Path Smoothing via Discrete Optimization

Michael D. Moffitt[†] David A. Papa[†♯] Zhuo Li[†] Charles J. Alpert[†]
[†]IBM Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78758
[♯]The University of Michigan, EECS Department, Ann Arbor, MI 48109
{mdmoffitt | dapapa | lizhuo | alpert}@us.ibm.com

ABSTRACT

A fundamental problem in timing-driven physical synthesis is the reduction of critical paths in a design. In this work, we propose a powerful new technique that moves (and can also resize) multiple cells simultaneously to smooth critical paths, thereby reducing delay and improving worst negative slack or a figure-of-merit. Our approach offers several key advantages over previous formulations, including the accurate modeling of objectives and constraints in the true timing model, and a guarantee of legality for all cell locations.

Categories and Subject Descriptors
B.7.2 [**Integrated Circuits**]: Design Aids – *placement and routing*
J.6 [**Computer-Aided Engineering**]: Computer-Aided Design
G.4 [**Mathematical Software**]: Algorithm Design and Analysis

General Terms: Algorithms, Design, Performance.

Keywords: Timing-driven placement, static timing analysis

1. INTRODUCTION

Timing-driven placement [3, 15, 20] is a critical step in any physical synthesis flow, and has received steadily increased attention in recent years. Due to its computational expense and complexity, several algorithms optimize timing objectives indirectly by relying on edge- or net-weighting methods to cast the problem into one of weighted wirelength-driven placement. Whether such approaches can truly be considered "timing-driven" – or instead, merely "timing-influenced" – remains a matter of debate.

A great deal of focus has been given specifically to the construction of cheap, incremental methods for improving timing along critical paths in an optimized design, a problem we loosely refer to as "path-smoothing." Prior work on this topic has varied widely in the treatment of *model accuracy*, including various assumptions about physical properties (e.g., gate delay and wire delay) as well as the set of constraints that must be enforced in the final solution (e.g., whether the design must be legalized, or subsequently buffered, or repowered, etc.). They also differ in the specific computational frameworks used to achieve the optimization (e.g., a local search, greedy algorithm, or dynamic program). These two considerations – choice of model and choice of algorithm – are typically strongly coupled, as a particular model often gives rise to a specific search-space or methodology.

One of the more popular approaches to incremental timing-driven placement in the literature is that of *linear programming* (LP) [11]. While several flavors exist, a conventional LP formulation typically involves the association of decision variables with the coordinate(s) of each gate or pin, and the expression of pairwise timing dependencies between these variables using linear constraints. Since the relationship between pin-to-pin wire delay and Manhattan distance is quadratic rather than linear, the inaccuracy of this linear model has been addressed in various ways. For instance, Choi and Bazargan [6] consider an objective function that minimizes total cell displacement to prevent cases where large cell movement invalidates the linear model. The model of Wang et al. [22] assumes that LP-based optimization is followed by perfect buffer insertion. A piecewise-linear approximation of the quadratic function is employed by Chowdhary et al. [7], along with additional constraints to capture net length and load capacitance using differential timing analysis. Luo et al. [14] optimize a weighted slack objective in which Elmore delays computed from the original placement are scaled linearly by a coefficient; to ensure that gates are not displaced by extremely large distances, movement of individual cells is constrained by bounding boxes, similar to the approach taken by Halpin et al. [10]. Most recently, Papa et al. [18] deploy an LP formulation in late stages of refinement after stripping all repeaters out from combinational logic and subsequently re-buffering long wires as a post-processing step.

Despite these efforts, linear programming formulations suffer from additional complications aside from their inability to capture a faithful delay model. Among these deficiencies includes the potential to create cell overlap; although several post-placement legalization techniques have been adopted in academia and industry, there is no guarantee that these procedures will preserve improvements made to timing. Furthermore, a trend in modern ASIC designs is the presence of large fixed macros that serve as blockages and limit the possible legal locations for movable logic. For such designs, an accurate model should avoid solutions that place gates on top of fixed obstacles, without needlessly predefining their relative relationships in advance. Finally, optimizing other discrete design parameters (such as gate sizes) and placement simultaneously requires an approach that accounts for decisions with finitely-many alternatives, since solutions produced by continuous gate-sizing [5] may degrade unacceptably when mapped to a standard cell library. Such continuous-to-discrete mappings present challenges for any of the aforementioned mathematical programming approaches.

In this paper, we introduce a new means to perform incremental, timing-driven placement. In contrast to prior efforts that approximate timing objectives using weighted wirelength-driven metrics (and approximate discrete decision variables using lossy, continuous models), our approach maintains a high degree of accuracy by explicitly encoding placement alternatives into a fully discretized graph-based representation, matching the true timing objectives as computed by an industrial static timing analysis engine. Specifically, we consider a formulation in which a finite set of *pre-legalized* candidate locations (as well as candidate power levels, threshold voltage assignments, etc.) are identified for each movable gate, allowing a more faithful and accurate encoding of pairwise delay, as well as enabling the avoidance of large fixed macros that serve as blockages. This formulation gives way to a *disjunctive timing graph*, a compact structure that captures these candidates along with conditional pairwise timing arcs. We then compute optimal solutions to this model using an efficient branch-and-bound framework that considers the simultaneous placement of multiple gates. The algorithm also exhibits an *anytime* property useful in cases where high-quality suboptimal solutions must be found quickly. To obtain upper bounds on worst negative slack (WNS) and a figure-of-merit (FOM), we develop a means to perform *Optimistic Static Timing Analysis* (OSTA), an extension of traditional static timing analysis that produces optimistic slack values even when only a subset of gates have been assigned to their respective candidates. Due to the discrete nature of our model, this algorithm extends to design decisions other than just the physical location of gates (for instance, to achieve simultaneous gate repowering).

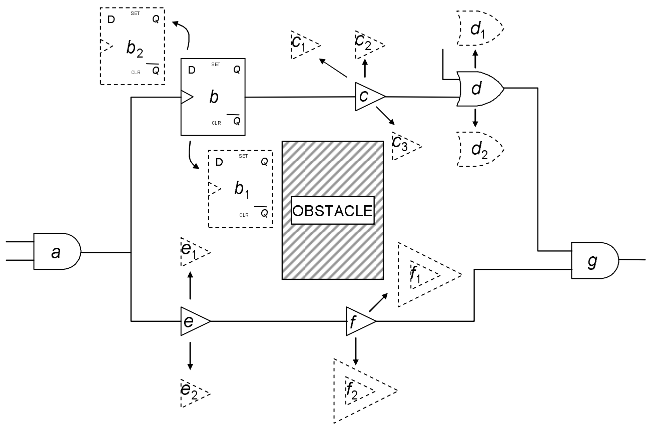

Figure 1: Gates *a* and *g* are fixed. Alternate candidate locations for movable gates *b*, *c*, *d*, *e*, and *f* have been determined. Gate *f* also has two candidate power levels.

2. BACKGROUND

In static timing analysis [19], a combinational logic circuit is encoded as a *timing graph* $G = (V, E)$, where each element $v \in V$ corresponds to a logic gate, and pair of vertices, $u, v \in G$, are connected by a directed edge $e(u, v) \in E$ if there is a connection from the output of gate u to the input of gate v. Each edge has an associated delay $\delta(u, v)$ indicating the delay between u and v.[1]

To determine the delay of the circuit, a topological traversal is performed on the graph beginning at the source(s). The actual arrival time $AAT(v)$ at a vertex v in the circuit is the latest arrival time of any of its predecessors after considering the net delay:

$$AAT(v) = \max_{\{u|e(u,v)\}} (AAT(u) + \delta(u, v))$$

The required arrival time $RAT(u)$ at a vertex u in the circuit is computed in a similar fashion, traversing backwards from the end of the circuit:

$$RAT(u) = \min_{\{v|e(u,v)\}} (RAT(v) - \delta(u, v))$$

After these two topological traversals have been made, the *slack* of any vertex v may be defined as the difference between required arrival time and actual arrival time:

$$slack(v) = RAT(v) - AAT(v)$$

Timing-driven placement seeks non-overlapping locations of cells such that the worst slack in the design is maximized.

The problem that *incremental* timing-driven placement aims to solve is the following: given an optimized design, select a subset of gates M from G (where M may just consist of a single gate) and find a new location for each gate in M such that the worst negative slack (WNS) in the entire subcircuit is improved:

$$WNS(G) = \min_{v \in V(G)} (min(0, slack(v)))$$

A figure-of-merit (FOM) component may also be optimized, which is equal to the sum of all slacks below a specified threshold:

$$FOM(G) = \sum_{v \in V(G)} min(0, slack(v) - threshold)$$

An algorithm that solves this problem is called a *transform*, using the terminology of [8, 21]. More generally, a transform is any optimization procedure designed to incrementally improve timing while preserving the logical correctness of a circuit. Transforms are invoked by *drivers*; for example, a driver designed for critical path optimization may attempt a transform on the 100 most critical cells.

[1]We assume that all delays are input-independent (i.e., taking the worst-case delay over all input combinations), as is the convention in static timing analysis. We also assume that $\delta(u, v)$ includes the appropriate gate delay.

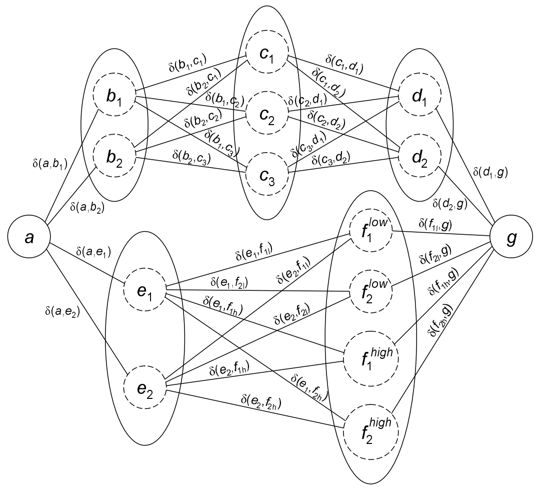

Figure 2: The *disjunctive timing graph* for our running example.

3. PROBLEM FORMULATION

In formulating our problem, we require three steps to be performed in sequence: identification of movable logic, generation of candidate assignments, and timing arc extraction.

3.1 Selection of Movables

The task of selecting a set of movable gates is shared by many timing-driven placement algorithms. Since our transform can be enacted by any high-level driver, we are free to assume that an external mechanism chooses individual gates for relocation (e.g., such as all imbalanced latches [18]). In expanding the movable logic to include additional gates, various heuristics have been proposed that incorporate the degree of neighbors' criticality [22, 14]. We combine the criticality adjacency network of [14] with an N-hop neighborhood, in which any gate within N steps of the targeted gate is included in the set of movable cells; however, we stress that our core timing-driven placement engine can be parameterized with any well-formed gate-selection strategy. All peripheral gates connected to movable logic form a set of fixed nodes.

3.2 Selection of Candidate Assignments

After the set of movable gates has been determined, we precompute a discrete set of *candidate assignments* for each. Our method imposes no restrictions on how these candidates are obtained, as there are several possible strategies ranging from simple to exotic:

- For a gate with coordinate (x, y), consider the candidates $(x \pm \Delta x, y)$ and $(x, y \pm \Delta y)$ for a given $(\Delta x, \Delta y)$. Such a set corresponds to the directions *up*, *down*, *left*, and *right*.
- The closest *feasible* locations to each of the candidates in the above set (i.e., respecting blockages and large fixed macros).
- The n nearest feasible locations closest to the gate's current coordinate, for some specified number n.
- A set of m or more locations obtained by m other incremental timing-driven placement algorithms for single gates.

The precomputation of candidate assignments bears some resemblance to graph-based approaches to buffer insertion [9]; however, it reflects a fundamental deviation from the vast majority of existing incremental timing-driven placement approaches that assume a continuous (and globally feasible) geometric plane. Refer to Figure 1 for an example in which each of five movable gates has between two and four candidates each. The presence of a large macro prevents gates from being placed toward the center of the subcircuit.

It is important to note that candidate assignments need not necessarily be new physical locations; for instance, cell *f* is shown to have two possible sizes, indicating different candidate power levels for the gate. This generalization permits the simultaneous optimization of placement and other transforms, in a similar spirit to [5] but imposing discrete (rather than continuous) values.[2]

[2]In practice, a discrete set of candidate values is more appropriate when working with a predefined cell library, and discretization from continuous values is NP-complete in general [13].

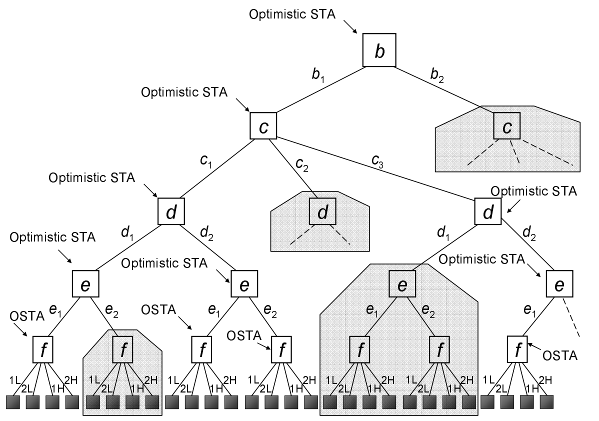

Figure 3: *Branch-and-bound* computes an upper bound on the worst negative slack at every node in search.

3.3 Timing Arc Extraction

The final step in our problem formulation is to construct a *conditional timing arc* for each pair (l_i, l_j) of candidate assignments between source and sink, which specifies the appropriate interconnect delay. We refer to the arcs between these nodes as being "conditional" since they depend on the chosen candidate(s). Our algorithm makes no assumptions about the correlation between the values of these timing arcs, and any delay model may be used. For nets with relatively few sinks, approximations such as Elmore delay may be appropriate. The delay between gates on higher degree nets may be obtained by more elaborate and accurate methods, such as querying a full-blown industrial timing engine, reconstructing Steiner trees from scratch [4] or via topological repair [1], or instead by cheaper methods of estimation.

4. THE DISJUNCTIVE TIMING GRAPH

Given our problem formulation, we now formally define an extension of the classical timing graph that captures the attributes we wish to express:

Definition: A *disjunctive timing graph* G is defined by a tuple (V, C, E), where (as in the traditional timing graph) each element $v \in V$ corresponds to a logic gate, and a pair of vertices, $u, v \in G$, are connected by a directed edge $e(u, v) \in E$ if there is a connection from the output of gate u to the input of gate v. The additional parameter C is a mapping from any gate $v \in V$ to a set of candidate assignments $\{v_1, ..., v_{C_v}\}$. Each edge has an associated *conditional delay function*, $\delta(u_i, v_j) \to \Re^+$, indicating the delay between any pair of candidates u_i and v_j. □

The disjunctive timing graph encodes all combinations of pairwise net delays, with each vertex corresponding to a "meta-node" representing a set of candidates. See Figure 2 for an illustration corresponding to our example. In subsequent sections, it will be useful to refer to a solution to a disjunctive timing graph, which is obtained simply by selecting a candidate for each gate and extracting the appropriate timing arcs.

Definition: A *solution* S to a disjunctive timing graph G is a mapping $V \to C(V)$, in which a single candidate is selected from the domain of each gate v in V. A solution corresponds to a traditional timing graph $G' = (V', E')$, in which the vertices V' of G' correspond to the candidates selected from G, and the weight of each edge $e'(u, v) \in E'$ is taken from $\delta(u_i, v_j)$, where u_i and v_j are the candidates chosen for gates u and v (respectively). □

A solution S to a disjunctive timing graph is deemed *optimal* with respect to an objective function O (e.g., worst negative slack, or FOM) if the value $O(S)$ is as good or better than $O(S')$ for every other solution S'. Observe that, in contrast to a traditional timing graph, a simple longest path calculation through the disjunctive graph does not suffice, even if optimizing for delay; such a computation maximizes the longest path, whereas we instead seek to select a set of candidates such that the longest path is *minimized*.

5. THE RATCHET ALGORITHM

The previous section alludes to one possible algorithm for the optimization of a disjunctive timing graph: generate every possible solution S, evaluate its cost, and return the best solution. However, when considering even moderately-sized problems, the computational expense of this brute-force procedure may be prohibitively expensive. Of course, if strict optimality is not required, other possibilities exist. A simple greedy strategy could consider the movement of each gate individually, yet in many practical cases, it is impossible to improve timing by moving a single gate (i.e., if a large gate is being driven by a weak driver, neither gate can be moved a significant distance from the other without imposing an electrical violation). To accommodate a wide range of instances, our algorithm must consider the simultaneous movement of multiple gates.

5.1 Recursive Branch-and-Bound Search

Branch-and-bound is a widely-studied, commonly used depth-first-search optimization technique. Rather than explore all possible combinations of assignments, branch-and-bound prunes partial solutions based on estimates of the objective function calculated during search. Backtracking occurs whenever the upper bound on the value of a partial solution is no better than that of the best found. Recent work in the coupling of graph-based procedures with branch-and-bound have demonstrated runtime reductions from days to seconds in floorplanning domains [17], although such advances have yet to be extended toward timing-driven placement.

In Figure 3, we display a possible search tree for our running example. The partial solution $S = \{(b \leftarrow b_1), (c \leftarrow c_1), (d \leftarrow d_1)\}$ is eventually extended to form a complete solution; however, in exploring the partial solution $S' = \{(b \leftarrow b_1), (c \leftarrow c_3), (d \leftarrow d_1)\}$, search is aborted. By visual inspection of Figure 1, the distance between candidates c_3 and d_1 is relatively large, and contributes to an excessively long delay in S'.

5.2 Optimistic Static Timing Analysis

One question raised by the backtracking framework is how to compute upper bounds on worst negative slack and FOM when only a subset of candidate assignments have been chosen. Traditional versions of Static Timing Analysis assume that all timing arcs are fixed, whereas in our model, a disjunctive set of choices remains until a fully instantiated solution in search is encountered.

We resolve this by performing a generalized version of Static Timing Analysis, which we call *Optimistic Static Timing Analysis* (OSTA). In OSTA, each edge in the graph corresponding to a source/sink pair is replaced with the most optimistic (or least constraining) possible timing arc. These weakened values may be safely propagated through the graph in place of any particular timing arc. Actual arrival times, required arrival times, and slacks for a partial solution S are computed as is typically done in STA, using these weakened values during propagation:

$$AAT(v) = \max_{\{u|e(u,v)\}} (AAT(u) + \min_{u_i \in C(u), v_j \in C(v)} (\delta_S(u_i, v_j)))$$

$$RAT(u) = \min_{\{v|e(u,v)\}} (RAT(v) - \min_{u_i \in C(u), v_j \in C(v)} (\delta_S(u_i, v_j)))$$

Since these weakened delay values must hold in any fully instantiated solution, the soundness of the procedure is preserved. Although the worst slack estimate calculated from this procedure may not be achievable in any complete solution, [3] we are guaranteed that no extension of the partial solution can improve upon it. For similar reasons, the summation over all optimistic slack values can provide an optimistic upper bound on FOM.

If a candidate assignment for a movable gate has been chosen, some entires in the conditional delay function may be disregarded (resembling a form of channeling used in artificial intelligence for hybrid constraint satisfaction problems [16]). For instance, in Figure 5, we consider the case when the partial solution S includes the decisions $\{(d \leftarrow d_1), (e \leftarrow e_1)\}$. Since no extension of this particular search node will consider the selection of candidate d_2, an entire column of entries can be ignored, raising the optimistic delay of the conditional function $\delta_S(c, d)$ up to 3ps from 2ps. A similar

[3]Interestingly, for subcircuits whose topology is that of a tree, a slight variation of OSTA can provide provably achievable upper bounds; however, due to space limitations, we omit the details here.

```
RATCHET(Design $D$, int *Iterations*)
1. Gate $G \leftarrow$ SELECTTARGETEDGATE($D$)
2. Set⟨Gate⟩ $M \leftarrow$ SELECTMOVABLES($D$, $G$)
3. Set⟨(Gate,Loc)⟩ *BestSol* $\leftarrow$ CURRENTASSIGNMENTS($M$)
4. for each $u \in M$
5.     Set⟨Loc⟩ $C_u \leftarrow$ GETCANDIDATEASSIGNMENTS($u$)
6. for each pair of adjacent gates $u, v \in M$
7.     for each candidate $u_i \in C_u$
8.         for each candidate $v_j \in C_v$
9.             $arcs_{u,v}(u_i, v_j) \leftarrow$ GETTIMINGARC($u_i$,$v_j$)
10. SOLVE($\oslash$, $M$, $C$, $arcs$)
11. return *BestSol*
```

```
SOLVE(Set⟨(Gate,Loc)⟩ $S$, Set⟨Gate⟩ $U$, Set⟨Set⟨Loc⟩⟩ $C$, $arcs$)
1. if (OBJECTIVE$_{OSTA}$($S$, $arcs$) $\leq$ OBJECTIVE(*BestSol*, $arcs$))
2.     return
3. if (TERMINATIONCRITERIONREACHED()) return
4. if ($U = \oslash$) *BestSol* $\leftarrow S$; return
5. $u \leftarrow$ CHOOSEMOVABLE($U$)
6. Set⟨Gate⟩ $U' \leftarrow U - \{u\}$
7. for each candidate $u_i \in C_u$
8.   Set⟨(Gate,Loc)⟩ $S' \leftarrow S \bigcup \{(u, u_i)\}$
9.   COMPUTEDAG($S'$, $arcs$)
10.   SOLVE($S'$, $U'$, $C$, $arcs$)
```

Figure 4: Pseudocode for the RATCHET algorithm.

effect is observed for $\delta_S(e, f)$. If both gates have been instantiated with candidate assignments, the actual timing arc between those specific candidates may be used.

Observe that in the case that all gates have only a single candidate assignment (or, equivalently, that a single candidate has been chosen for each movable gate), Optimistic STA reduces to traditional STA. It should also be noted that our branch-and-bound technique is an *anytime* algorithm, and may be interrupted prior to completion to obtain a suboptimal solution (e.g., based on a timeout limit, maximal number of nodes, etc.).

5.3 The Complete Flow

In Figure 4, we present the full pseudocode for our algorithm, named RATCHET. After selecting the targeted gate (line 1) and its surrounding movable neighbors (line 2), the current location of each gate is stored into the best known solution (*BestSol*). Candidate assignments for each movable gate are computed (lines 4 – 5), as well as the appropriate timing arcs for pairs of candidate assignments between adjacent gates (lines 6 – 9). These data are passed to the recursive function SOLVE (line 10). Upon its return (line 11), the optimized solution will be stored in *BestSol*.

Function SOLVE is given a partial solution of candidate assignments to gates (S), the unassigned gates (U), the candidate assignments (C), and the timing arcs ($arcs$). If branch-and-bound detects that the objective function cannot be improved in any extension of this node, search is aborted (lines 1 – 2). Similarly, if any other termination criteria are met (such as a timeout limit, or a maximal number of search nodes), the function returns (line 3). If a leaf node is reached, the fully instantiated solution is recorded as the best known (line 4). Otherwise, a movable gate is selected (line 5), removed from the set of unassigned gates (line 6), and each of its candidate assignments is attempted (line 7). For each location, the partial solution is extended (line 8) and the DAG is recomputed to reflect the new assignment (line 9). The function then recurses (line 10) and returns when all candidates have been attempted.

6. PRELIMINARY OBSERVATIONS

The integration of RATCHET into the Placement Driven Synthesis flow [2] at IBM is an ongoing process, yet preliminary results show strong potential. Initial observations suggest that RATCHET not only provides consistent improvement as a standalone transformation, but that it also diminishes the need for a broader family of local operations whose functionality is strictly subsumed by our approach. When sophisticated branch-and-bound techniques are employed, the solver itself appears to incur relatively negligible runtime; instead, the most expensive process is by far the extraction of values for conditional timing arcs, since (1) Steiner trees are being created on the fly to evaluate each pair of candidates, and (2) executing such queries has a tendency to "thrash" the static timing analysis engine upon which our algorithm is implemented. Since these temporary changes posed to the engine are ultimately not committed, this problem highlights a need for the engine to efficiently support hypothetical queries without invalidating previous timing information, a subject of our future work.

	d_1	~~d_2~~
c_1	7 ps	~~9 ps~~
c_2	3 ps*	~~5 ps~~
c_3	4 ps	~~2 ps~~

	f_1^{low}	f_2^{low}	f_1^{high}	f_2^{high}
e_1	4 ps	2 ps*	6 ps	5 ps
~~e_2~~	~~7 ps~~	~~1 ps~~	~~8 ps~~	~~3 ps~~

Figure 5: The delay functions $\delta(c, d)$ and $\delta(e, f)$. Here we show the case where the partial solution S includes the decisions $(d \leftarrow d_1)$ and $(e \leftarrow e_1)$. Entries marked with * indicate the conditional optimistic delay value.

7. CONCLUSIONS

The path smoothing problem in timing-driven placement is one that fundamentally admits a discrete solution space, and requires a corresponding methodology to efficiently perform discrete optimization. In response, we have proposed a new direction for incremental, timing-driven physical synthesis that directly optimizes timing objectives using accurate, high-fidelity models. RATCHET couples the graph-based principles of static timing analysis with a powerful branch-and-bound strategy to achieve efficient optimization of critical paths in late stages of refinement.

8. REFERENCES

[1] A. H. Ajami and M. Pedram. Post-layout timing-driven cell placement using an accurate net length model with movable Steiner points. In *Proc. of ASP-DAC 2001*, pages 595–600, 2001.

[2] C. J. Alpert, S. K. Karandikar, Z. Li, G.-J. Nam, S. T. Quay, H.. Ren, C. N. Sze, P. G. Villarrubia, M. C. Yildiz. Techniques for fast physical synthesis. In *Proc. of the IEEE* 95(3), pages 573–599, 2001.

[3] M. Burstein and M. N. Youssef. Timing influenced layout design. In *Proc. of DAC 1985*, pages 124–130, 1985.

[4] K.-H. Chang, I. L. Markov, and V. Bertacco. Safe delay optimization for physical synthesis. In *Proc. of ASP-DAC 2007*, pages 628–633, 2007.

[5] W. Chen, C.-T. Hsieh, and M. Pedram. Simultaneous gate sizing and placement. In *TCAD* 19(2), pages 206–214, 2000.

[6] W. Choi and K. Bazargan. Incremental placement for timing optimization. In *Proc. of ICCAD 2003*, pages 463–466, 2003.

[7] A. Chowdhary et al. How accurately can we model timing in a placement engine? In *Proc. of DAC 2005*, pages 801–806, 2005.

[8] W. E. Donath, P. Kudva, L. Stok, P. Villarrubia, L. N. Reddy, A. Sullivan, K. Chakraborty. Transformational placement and synthesis. In *Proc. of DATE 2000*, pages 194–201, 2000.

[9] Y. Gao and D. F. Wong. A graph based algorithm for optimal buffer insertion under accurate delay models. In *Proc. of DATE 2001*, pages 535–539, 2001.

[10] B. Halpin, C. Y. R. Chen, and N. Sehgal. Timing driven placement using physical net constraints. In *Proc. of DAC 2001*, pages 780–783, 2001.

[11] M. A. B. Jackson and E. S. Kuh. Performance-driven placement of cell based IC's. In *Proc. of DAC 1989*, pages 370–375, 1989.

[12] A. B. Kahng, S. Mantik, and I. L. Markov. Min-max placement for large-scale timing optimization. In *Proc. of ISPD 2002*, pages 143–148, 2002.

[13] W. N. Lee. Strongly NP-Hard Discrete Gate Sizing Problems. In *Proc. of ICCD 1993*, pages 468–471, 1993.

[14] T. Luo, D. Newmark, and D. Z. Pan. A new LP based incremental timing driven placement for high performance designs. In *Proc. of DAC 2006*, pages 1115–1120, 2006.

[15] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for FPGAs. In *Proc. of FPGA*, pages 203–213, 2000.

[16] M. D. Moffitt, B. Peintner, and M. E. Pollack. Augmenting disjunctive temporal problems with finite-domain constraints. In *Proc. of AAAI 2005*, pages 1187–1192, 2005.

[17] M. D. Moffitt, M. E. Pollack. Optimal rectangle packing: a meta-CSP approach. In *Proc. of ICAPS 2006*, pages 93–102, 2006.

[18] D. A. Papa, T. Luo, M. D. Moffitt, C. N. Sze, Z. Li, G.-J. Nam, C. J. Alpert, I. L. Markov RUMBLE: an incremental, timing-driven, physical-synthesis optimization algorithm. In *Proc. of ISPD 2008*, pages 2–9, 2008.

[19] S. Sapatnekar. Timing. Kluwer Academic Publishers, Norwell, MA, 2004.

[20] W. Swartz and C. Sechen. Timing driven placement for large standard cell circuits. In *Proc. of DAC 1995*, pages 211–215, 1995.

[21] L. Trevillyan, D. S. Kung, R. Puri, L. N. Reddy, and M. A. Kazda. An integrated environment for technology closure of deep-submicron IC designs. *IEEE Design & Test of Computers*, 21(1):14–22, 2004.

[22] Q. B. Wang, J. Lillis, and S. Sanyal. An LP-based methodology for improved timing-driven placement. In *Proc. of ASP-DAC 2005*, pages 1139–1143, 2005.