamma = 10\%$ of power budget to facilitate more accurate optimizations.

D. Sink-polarity correction

The $O(n \log n)$ variant of van Ginneken's algorithm [12] used in our work assumes that all available clock buffers preserve polarity. However, when polarity-changing inverters are used, as in the ISPD'09 contest, it will typically produce trees with incorrect sink polarity (inverted sinks).

While the algorithm can be extended to account for sink polarity, we found this unnecessary. Even a simple patch — placing additional inverters at each of $n_\times$ inverted sinks — works reasonably well, because the skew introduced by new inverters can be fixed by downstream optimizations. This technique inserts inverters at half the sinks ($n/2$) on average. To reduce the added capacitance in cases when $n_\times > n/2$, Contango inserts one inverter at the top of the tree, leaving only $n_\sharp = (n - n_\times) < n/2$ sinks with wrong polarity. The average number of inserted inverters would now be $(n+2)/4$. Instead, Contango traverses the tree bottom-up and marks each node (i) whose all sinks have equal polarity, but (ii) whose parent does not satisfy (i). An inverter is inserted at each marked node with downstream sinks of incorrect polarity. As a result, the number of added inverters is significantly reduced, as shown in Table II.

Proposition 2: The above algorithm runs in $O(n)$ time, corrects all inverted sinks and minimizes the number of added inverters, subject to ≤ 1 inverter on every root-to-sink path.

	f11	f12	f21	f22	f31	f32	fnb1
Inverted sinks	77	71	46	57	140	47	153
Added inverters	9	7	8	9	16	13	2

TABLE II
INVERTED SINKS IN ISPD'09 BENCHMARKS (AFTER BUFFER INSERTION) VS. POLARITY-CORRECTING INVERTERS.

Algorithm 1 IterativeWireSizing

```
$T_{ws}$ = TwsEstimation();
repeat
    SaveSolution(); ComputeWireSlacks();
    $Q = \{\text{root}\}$; $RSlack = \{0\}$; $i = 0$;
    while $i < \text{size}(Q)$ do
        if ($Slack[Q_i] - RSlack_i > T_{ws}$) then
            DownSize($Wire[Q_i]$); $RSlack_i += T_{ws}$;
        end if
        for $j = 1$ to Size($Child[Q_i]$) do
            $Q$.push($Child[Q_i][j]$); $RSlack$.push($RSlack_i$);
        end for
        $++i$;
    end while
    SpiceSimulation();
until (no improvement || slew violation)
```

E. Iterative top-down wiresizing

After the initial SPICE run, Contango computes slow-down slacks at every edge as described in Section III, and the Δ_e^{slow} parameters. This suggests the amount by which a given tree edge can be slowed down before skew would be negatively affected. Since fast sinks often cluster together, skew can be lowered by slowing down either many bottom-level wires or few wires higher in the tree. Our top-down algorithm pursues the latter, seeking to minimize tree modifications.

We build an *ad hoc* linear model based on the impact of downsizing a unit-length wire segment. Contango chooses several independent wire segments in the middle of the tree and downsizes them to observe the impact on latencies of downstream sinks. This requires a single SPICE run and produces a single parameter T_{ws} — maximum latency increase (over all sinks). When downsizing a wire, we multiply T_{ws} by its length to estimate the impact on downstream sink latencies. To understand why this linear model works well in practice, assume that delay is modeled by a sum of RC terms. When a short wire segment is sized, the affected Rs and Cs do not appear in the same term, thus, the impact on delay is linear.

F. Iterative top-down wiresnaking

Wiresizing can reduce large skew by applying small changes, which is appropriate after the initial tree construction. An experienced clock-network designer suggested to us that a small amount of wire-snaking is often used to improve clock skew, as long as added capacitance does not significantly affect power. Therefore, we developed an accurate top-down wiresnaking process, to be invoked *after* top-down wiresizing. This step uses the same slow-down slack computation we described earlier. A SPICE simulation is performed (other accurate delay model can be used) to measure T_{wn}, the worst-case delay of wiresnaking with unit length l_{wn}. l_{wn} affects the accuracy of the wiresnaking algorithm; smaller l_{wn} offers greater accuracy but typically leads to more SPICE runs since skew reduction in each round of top-down wiresnaking is smaller. l_{wn} was set based on empirical data.

	ISPD09F11		ISPD09F12		ISPD09F21		ISPD09F22		ISPD09F31		ISPD09F32		ISPD09FNB1	
	CLR	Skew	CLR	Skew	CLR	Skew	CLR	Skew	CLR	Skew	CLR	Skew	CLR	Skew
INITIAL	56.18	30.58	75.81	48.96	89.29	59.17	52.01	31.55	151.8	116.5	121.6	88.19	31.86	21.15
TBSZ	55.61	46.78	80.03	66.24	89.49	76.31	43.16	33.65	140.3	129.2	110.7	98.27	31.54	21.13
TWSZ	23.38	15.07	19.70	8.127	26.00	12.25	16.35	6.933	43.08	32.21	27.23	14.84	30.75	20.44
TWSN	13.75	2.929	16.21	3.384	17.60	2.826	12.58	**1.99**	12.81	3.91	17.92	4.594	13.94	**3.149**
BWSN	**13.36**	**2.867**	**15.27**	**2.611**	**17.40**	**2.738**	**12.36**	2.227	**12.81**	**3.91**	**17.92**	**4.594**	**13.40**	3.5

TABLE III
PROGRESS ACHIEVED BY INDIVIDUAL STEPS OF CONTANGO ON ISPD'09 BENCHMARKS: THE FIRST LETTER IN EACH ACRONYM INDICATES TOP-DOWN (T) OR BOTTOM-LEVEL (B) OPTIMIZATION, SECOND LETTER DIFFERENTIATES WIRES (W) FROM BUFFERS (B), WHILE "SZ" STANDS FOR "SIZING" AND "SN" STANDS FOR "SNAKING". GRAY HIGHLIGHTS INDICATE WHETHER SKEW OR CLR WAS THE PRIMARY OPTIMIZATION OBJECTIVE.

G. *Bottom-level fine-tuning*

After two top-down skew reduction phases, skew becomes small enough to perform bottom level optimizations. Bottom-level wiresizing and wiresnaking optimize the wires directly connected to sinks. Contango performs SPICE-driven bottom-level wiresizing and wiresnaking until the results stop improving. Typically the gain of bottom-level tuning is under $2ps$, but can be a significant fraction of remaining skew.

We found that when skew is under 5ps, the corner sinks of rising transition and falling transition are often different. This *rise-fall divergence* makes further improvements to the clock-tree very difficult. Indeed, reducing *rising skew* by slowing down a *fast sink for rising transition* may increase *falling skew* due to excessive slowdown of a *slow sink for falling transition*. The average skew after bottom-level tuning is $3.21ps$ on ISPD'09 CNS contest benchmarks.

H. *Buffer sliding and interleaving*

Optimization techniques covered so far focus on skew, possibly in combination with the CLR objective. We now discuss targeted improvement of robustness to variations in device performance. Extensive experiments suggest that the impact of variations on skew is best reduced by (i) decreasing sink latency (insertion delay), and (ii) using the strongest possible buffers. Each measure must be applied to achieve balance over all source-to-sink paths, or else skew will increase. Recall from Section IV-C that Contango minimizes total wirelength and initially minimizes insertion delay by using strongest possible buffers, subject to power limit and a 10% reserve for downstream optimizations.

Sizing up a single inverter increases its input pin capacitance and can lead to slew violations. To prevent such violations, it is often possible to slide the inverter up the tree to reduce upstream wire capacitance and interleave an inverter when two inverters move too far apart after sliding. The increase in downstream wire capacitance is balanced with the increase in the inverter's driving strength. Sizing a single inverter may increase the skew and require further correction. Therefore, we focused on the top-most levels of the tree, whose impact on skew is relatively small. Given a clock source at the chip boundary, DME algorithms generate a long wire leading to the center of the chip, and the tree branches out from the center. This long wire — the *tree trunk* — is later populated with a chain of inverters, which can be upsized without significant impact on skew because this equally affects all sinks. However, since roughly 1/3 to 1/2 of sink latency is due to the tree trunk, it accounts for a large fraction of variational impact on latency.

I. *Iterative buffer sizing*

After sliding and interleaving top-level buffers, we invoke iterative buffer sizing. First, this algorithm sizes up buffers in the tree trunk. At the i-th iteration of buffer sizing, Contango sizes up the composite inverters by at most $p_i = 100/(i+3)\%$. The iterations continue until results improve without slew violation. Buffer sizing in tree branches incurs a greater capacitance penalty. To compensate, Contango borrows capacitance by downsizing bottom-level buffers.

However, sizing up buffers after the trunk often makes the tree unbalanced in terms of skew and results in more load for the skew optimization algorithms. For better performance of skew optimizations, typically 4 or 5 levels after the first branch are sized up by capacitance borrowing buffer sizing algorithm. Table III shows the improvement of CLR by each optimization algorithms. Buffer sizing increases skew, but subsequent skew optimizations bring it back down.

V. EMPIRICAL VALIDATION

ISPD'09 benchmarks include seven 45nm chips up to $17mm \times 17mm$ in size, with up to 330 selected clock sinks [13]. Table IV compares results of our software Contango to the top three teams of the ISPD'09 clock-network synthesis contest. On average, Contango reduces CLR by **2.15**$\times$, **3.99**$\times$ and **2.35**$\times$ versus results by NTU, NCTU and U. of Michigan respectively. All results are within the capacitance limits.

Contango runs faster than NTU and NCTU on most benchmarks (we measured runtimes on a 2.4GHz Intel QuadCore CPU, similar to CPUs used at the contest). A detailed breakdown of Contango optimizations is given in Table III. A clock tree produced by Contango is illustrated in Figure 3.

Scalability studies. The ISPD'09 contest was limited to unrealistically small numbers of sinks due to limitations of the open-source ngSPICE software it relied upon. To evaluate the scalability of our optimizations, we replaced ngSPICE with HSPICE. Working with a recent Texas Instruments chip sized $4.2mm \times 3.0mm$, we identified locations of 135K sinks and randomly sampled them to create a family of benchmarks. For this experiment, our algorithm used groups of large inverters instead of groups of 8 parallel small inverters, improving runtime eightfold at the cost of increasing CLR and skew by 1-2ps and increasing capacitance by 15%. It produced highly-optimized clock trees with up to 50K sinks. Table V shows that total capacitance scales linearly with the number of sinks, and skew remains in single ps. The number of HSPICE runs grows very slowly, but HSPICE remains the runtime bottleneck.

Benchmark	CONTANGO(THIS WORK) 9/10/2009			NTU 3/30/2009			NCTU 3/30/2009			U. OF MICHIGAN 3/30/2009		
	CLR	Cap.	🕒	CLR	Cap.	🕒	CLR	Cap.	🕒	CLR	Cap.	🕒
ispd09f11	**13.36**	99.61	6488	26.71	85.53	14764	**22.31**	89.90	23358	32.29	73.86	3892
ispd09f12	**15.27**	99.99	6564	25.73	84.72	13934	**22.18**	87.86	14992	32.17	73.45	3944
ispd09f21	**17.40**	96.74	6673	30.54	80.79	14978	**19.61**	86.65	26420	34.31	74.30	4587
ispd09f22	**12.36**	97.43	3618	24.51	81.82	7189	**16.38**	85.01	9432	30.45	70.01	2005
ispd09f31	**12.81**	98.29	21379	**45.07**	73.49	40088	212.0	92.38	1.29	51.34	81.53	17333
ispd09f32	**17.92**	99.24	12895	**36.90**	80.14	3566	fail	-	-	40.32	77.39	10599
ispd09fnb1	**13.40**	78.38	778	fail	-	-	fail	-	-	**19.84**	63.10	477
Average	**14.65**	**95.66**	**8342**	**31.57**	**81.08**	**15753**	**58.49**	**88.36**	**14841**	**34.39**	**73.38**	**6120**
Relative	1.0	1.0	1.0	2.15	0.85	1.89	3.99	0.92	1.78	2.35	0.77	0.73

TABLE IV

RESULTS ON THE ISPD'09 CONTEST BENCHMARK SUITE. CLR IS REPORTED IN *ps*, CAPACITANCE IN % OF THE LIMIT SPECIFIED IN BENCHMARKS, AND CPU TIME IN *s*. BEST RESULTS FROM THE ISPD'09 CONTEST AND BEST RESULTS OVERALL ARE SHOWN IN BOLD. SKEW AND CLR BELOW 20PS ARE CONSIDERED NEGLIGIBLE IN MODERN INDUSTRY PRACTICE. RUNTIME IS DOMINATED BY SPICE RUNS. IT WAS NOT USED FOR SCORING AT THE ISPD'09 CONTEST AND CAN BE IMPROVED BY USING FASTSPICE, ARNOLDI APPROXIMATION, OR OTHER AVAILABLE TOOLS.

VI. CONCLUSIONS

Existing literature on clock networks offers several highly successful algorithms, but does not detail end-to-end solutions to clock-network synthesis that can handle modern interconnect. Our work makes several contributions to this end. *First*, we develop specialized optimization algorithms necessary to bridge the gaps between well-known point-optimizations. Our emphasis is on robust techniques, that do not require tuning and are amenable to embedding into design flows. *Second*, we develop an EDA methodology for integrating clock-network optimization steps. *Third*, we describe a robust software implementation, called Contango, that outperforms best results from the ISPD'09 contest [13] by a factor of two.[2] *Fourth*, we scale our implementation to handle large industrial clock networks.

Our work relies in many ways on tree topologies and, by achieving strong empirical results, can make it difficult to justify the insertion of cross-links, advocated in previous literature. On the other hand, trees synthesized by our techniques can be integrated with meshes, as is common in modern CPU design [11]. In CPUs, better trees allow using smaller meshes, thereby reducing power of high-performance CPUs, increasing performance of embedded CPUs, and improving battery life of portable applications.

[2]The use of two wire sizes, two inverter types, and two process corners in the ISPD'09 contest is not a limitation of our algorithms and methodology. Likewise, any accurate delay evaluator can be used, including FastSpice, Arnoldi approximations, etc (with an appropriate PERL script).

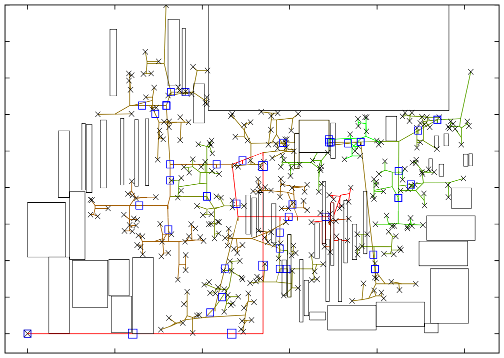

Fig. 3. The clock tree produced by Contango on *ispd*09*fnb*1. Sinks are indicated by crosses, buffers are indicated by blue rectangles. L-shapes are drawn as "diagonal wires" to reduce clutter. Wires are colored by a red-green gradient to reflect slow-down slacks, as described in Section III-B.

# sinks	CLR, ps	Skew, ps	Latency, ps	Cap., pF	🕒 min
200	13.47	2.124	506.8	52.21	2.2 (21)
500	14.84	2.174	528.0	99.53	6.28 (20)
1K	17.53	3.138	543.1	162.3	12.5 (20)
2K	16.56	3.136	543.9	276.1	19.3 (15)
5K	23.20	3.853	538.5	591.1	99.6 (22)
10K	25.54	5.562	538.0	1130	352.8 (23)
20K	32.47	10.46	546.8	2243	1867 (35)
50K	31.52	8.774	545.1	5243	16027 (45)

TABLE V

SCALABILITY ON TEXAS INSTRUMENTS BENCHMARKS. THE COLUMN "LATENCY" REPRESENTS MAXIMUM 1.2V LATENCIES. SPICE RUNS FOR EACH BENCHMARK ARE COUNTED IN PARENTHESIS.

REFERENCES

[1] K. D. Boese, A. B. Kahng, "Zero-Skew Clock Routing Trees with Minimum Wirelength," *ASIC*'92, pp.111-5.

[2] T.-H. Chao et al., "Zero Skew Clock Routing with Minimum Wirelength," *IEEE Trans. on Circ. & Sys.*, 39(11), pp. 799-814, 1992.

[3] J. Cong et al, "Bounded-Skew Clock and Steiner Routing," *ACM Trans. on DAES* 1998, pp. 341-388.

[4] M. Edahiro, "A Clustering-Based Optimization Algorithm in Zero-Skew Routings," *DAC*'93, pp. 612-616.

[5] L. v. Ginneken, "Buffer Placement in Distributed RC-tree Networks For Minimal Elmore Delay," *ISCAS*'90,pp.865-868.

[6] R. Ho, K. Mai, M. Horowitz, "The Future of Wires," *Proc. IEEE*, 89(4), pp. 490-504, 2001.

[7] J.-H. Huang, A. B. Kahng, C.-W. Tsao, "On Bounded-Skew Routing Tree Problem," *DAC*'95, pp.508-513.

[8] A. B. Kahng,C.-W. Tsao,"Practical Bounded-Skew Clock Routing," *J. VLSI Signal Proc.* 16(1997), pp.199-215.

[9] A. B. Kahng et al, "Interconnect Tuning Strategies for High-Performance ICs," *DATE*'98, pp. 471-478.

[10] J. Long, H. Zhou, S.O. Memik, "An O(nlogn) Edge-Based Algorithm for Obstacle-Avoiding Rectilinear Steiner Tree Construction," *ISPD*'08, pp. 126-133.

[11] R. S. Shelar, "An Algorithm for Routing with Capacitance/Distance Constraints for Clock Distribution in Microprocessors," *ISPD*'09, pp. 141-148.

[12] W. Shi, Z. Li, "A Fast Algorithm for Optimal Buffer Insertion," *IEEE Trans. on CAD* 24(6), pp.879-891,2005.

[13] C. Sze et al, "ISPD 2009 Clock-network Synthesis Contest," *ISPD*'09, pp. 149-150. http://www.sigda.org/ispd/contests/ispd09cts.html.

[14] R. S. Tsay, "An Exact Zero-Skew Clock Routing Algorithm," *IEEE Trans. on CAD* 12(2), pp.242-249, 1993.

[15] W. Zhao, Y. Cao, "New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration," *IEEE Trans. on Electron Devices*, 53(11), pp. 2816-2823, 2006.