RUMBLE: An Incremental, Timing-driven, Physical-synthesis Optimization Algorithm

David A. Papa[†♯], Tao Luo[‡], Michael D. Moffitt[♯], C. N. Sze[♯],
Zhuo Li[♯], Gi-Joon Nam[♯], Charles J. Alpert[♯] and Igor L. Markov[†]
†University of Michigan / EECS Department / Ann Arbor, MI 48109
‡University of Texas at Austin / Department of ECE / Austin, TX 78712
♯IBM Austin Research Lab / 11501 Burnet Rd. / Austin, TX 78758
iamyou@umich.edu,tluo@ece.utexas.edu,{mdmoffitt,csze,lizhuo,gnam,alpert}@us.ibm.com,imarkov@umich.edu

***Abstract*—Physical synthesis tools are responsible for achieving timing closure. Starting with 130nm designs, multiple cycles are required to cross the chip, making latch placement critical to success. We present a new physical synthesis optimization for latch placement called RUMBLE (Rip Up and Move Boxes with Linear Evaluation) that uses a linear timing model to optimize timing by simultaneously re-placing multiple gates. RUMBLE runs incrementally and in conjunction with static timing analysis to improve the timing for critical paths that have already been optimized by placement, gate sizing, and buffering. Experimental results validate the effectiveness of the approach: our techniques improve slack by 41.3% of cycle time on average for a large commercial ASIC design.**

I. INTRODUCTION

Physical synthesis is a complex multi-phase process primarily designed to achieve timing closure, though power, area, yield and routability also need to be optimized. Starting with 130nm designs, signals can no longer cross the chip in a single cycle, which means that *pipeline latches* need to be introduced to create multi-cycle paths. This problem becomes more pronounced for 90-, 65- and 45-nanometer nodes, where interconnect delay increasingly dominates gate delay [10]. Indeed, for high-performance ASIC scaling trends, the number of pipeline latches increases by $2.9\times$ at each technology generation, accounting for as much as 10% of the area of 90nm designs [8] and as many as 18% of the gates in 32nm designs [22]. Hence, the proper placement of pipeline latches is a growing problem for timing closure.

The choice of computational techniques for latch placement depends on where this optimization is invoked in a physical synthesis flow. To this end, we review the major phases of such flows following [1], [4].

1) **Global placement** computes non-overlapping physical locations for gates and typically optimizes half-perimeter wirelength (HPWL) or weighted HPWL. As part of this phase, usually some amount of detail placement is done, and legalization is called to ensure a legal optimization result.
2) **Electrical correction** fixes capacitance and slew violations with gate sizing and buffering.
3) **Legalization** is an incremental placement capability that removes overlaps caused by optimization with minimal disturbance to placement and timing.
4) **Timing analysis** assesses the speed of the design and determines if performance targets are met. Among other metrics, this phase determines the *slack* of every path in the circuit — the difference between the clock period and how long it takes a signal to traverse the path.
5) **Detail placement** moves gates to further reduce wirelength and improve timing. In this phase it is possible to do *timing-driven* detail placement wherein timing information is explicitly considered when optimizing gate placements.
6) **Critical-path optimization** identifies most-critical paths and focuses on techniques to improve the slack for the worst timing violations. Relevant optimizations include buffering, gate sizing and incremental synthesis [23].
7) **Compression** optimizes the remaining paths that violate timing constraints when improvements on most-critical paths are no longer possible. The goal is to *compresses* the timing histogram and reduce number of negative-slack paths that require designer intervention.

The flow can be repeated with net weighting and timing-driven placement to further improve results.

One can think of physical synthesis as progressing with variable detail and variable accuracy. For example, during global placement, large changes are made to the design using a coarse objective (such as wirelength) that is oblivious to timing considerations. Later, one may perform more accurate optimization using an Elmore interconnect-delay model with Steiner-tree estimates for net capacitance. As timing begins to converge, one can apply more costly, fine-grained buffering along actual detailed routes using a statistical timing model.

Figure I(a)-(d) illustrates the complications of using existing global placement techniques to solve the latch placement problem for a single two-pin net. Assume that, for all four figures, the source A and sink B are fixed in their respective locations, and that global placement must find the correct location for the latch. This example is representative of situations in which a fixed block in one corner of the chip must communicate

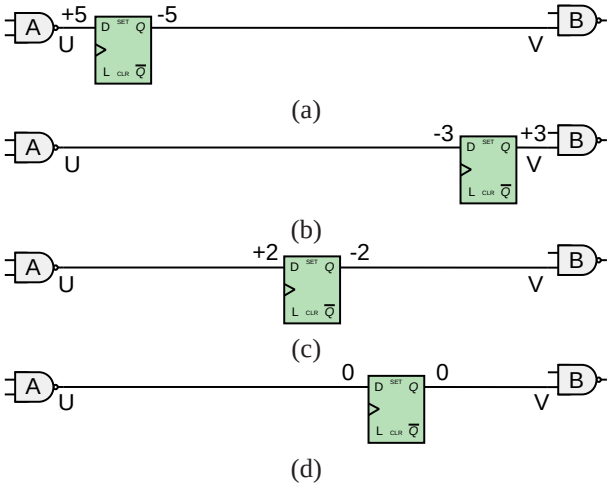

Fig. 1. The placement of a pipeline latch impacts the slacks of both input and output paths. A wirelength objective does not capture the timing effects of this situation.

with a block in the opposite corner, but signal delay inevitably exceeds a single clock period. All four placements have equal wirelength, therefore unless global placement is timing driven, the placement of the latch between A and B is arbitrary. Consider the following scenarios:

- Suppose the placement tool chooses (a), which is the worst location for the latch. In this case, the latch is so far from B that the timing constraint at B cannot be met. This results in a slack on the input net (U) of $+5$ns and a slack on the the output net (V) of -5ns (even after optimal buffering).[1]
- With a second iteration of physical synthesis, timing-driven placement could try to optimize the location of this latch by adding net weights. Any net weighting scheme will assign a higher weight to net V than U, resulting in a placement where the latch is very close to B, as in (b). While the timing is improved, there now is a slack violation on the other side of the latch with -3ns of slack on U and $+3$ns on V.
- A global or detail placer could use a quadratic wirelength objective to handle these kinds of nets, giving the location (c), which, while better than (a) and (b), is suboptimal.
- To achieve the optimal location with no critical nets (0 slack on U and V), the latch must be placed as shown in (d). In this case, there is only one location that meets both constraints.

This example suggests that wirelength optimization is not well-suited for latch placement, especially when there is little room for error. Instead, one must be able to couple latch placement with timing analysis and model the impact of buffering. The problem is more complex in practice, and some aspects are not illustrated above. In particular, many latches have buffer trees in the immediate fan-in and fan-out. Such complications pose additional challenges that we address in this work. We make the following contributions.

- We show that a linear-wire-delay model is sufficient to model the impact of buffering for the latch placement problem.
- We develop RUMBLE, a linear-programming-based, timing-driven placement algorithm which includes buffering for slack-optimal placement of individual latches under this model and show its effectiveness experimentally.
- We extend RUMBLE to improve the locations of individual logic gates other than latches. Further, we show how to find the optimal locations of multiple gates (and latches) *simultaneously*, with additional objectives. Incremental placement of multiple cells requires additional care to preserve timing assumptions, optimizing a set of slacks instead of a single slack, while also biasing the solution towards placement stability. We describe how RUMBLE handles these situations.
- We empirically validate proposed transforms and the entire RUMBLE flow. We show how these techniques can be used to significantly improve initial latch placement in a reasonably optimized ASIC design with "do no harm" acceptance criteria that reject solutions if any quality metrics are degraded. This facilitates the use of RUMBLE later in physical synthesis.

The remainder of the paper is organized as follows. Section II discusses background and previous work. Section III describes the timing model we use in this work. Section IV describes how RUMBLE performs timing-driven placement. Section V describes the RUMBLE algorithm. Section VI shows experimental results. Conclusions are drawn in Section VII.

II. BACKGROUND

Several approaches improve IC performance by modifying wirelength-driven global placement through timing-based net weights [9], [11]–[13], [15], [18]. Such algorithms are generally referred to as timing-driven placement, but the literature has not yet considered the impact of buffering on latch placement during global placement. Due to the lack of such algorithms, it is inevitable that some latches will be suboptimally placed during global placement. Therefore, new algorithms are needed for post-placement performance-driven incremental latch movement.

We introduce a high-level description of the incremental latch placement problem below, and elaborate on its multi-move formulation in Section IV. Given an optimized design and a small set of gates M (M may consist of a single latch) find new locations for each gate in M and new buffering solutions for nets incident to M such that the timing characteristics of the design are improved.

While moving a cell can improve delay, especially if it has been poorly placed, moving a latch has special significance since it facilitates time-borrowing: reallocating circuit delay from a longer (slow) combinational stage to a shorter (fast) combinational stage. This fact offers a particularly significant boost to our basic approach, and is enhanced even further when surrounding gates are also free to move.

A solution to this problem is called a *transform* using the terminology of [23]. A transform is an optimization designed

[1]The nets in each scenario could include buffers without changing the trends discussed.

to incrementally improve timing. Other examples of transforms include buffering a single net, resizing a gate, cloning a cell, swapping pins on a gate, etc. The way transforms are invoked in a physical synthesis flow is determined by the *drivers*. For example, a driver designed for critical path optimization may attempt a transform on the 100 most critical cells. A driver designed for the compression stage (see Section I) may attempt a transform on every cell that fails to meet its timing constraints.

A driver has the option of avoiding transforms that may harm the design (e.g., generating new buffering solutions inferior to the original) and can then reject this solution. This *do no harm* philosophy of optimization has received significant recognition in recent work [5], [20]. The RUMBLE approach adopts this same convention which makes it more trustworthy in a physical synthesis flow.

While no previous work has attempted to solve this particular problem, other solutions do exist that may be able to help with the placement of poorly placed latches. The authors of [24] propose a linear programming formulation that minimizes downstream delay to choose locations for gates in field-programmable gate arrays (FPGAs). The authors of [6] model static timing analysis (STA) in a linear programming formulation by approximating the quadratic delay of nets with a piecewise-linear function. Their formulation's objective is to maximize the improvement in total negative slack of timing end points. The authors of both approaches conclude that the addition of buffering would improve their techniques [6], [24]. When these transformations are applied at the same point in a physical synthesis flow that we propose, they will be restricted by previous optimizations. When applied somewhat earlier (e.g., following global placement) they are incapable of certain improvements. Namely, downstream optimizations, such as buffer insertion, gate sizing, and detail placement may invalidate the optimality of latch placement. Therefore our technique focuses on the bad latch placements that we observed in large commercial ASIC designs after state-of-the-art physical synthesis optimizations. However, we believe that RUMBLE may be too disruptive to use after routing.

III. THE RUMBLE TIMING MODEL

We now introduce the timing model critical to RUMBLE's success.

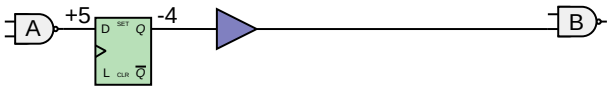

Fig. 2. A poorly-placed latch with buffered interconnect. In this case, the buffer must be moved or removed in order to have the freedom to move the latch far enough to fix the path.

Figure III shows an intuitive example of the problem when we try to find new locations for movable gates. Similar to Figure I, the latch has to be moved to the right to improve timing. However, since the latch drives a buffer which is placed next to it, we must move the buffer in order to improve the slack of the latch, and other complications are illustrated by Figure 3. At the same time, the optimal new location of the latch depends on how the input and output nets are buffered. As a result, the optimal approach is to simultaneously move the latch and perform buffering, but this is computationally prohibitive because a typical multiple-objective buffering algorithm runs in exponential time. As mentioned in Section I, we propose a sequential approach in which we first compute the new locations for a selected set of movable gates based on timing estimation considering buffers. Then, buffering is applied to the input and output nets of the selected movable gates. Being practical, effective and efficient, this approach can be integrated into a typical VLSI physical synthesis flow. The calculation of optimal movement depends on a simple but effective buffered-interconnect delay model, which is discussed the following section.

A. Linear Buffered-Path Delay Estimation

Buffering has become indispensable in timing closure and cannot be ignored during interconnect delay estimation [3], [7], [22]. Therefore to calculate new locations of movable gates, one must adopt a buffering-aware interconnect delay model that accounts for future buffers. We found that the linear delay model described in [3], [17] is best suited for this application. In this model, the delay along an optimally buffered interconnect is

$$delay(L) \quad = \quad L(R_bC + RC_b + \sqrt{2R_bC_bRC}) \qquad (1)$$

where L is the length of a 2-pin buffered net, R_b and C_b is the intrinsic resistance and input capacitance of buffers and gates while R and C are unit wire resistance and capacitance respectively.

Empirical results in [3] indicate that Equation 1 is accurate up to 0.5% when at least one buffer is inserted along the net. Furthermore, our own empirical results in Section VI-B suggest a 97% correlation between this linear delay model and the output of an industry timing analysis tool.

B. The Timing Graph

In RUMBLE, a set of movable gates is selected, which must include fixed gates or input/output ports to terminate every path. Fixed gates and I/Os help formulate timing constraints and limit the locations of movables. In Figure III-B(a), we assume that new locations have to be computed for the latch and the two OR gates, while all NAND gates are kept fixed.

In the timing graph, each logic gate is represented by a node, while a latch is represented by two nodes because the inputs and outputs of a latch are in different clock cycles and can have different slack values. Each edge represents a driver-sink path along a net and is associated with a delay value which is linearly proportional to the distance between the driver and the sink gate. In other words, we decompose each multi-pin net into a set of 2-pin edges that connect the driver to each sink of the net. This simplification is crucial to our linear delay model and is valid because the linear relationship can be preserved for the most critical sinks by decoupling less critical paths with buffers. Therefore the 2-pin edge model in the timing graph can guide the computation of new locations for the movable gates.

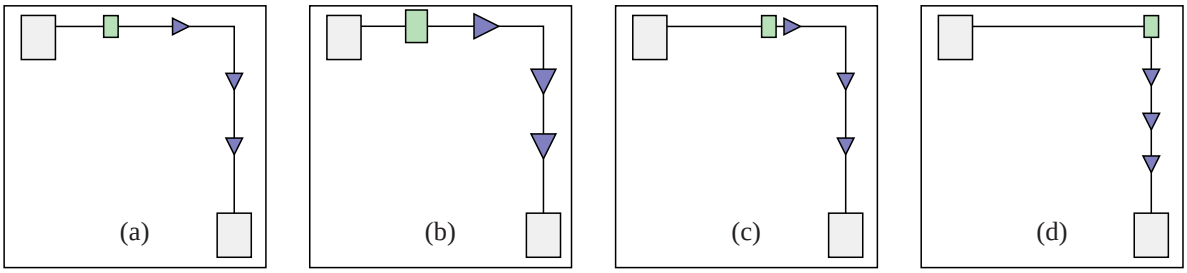

Fig. 3. The layout in (a) has a poorly-placed latch, and existing critical path optimizations do not solve the problem. Repowering the gates may improve the timing some in (b), but if it cannot fix the problem, the latch must be moved. Moving the latch up to the next buffer, shown in (c), does not give optimization enough freedom. If we move the latch but do not re-buffer in (d), timing may degrade. Figure 9(d) shows the ideal solution to this problem.

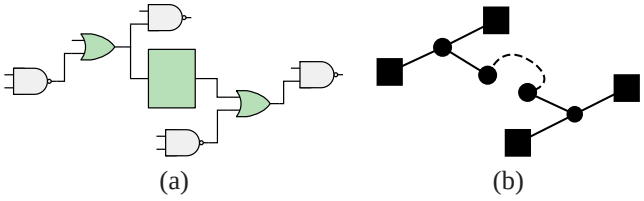

Fig. 4. (a) An example subcircuit and (b) corresponding timing graph used in RUMBLE. The AATs or RATs of unmovable objects (squares) are considered known. STA is performed on movable objects (round shapes).

In the timing graph, an edge which represents a timing arc is created only for (1) each connection between the movable gates, and (2) each connection between a movable gate and a fixed gate. This is because we only care about the slack change due to the displacement of movable gates. For the subcircuit in Figure III-B(a), the resultant timing graph is shown in Figure III-B(b).

For each fixed gate, we assume that the required arrival time (RAT) and the actual arrival time (AAT) are fixed. The values of RAT and AAT are generated by a static timing analysis (STA) engine using a set of timing assertions created by designers. An in-depth exposition of STA can be found in [16], [21] along with algorithms to generate RAT and AAT. A movable latch corresponds to two nodes in the timing graph, one for the data input pin and one for the output pin. For the input pin, the RAT is fixed based on the clock period. Similarly, the AAT is fixed for the latch's output pin. Based on all the fixed RAT and AAT at fixed gates and latches, the AAT and RAT are propagated along the edges according to the delay of the timing arcs. The values of AAT are propagated forward to fan-out edges, adding the edge delay to the AAT. On the contrary, RATs are propagated backward along the fan-in edges, subtracting the edge delay from the RAT values. Details of edge delay, RAT and AAT calculation in our algorithm are covered in Section IV.

IV. TIMING-DRIVEN PLACEMENT

The goal of RUMBLE is to find new locations for movable gates in a given selected subcircuit such that the overall circuit timing improves. Therefore we maximize the minimum (worst) slack of source-to-sink timing arcs in the subcircuit. In contrast to other objectives used in previous work, we select this objective because we are targeting critical-path optimization. Hence, we prefer 1 unit of worst-slack improvement over 2 units of slack improvement on less-critical nets. Below we introduce the timing-driven placement technique in RUMBLE that directly maximizes minimum slack. In the following placement formulation we account for the timing impact of our changes by implicitly modeling static timing analysis in our timing graph. In this work, we estimate net length by the half-perimeter wirelength (HPWL) and then scale it to represent net delay. More accurate models are possible, but may complicate optimization.

A. Problem Formulation

Consider the problem of maximizing the minimum slack of a given subcircuit G with some movable gates and some fixed gates, or ports.

Let the set of nets in the subcircuit be $\mathbf{N} = n_0, n_1, \ldots, n_h$. Let the set of all gates in the subcircuit (movable and fixed) be $\mathbf{G} = g_0, g_1, \ldots, g_f$. Let the set of movable gates in the subcircuit (a subset of $\mathbf{G}$) be $\mathbf{M} = m_0, m_1, \ldots, m_k$.

τ is a technology dependent parameter that is equal to the ratio of the delay of an optimally-buffered, arbitrarily-long wire segment to its length

$$\tau = \frac{delay(wire)}{length(wire)} \tag{2}$$

The following equations govern static timing analysis and are used in the next section. A timing arc is specified for a given net n driven by gate u and having sink v as $n_{u,v}$. The delay of a gate g is D_g.

The Required Arrival Time (RAT) of a combinational gate g is

$$R_g = \min_{o_j : 0 \le j \le m} \{R_{o_j} - \tau * \mathrm{HPWL}(n_{g,o_j}) - D_g\} \tag{3}$$

The Actual Arrival Time (AAT) of a combinational gate g is

$$A_g = \max_{i_j : 0 \le j \le l} \{A_{i_j} + \tau * \mathrm{HPWL}(n_{i_j,g}) + D_g\} \tag{4}$$

Given a clocked latch r, we assume for simplicity that the RAT (R_r) and AAT (A_r) are fixed and come from the timer. Unclocked latches are treated similarly to the combinational gates above.

The slack of a timing arc $n_{p,q}$ connecting two gates (combinational or sequential, movable or fixed) p and q is

$$S_{n_{p,q}} = R_q - A_p - \tau * \mathrm{HPWL}(n_{p,q}) \tag{5}$$

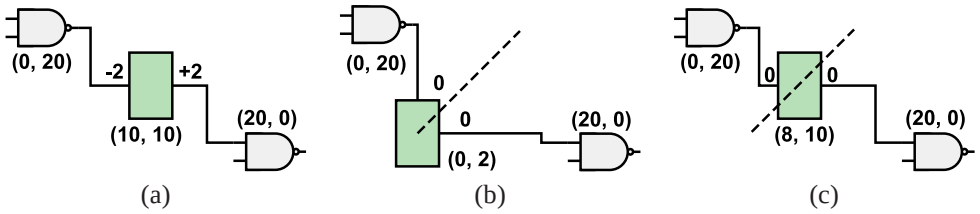

Fig. 5. In many subcircuits there are multiple slack-optimal placements. In RUMBLE we add a secondary objective to minimize the displacement from the original placement. This helps to maintain the timing assumptions made initially and reduces legalization issues. (a) shows the initial state of and example subcircuit, (b) a slack-optimal solution commonly returned by LP solvers, all optimal solutions lie on the dotted line and (c) a solution given by RUMBLE that maximizes worst-slack then minimizes displacement.

B. The RUMBLE Linear Program

We define a linear program to maximize the minimum slack S of a subcircuit as follows.

VARIABLES:

$$\begin{array}{lllllllll} S & & & \cup & \forall_{n\in N} & : & S_n & \cup \\ \forall_{m\in M} & : & \beta_x^m & \cup & \forall_{m\in M} & : & \beta_y^m & \cup \\ \forall_{n\in N} & : & U_x^n & \cup & \forall_{n\in N} & : & U_y^n & \cup \\ \forall_{n\in N} & : & L_x^n & \cup & \forall_{n\in N} & : & L_y^n & \cup \\ \forall_{m\in M} & : & R_m & \cup & \forall_{m\in M} & : & A_m & \end{array} \tag{6}$$

Of the above, β are independent variables for gate locations. The U and L variables represent upper and lower bounds of nets (highest and lowest coordinates in the $x-$ and $y-$ directions) for computing HPWL. R and A compute required and actual arrival times. Each S_n computes the slack of a particular net, while S is the minimum slack of all nets.

OBJECTIVE: **Maximize** S

CONSTRAINTS: For every gate g_j on net n_i

$$U_x^{n_i} \geq \beta_x^{g_j}, \quad U_y^{n_i} \geq \beta_y^{g_j} \tag{7}$$

$$L_x^{n_i} \leq \beta_x^{g_j}, \quad L_y^{n_i} \leq \beta_y^{g_j} \tag{8}$$

For every movable gate m_i and sink it drives g_j via net n_k

$$R_{m_i} \leq R_{g_j} - \tau * (U_x^{n_k} - L_x^{n_k} + U_y^{n_k} - L_y^{n_k}) - D_g \tag{9}$$

For every movable gate m_i and gate g_j that drives one of its inputs via net n_k

$$A_{m_i} \geq A_{g_j} + \tau * (U_x^{n_k} - L_x^{n_k} + U_y^{n_k} - L_y^{n_k}) + D_g \tag{10}$$

For every timing arc in the subcircuit $n_{p,q}$ associated with net n_i

$$S_{n_i} \leq R_q - A_p - \tau * (U_x^{n_i} - L_x^{n_i} + U_y^{n_i} - L_y^{n_i}) \tag{11}$$

For each net n_i:

$$S \leq S_{n_i} \tag{12}$$

C. Extensions to Minimize Displacement

The linear program of RUMBLE is defined to maximize the minimum slack of a subcircuit. Additional objectives can be considered as well, such as total cell displacement, which sums Manhattan distances between cells' original and new locations. We subtract the minimum slack objective from a weighted total cell displacement term to avoid unnecessary cell movement. The weight W_d for the total cell displacement objective is set to a small value. Therefore the weighted total displacement component is used as a tie-breaker and has little impact on worst-slack maximization. Instead, the combined objective is maximized by a slack-optimal solution closest to cells' original locations. During incremental timing-driven placement, minimizing total cell displacement encourages higher placement stability and often translates into fewer legalization difficulties.

Figure 5 shows an example of the RUMBLE formulation with and without the total displacement objectives. The only movable object in Figure 5(a) is the latch. An input net n_1 and an output net n_2 are connected to the latch with slacks -2 and $+2$ respectively. Figure 5(b) shows the optimal LP solution without the total displacement objective. The Manhattan net length of n_1 is reduced from 20 to 18, and the net length of n_2 is increased from 20 to 22. This improves the worst slack of the subcircuit from -2 to 0. However, the latch moves a large distance. Figure 5(c) illustrates that including the total displacement objective may preserve optimal slack, while minimizing latch displacement.

In order to minimize displacement by adding a new objective, we introduce the following variables and constraints to the linear program.

DISPLACEMENT VARIABLES:

$$\begin{array}{lllllllll} \forall_{m\in M} & : & \delta_x^m & \cup & \forall_{m\in M} & : & \delta_y^m & \cup \\ \forall_{m\in M} & : & \phi_x^m & \cup & \forall_{m\in M} & : & \omega_x^m & \cup \\ \forall_{m\in M} & : & \phi_y^m & \cup & \forall_{m\in M} & : & \omega_y^m & \end{array} \tag{13}$$

DISPLACEMENT CONSTRAINTS:

For every movable gate m_i, $\alpha_x^{m_i}$ and $\alpha_y^{m_i}$ denote the original x and y coordinates. The upper and lower bounds of the new and original coordinates ϕ and ω in each dimension are:

$$\begin{array}{llllll} \phi_x^{m_i} & \geq & \beta_x^{m_i}, & \omega_x^{m_i} & \leq & \beta_x^{m_i} \\ \phi_y^{m_i} & \geq & \beta_y^{m_i}, & \omega_y^{m_i} & \leq & \beta_y^{m_i} \\ \phi_x^{m_i} & \geq & \alpha_x^{m_i}, & \omega_x^{m_i} & \leq & \alpha_x^{m_i} \\ \phi_y^{m_i} & \geq & \alpha_y^{m_i}, & \omega_y^{m_i} & \leq & \alpha_y^{m_i} \end{array} \tag{14}$$

The displacements δ^{m_i} for a movable gate m_i are defined as

$$\delta_x^{m_i} = \phi_x^{m_i} - \omega_x^{m_i}, \quad \delta_y^{m_i} = \phi_y^{m_i} - \omega_y^{m_i} \tag{15}$$

D. Extensions to Improve the Slack Histogram

The minimum slack is the worst slack in a subcircuit. For two subcircuits with identical worst slack, it is possible that one subcircuit has few critical paths with worst slack while the other one has many. A timing optimization has to improve both the worst slack and the overall figure of merit (FOM) in a subcircuit. FOM is defined as the sum of all slacks below

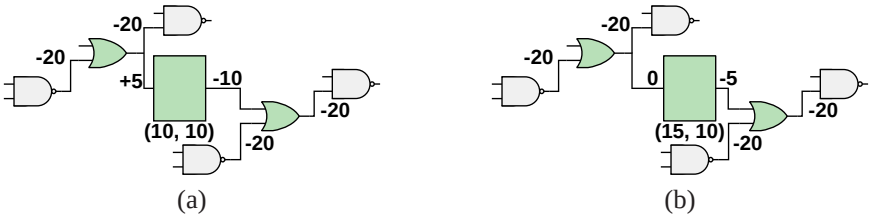

Fig. 6. (a) An example subcircuit with an imbalanced latch whose worst-slack cannot be improved. Nevertheless, it is possible to improve timing of the latch while maintaining slack-optimality. By including a FOM component in the objective, the total negative slack can be reduced, as shown in (b).

a threshold. If the slack threshold is zero, FOM is equivalent to the total negative slack. With the minimum slack as the only objective, a small improvement in the worst slack may cause a large FOM degradation. Therefore we must add a FOM component to the optimization objective. The balance between the minimum slack and the FOM is controlled by a parameter W_f, which is set to a relatively small value because the worst slack objective is more important.

Figure 6 shows another scenario where the FOM component may help. During optimization, it may not be always possible to improve the minimum slack of the subcircuit. In that case, we can still reduce the number of critical cells by improving the FOM. In Figure 6, there are three movables in the subcircuit. The minimum slack of the subcircuit is -20, and it is not possible to improve the minimum slack by moving any of the gates. With the additional FOM component in the objective, the FOM of the subcircuit is improved from -90 to -85, as shown in Figure 6(b).

Let S_n denote the slack on net n, then the combined objective has the displacement and FOM components
Maximize:

$$\begin{array}{rcl} S & - & W_d \sum_{m \in M} (\delta_x^m + \delta_y^m) \\ & + & W_f \sum_{n:n \in N, S_n < T_s} S_n \end{array} \quad (16)$$

where T_s is the small slack threshold used to compute the FOM. We have earlier assumed W_f and W_d to be small, with $W_d < W_f$. In our implementation we set W_f to 0.005 times the absolute value of the average slack in the subcircuit, and we set W_d to 10^{-6}. These additional terms change the optimal region, but because the weights are so small the combined optimal region is very near the slack-optimal region.

E. Preventing Harm to FOM

The primary goal of the RUMBLE linear program as presented in previous sections is to maximize the worst slack of the subcircuit. We define two additional objectives — one preserves the initial solution as much as possible, the other can improve the slack histogram when the worst slack cannot be further improved. However, it is possible that in order to improve a single worst slack path, multiple paths may degrade to the point of being critical. If RUMBLE is deployed late enough in a physical synthesis flow, the corresponding FOM degradation may be undesirable. To address this problem, we have devised an additional constraint which, at the cost of reduced improvement in worst slack, can prevent this type of FOM degradation. When FOM should not be degraded, we add the following constraints to the RUMBLE linear program to preserve FOM.

For each net n_k whose slack is greater than the slack threshold T_s, add the following constraint.

$$S_{n_k} \geq T_s \quad (17)$$

This addition may over-constrain the linear program, in which case it is not possible to improve the worst slack without harming FOM.

V. THE RUMBLE ALGORITHM

In this section we discuss the details of the RUMBLE algorithm, which employs the linear program from the previous section to incrementally improve the timing of poorly placed latches.

A. Subcircuit Selection

RUMBLE identifies *imbalanced latches*, which we define as those that exhibit positive slack on their inputs and negative slack on their outputs (or vice versa). As illustrated in Figure I, the movement of any such imbalanced latch has the potential to improve timing, even if all surrounding cells are held fixed. More generally, however, the neighbors and extended neighbors of the targeted latch may also be included to form a set M of movable cells. In our technique, shown in Figure 8, we adopt a basic N-hop neighborhood approach, where any gate within N steps of the imbalanced latch is included in the set of movable cells. This requires both a forward sweep (to collect sinks) and a backward sweep (to collect sources), which are performed in tandem. Those cells that fall $N+1$ steps from the latch form a set P of fixed peripheral nodes.[2]

In contrast to prior work that has assumed operation within a pre-buffering stage, our subcircuit selection algorithm must address the presence of buffers. These buffers will be encountered in our neighborhood selection algorithm, as they are part of the current logic; however, since it is presumed that they would be ripped up when new locations are determined (a critical assumption that makes our linear-delay model possible), we must prevent their inclusion in our model of the subcircuit. Therefore, when fetching adjacent gates, we transparently skip these buffers and omit them from the set M. The recursive functions TRUE-SOURCE() and TRUE-SINK() in Figure 8 provide this additional level of indirection, returning only those combinational gates that reflect the logical structure of the subcircuit.

[2]Variations on this theme, such as metrics that incorporate the degree of neighbors' criticality [14], [24] and the size of the subcircuit bounding box are also possible.

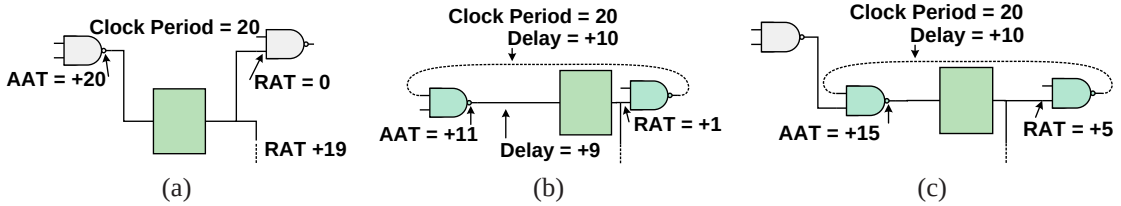

Fig. 7. Modeling feedback paths within logic requires a new type of gate. Pseudomovable gates have timing values that depend on the timing values of neighboring gates, but they cannot be moved. (a) Ignoring the presence of feedback paths is overly pessimistic, and it appears that the timing of the latch cannot meet its constraints. (b) Making the fixed gates along a feedback path pseudomovable allows the latch to meet its timing constraints, but doing only this can lead to the wrong placement. (c) Including all gates connected to pseudomovables as fixed timing points properly models the problem as a convex subcircuit.

```
BUILD-SUBCIRCUIT-FROM-SEED
    ▷ Input: Latch L , int N-hops
    ▷ Output: Set movables , Set pseudo , Set fixed
1   movables = BUILD-MOVABLES-FROM-SEED(L, N-hops)
2   pseudo = BUILD-PSEUDOMOVABLES-FROM-MOVABLES(movables)
3   fixed = BUILD-FIXED-FROM-CORE(movables ⋃ pseudo)

BUILD-MOVABLES-FROM-SEED
    ▷ Input: Latch L , int N-hops
    ▷ Output: Set movables
1   inputs = input-fringe = {L}
2   outputs = output-fringe = {L}
3   for i = 1 .. N-hops
4       input-fringe = ⋃ ( GET-INPUTS(input ∈ input-fringe) )
5       output-fringe = ⋃ ( GET-OUTPUTS(output ∈ output-fringe) )
6       inputs = inputs ⋃ input-fringe
7       outputs = outputs ⋃ output-fringe
8   movables = inputs ⋃ outputs

BUILD-PSEUDOMOVABLES-FROM-MOVABLES
    ▷ Input: Set movables
    ▷ Output: Set pseudo
1   pseudo = ∅
2   do
3       Set fan_in = INPUT-CONE(movables ⋃ pseudo)
4       Set fan_out = OUTPUT-CONE(movables ⋃ pseudo)
5       Set pseudo′ = ( fan_in ⋂ fan_out ) - movables - pseudo
6       pseudo = pseudo ⋃ pseudo′
7   while pseudo′ ≠ ∅

BUILD-FIXED-FROM-CORE
    ▷ Input: Set core
    ▷ Output: Set fixed
1   fixed = ∅
2   for each Gate G ∈ core
3       Set neighbors = GET-INPUTS(G) ⋃ GET-OUTPUTS(G)
4       fixed = fixed ⋃ (neighbors - core)

GET-INPUTS
    ▷ Input: Gate G
    ▷ Output: Set inputs
1   S = ∅
2   for each pin ∈ IN-PINS(G)
3       S = S ⋃ TRUE-SOURCE(pin)
4   return S

GET-OUTPUTS
    ▷ Input: Gate G
    ▷ Output: Set outputs
1   S = ∅
2   for each pin ∈ OUT-PINS(G)
3       S = S ⋃ TRUE-SINKS(pin)
4   return S

TRUE-SOURCE
    ▷ Input: Pin p
    ▷ Output: Gate source
1   Net net = NET(p)
2   Gate G = DRIVER(net)
3   unless IS-BUFFER(G)
4       return G
5   p = IN-PIN(G)
6   return TRUE-SOURCE(p)

TRUE-SINKS
    ▷ Input: Pin p
    ▷ Output: Set sinks
1   Net net = NET(p)
2   Set driven = DRIVEN(net)
3   S = ∅
4   for each Gate G ∈ driven
5       if IS-BUFFER(G)
6           p = OUT-PIN(G)
7           S′ = TRUE-SINKS(p)
8       else S′ = G
9       S = S ⋃ S′
10  return S
```

Fig. 8. Subcircuit selection transparently skips buffers when building a neighborhood of movable gates, and requires detection of "pseudomovables."

B. Feedback Paths

As noted in [24], the process of extracting gates to form a subcircuit may suffer from complications when subpaths of combinatorial logic between peripheral nodes are not modeled. These subpaths introduce additional timing constraints that, if left absent from the model, could invalidate the optimality of the solution.

To illustrate, consider the example in Figure 7, in which a single latch has been selected as a movable gate. After collecting its inputs and outputs, a simple subcircuit is constructed as shown in Figure 7(a), with the two endpoints shown selected as fixed gates. With the timing constraints as given in the figure, an optimal solution to this problem will place the latch equidistantly from both endpoints to ensure that the slacks on either side are balanced. However, consider a scenario where a feedback path exists from the output to the input, as shown in Figure 7(b); in such an event, the RAT of the output and the AAT of the input are *dependent* on the location of the latch. If this dependency is modeled, the solution may be biased toward one of the two neighbors. We loosely refer to these neighbors as *pseudomovable* gates. Although timing must be propagated through them (as it is for movable gates), their physical locations may be fixed.

Pseudomovables are collected by intersecting the transitive cones of logic between inputs and outputs to detect feedback paths, as shown in the pseudocode of Figure 8.[3] To ensure accuracy, the inputs and outputs of pseudomovables themselves must be bounded by fixed endpoints, as shown illustrated in Figure 7(c). These fringe nodes completely isolate the timing of the resulting *convex* subcircuit from outer cones of logic.

C. The "Do no harm" Philosophy

After gates are moved, it is likely that timing has degraded due to, for example, a capacitance violation on a long wire. The subcircuit must be examined, and its interconnect improved through physical synthesis optimizations, which might

[3]To improve runtime, one can limit the depth of these cones to a reasonably small constant, as opposed to the exhaustive expansion in [5].

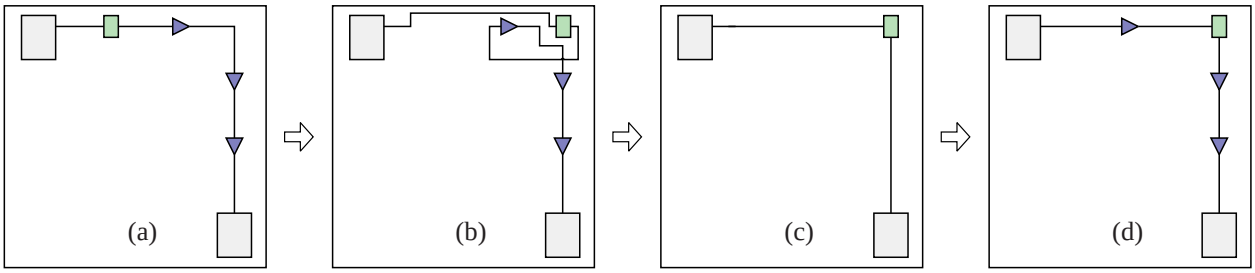

Fig. 9. The RUMBLE algorithm proceeds by (a) selecting a subcircuit to work on. An LP is formulated and solved, with movable gates being relocated as shown in (b). Existing repeater trees are no longer appropriate, and are subsequently removed in (c). Finally, the nets are re-buffered, forming the final subcircuit shown in (d).

include gate-sizing and buffer-insertion for delay or electrical considerations on nets.

Even though the linear program of Section IV-B can be solved optimally, it does not account for all the complexities of interconnect optimization. The linear program is an abstraction of the subcircuit timing that models physical synthesis optimizations (e.g., virtual-buffering) by prorating wire delay constants based on upcoming physical synthesis optimizations. Despite the high correlation to more accurate timing models in experimental results, the RUMBLE model could turn out to be too optimistic and its solution might result in a timing degradation. For example, nets can cross blockages or congested regions with no nearby legal locations. As a result, legalization could create a timing degradation.

When running RUMBLE in our physical synthesis flow, we mitigate the harmful effects of legalization by finding legal locations for gates and buffers when moving or inserting them. Insisting on legal locations can also contribute to a degradation not anticipated by the RUMBLE model. Fortunately, RUMBLE can examine the timing implications of its changes before committing to them. It simply stores the initial state of the subcircuit, and restores it if a timing degradation occurs. In this way, RUMBLE will "do no harm" to the circuit by ensuring that whatever solution it keeps is no worse than what existed before. Such safe delay optimizations are more easily inserted into physical synthesis flows [5], [20].

D. The RUMBLE Algorithm

Figure 10 shows pseudocode for the RUMBLE algorithm, which assumes a set of movable gates given at input, and Figure 9 illustrates the process. First, the subcircuit that is necessary for incremental placement is extracted (for a single movable, we extract its one-hop neighborhood of input gates). During this process, buffers are ignored (viewed as wires) as described in Section V-A. Next, RUMBLE performs timing analysis so as to measure timing improvement later. Line 3 stores the state of the circuit (gates and nets) so as to possibly undo most recent transformations we are considering. Once the initial state is safely stored, lines 4-6 use the linear program of Section IV to compute new gate locations, followed by buffer removal. If the model shows improvement we continue. Buffers are inserted on line 8, and other physical synthesis optimizations could also be applied here (e.g, repowering, V_{th} assignment, etc.). Lines 9-12 measure improvement, and in the case of timing degradation, restores the initial solution.

```
RUMBLE-ONE-LATCH
    ▷ Input: Gate movable
    ▷ Output: movable has optimized location and interconnect
 1  subcircuit = BUILD-SUBCIRCUIT-FROM-SEED(movable, 0)
 2  before-timing = MEASURE-TIMING(subcircuit)
 3  initial-solution = CACHE-SUBCIRCUIT(subcircuit)
 4  LP = new RUMBLE linear program for subcircuit
 5  after-locs = SOLVE(LP)
 6  SET-GATE-LOCATIONS(subcircuit, after-locs)
 7  REMOVE-BUFFERS(subcircuit)
 8  REINSERT-BUFFERS(subcircuit)
 9  after-timing = MEASURE-TIMING(subcircuit)
10  if(after-timing worse than before-timing)
11      RESTORE-GATE-LOCATIONS(subcircuit, initial-solution)
12      RESTORE-INTERCONNECT(initial-solution)
```

Fig. 10. The RUMBLE algorithm for moving one latch.

VI. EXPERIMENTAL RESULTS

RUMBLE is implemented in C++ (compiled with GCC 4.1.0) and integrated into an industrial physical synthesis flow. For our experiments, we examined an already optimized 130nm commercial ASIC with clock period 2.2ns and 3 million objects. We first examined the most critical latches and then filtered out the ones where the latch was already well placed. We use the algorithm from [2] to perform buffering after the cells have been moved. In practice, the LP-solving technique from RUMBLE requires only 17 milliseconds; the buffering algorithm dominates the runtime (over 75%). Since the overall runtime is dependent on the choice of the buffering algorithm we omit the (trivial) runtimes from our tables. Note that the "do no harm" approach of Section V-C is applied to all experiments, preventing timing degradation in our tables (i.e., a value of 0 appears in the imprv. column).

A. Re-buffering in RUMBLE

Previously published LP techniques for timing-driven placement do not allow for re-buffering during optimization. Instead, they are either applied before buffers have been inserted, or they do not differentiate the buffers from other gates. Our first experiment is designed to show how important it is to rip up buffers before replacing gates and subsequently rebuffering.

We modified our pseudocode in Figure 8 so that the function IS-BUFFER() always returns false. The effect of this is to stop "seeing through" the buffers, and instead to consider them fixed timing endpoints. This configuration models the work of [24]. We then calculate a new location for each latch with the

LP in Section IV. The final change is to skip line 8 of Figure 10, i.e., do not re-buffer. We call this algorithm KEEP-BUFFERS.

Table VI-A shows the results of RUMBLE on a single latch compared with KEEP-BUFFERS. Column 1 shows the name of the benchmark and columns 2 and 5 show worst-slacks in picoseconds before optimization. Columns 3 and 6 show the slacks after optimization of KEEP-BUFFERS and RUMBLE respectively. Columns 4 and 7 show the improvements of each technique.

Center-of-gravity vs. RUMBLE						
	COG Slack (ps)			RUMBLE Slack (ps)		
Subcircuit	orig	new	imprv.	orig	new	imprv.
latch A0	-1480	-527	**953**	-1480	26	**1506**
latch A1	-1268	-203	**1065**	-1268	186	**1454**
latch A2	-1020	-800	**219**	-1020	-791	**229**
latch A3	-953	-615	**338**	-953	-390	**563**
latch A4	-897	-78	**819**	-897	356	**1253**
latch A5	-848	-319	**529**	-848	-278	**570**
latch A6	-690	-690	**0**	-690	395	**1085**
latch A7	-645	-645	**0**	-645	-19	**626**
latch A8	-633	-633	**0**	-633	290	**923**
latch A9	-610	67	**677**	-610	262	**872**
avg	*-904*	*-444*	*460*	*-904*	*4*	*908*

TABLE I
KEEPING BUFFERS INSTEAD OF REMOVING AND REINSERTING THEM DEGRADES RUMBLE'S PERFORMANCE.

From the table we observe the following:

- Despite not ripping up buffers, KEEP-BUFFERS is still able to improve solution quality for nine out of ten testcases, though the improvement is never more than 220ps.
- When rip-up and re-buffering is allowed, RUMBLE is able to significantly outperform KEEP-BUFFERS for all ten testcases. On average the improvement grows by 7.4x.
- While KEEP-BUFFERS improves slack by an average of 123ps, RUMBLE improves slack by 908ps, which confirms how important it is to rip-up buffers so that they do not anchor the latch into an artificially small region.

B. Accuracy of the RUMBLE Timing Model

Theoretical results published by Otten [17] and discussed in Section III indicate that optimal buffer insertion on a 2-pin net results in a wire delay that is linearly-proportional to its length. The RUMBLE model heavily relies on these results.

Table VI-B compares the model-predicted values for subcircuit slack to values measured by running a commercial static timing analyzer. Measurements are taken after the RUMBLE LP is solved, the latches are moved and connected nets are buffered. Columns 2-4 report the initial, final, and improvement in worst-slack of the subcircuit measured by the timing model presented in Section III. Columns 5-7 report the same metrics measured by the STA engine.

We make the following observations:

- On average, the RUMBLE model overestimates the actual timing improvement by about 15%. This makes sense since it assumes an optimal ideal buffering will be achievable, but this is not always the case, especially for multi-sink nets.
- However, if one compares actual improvement to model improvement, there is a 97% correlation, suggesting that the model is reasonable enough to justify the latch location.

Model timing vs. reference timing						
	Model slack (ps)			Subcircuit slack (ps)		
Subcircuit	orig	new	imprv.	orig	new	imprv.
latch A0	-1799	-48	**1751**	-1480	26	**1506**
latch A1	-1509	65	**1574**	-1268	186	**1454**
latch A2	-1113	-868	**245**	-1020	-791	**229**
latch A3	-1147	-527	**620**	-953	-390	**563**
latch A4	-1090	180	**1269**	-897	356	**1253**
latch A5	-945	-295	**650**	-848	-278	**570**
latch A6	-920	320	**1241**	-690	395	**1085**
latch A7	-886	49	**935**	-645	-19	**626**
latch A8	-913	213	**1126**	-633	290	**923**
latch A9	-800	397	**1198**	-610	262	**872**
avg	*-1112*	*-51*	*1061*	*-904*	*4*	*908*

TABLE II
THE RUMBLE MODEL ACCURATELY PREDICTS THE SOLUTION QUALITY IMPROVEMENTS IN THE REFERENCE TIMING MODEL.

We now show how RUMBLE actually improves the design's timing characteristics.

C. RUMBLE on a Single Latch

Given that we are solving a new physical synthesis problem, existing solutions are scarce. Therefore we first consider straightforward approaches to solve this problem. One possibility is to take the *center-of-gravity* (COG) of adjacent pins. A timing-driven improvement of the center-of-gravity technique weights each pin by its slack. A reasonable version of this heuristic works in the following way. For a slack threshold T_s (see Section IV-D), let the weight w of a pin p with slack S_p be:

$$w_p = \begin{cases} 1 + |S_p - T_s| & S_p < 0 \\ \max(0.1, 1 - |S_p - T_s|) & S_p \geq 0 \end{cases}$$

Then we compute the x coordinate of movable gate m as the weighted average of the x coordinates of the set of neighboring pins P.

$$m_x = \frac{\sum_{p \in P} w_p p_x}{\sum_{p \in P} w_p}$$

and similarly for the y coordinate.

We implemented the above COG technique within the RUMBLE framework in place of the LP solver presented in Section IV. We still allow COG the benefits of ripping up buffers, and reinserting them after the latches are moved. Table VI-C shows a comparison between RUMBLE and slack-weighted COG on 10 latches. Column 1 shows the same latches as reported in Table VI-B. Columns 2-4 show the initial and final slacks, and improvement for COG. Columns 5-7 show the same for RUMBLE.

We observe the following:

- For all ten cases, RUMBLE generates a better solution than COG. For three of the cases, COG could not improve the

Iterated RUMBLE vs. RUMBLE: 1-hop												
	Iterated single-move RUMBLE						Multi-move RUMBLE					
	Slack (ps)			FOM (ps)			Slack (ps)			FOM (ps)		
Subcircuit	orig	new	imprv.	orig	new	imprv.	orig	new	imprv.	orig	new	imprv.
subcircuit B0	-1542	-1542	**0**	-6091	-6091	**0**	-1542	-130	**1412**	-6091	-130	**5962**
subcircuit B1	-1501	-277	**1223**	-5924	-277	**5647**	-1501	55	**1556**	-5924	0	**5924**
subcircuit B2	-1240	-1240	**0**	-4354	-4354	**0**	-1240	-980	**261**	-4354	-4044	**310**
subcircuit B3	-848	-278	**569**	-2523	-812	**1710**	-848	-279	**569**	-2523	-813	**1709**
subcircuit B4	-690	-79	**612**	-4090	-79	**4011**	-690	202	**893**	-4090	0	**4090**
subcircuit B5	-690	48	**739**	-2053	0	**2053**	-690	290	**980**	-2053	0	**2053**
subcircuit B6	-645	-18	**627**	-1921	-32	**1889**	-645	301	**945**	-1921	0	**1921**
subcircuit B7	-595	86	**681**	-1937	0	**1937**	-595	503	**1098**	-1937	0	**1937**
subcircuit B8	-444	-444	**0**	-889	-889	**0**	-444	-92	**352**	-889	-191	**698**
subcircuit B9	-418	-46	**372**	-857	-46	**811**	-418	6	**424**	-857	0	**857**
avg	*-861*	*-379*	*482*	*-3064*	*-1258*	*1806*	*-861*	*-12*	*849*	*-3064*	*-518*	*2546*

TABLE IV
RUMBLE SIMULTANEOUSLY MOVING A *one-hop* NEIGHBORHOOD COMPARED TO ITERATIVELY MOVING THE SAME GATES INDIVIDUALLY.

Iterated RUMBLE vs. RUMBLE: 2-hop												
	Iterated single-move RUMBLE						Multi-move RUMBLE					
	Slack (ps)			FOM (ps)			Slack (ps)			FOM (ps)		
Subcircuit	orig	new	imprv.	orig	new	imprv.	orig	new	imprv.	orig	new	imprv.
subcircuit C0	-719	-719	**0**	-8313	-8313	**0**	-719	-675	**44**	-8313	-5028	**3285**
subcircuit C1	-719	-719	**0**	-8004	-8004	**0**	-719	-653	**66**	-8004	-4386	**3617**
subcircuit C2	-690	-79	**612**	-4090	-79	**4011**	-690	314	**1004**	-4090	0	**4090**
subcircuit C3	-690	-79	**612**	-4090	-79	**4011**	-690	337	**1027**	-4090	0	**4090**
subcircuit C4	-681	-349	**333**	-3865	-349	**3516**	-681	-158	**524**	-3865	-158	**3707**
subcircuit C5	-645	-91	**554**	-3767	-306	**3462**	-645	371	**1015**	-3767	0	**3767**
subcircuit C6	-645	-33	**612**	-3767	-52	**3716**	-645	324	**969**	-3767	0	**3767**
subcircuit C7	-318	-318	**0**	-940	-940	**0**	-318	531	**848**	-940	0	**940**
subcircuit C8	-490	227	**716**	-966	0	**966**	-490	466	**956**	-966	0	**966**
subcircuit C9	-217	-217	**0**	-652	-652	**0**	-217	60	**277**	-652	0	**652**
avg	*-581*	*-238*	*344*	*-3846*	*-1877*	*1968*	*-581*	*92*	*673*	*-3846*	*-957*	*2888*

TABLE V
RUMBLE SIMULTANEOUSLY MOVING A *two-hop* NEIGHBORHOOD COMPARED TO ITERATIVELY MOVING THE SAME GATES INDIVIDUALLY.

Center-of-gravity vs. RUMBLE						
	COG			RUMBLE		
	Slack (ps)			Slack (ps)		
Subcircuit	orig	new	imprv.	orig	new	imprv.
latch A0	-1480	-527	**953**	-1480	26	**1506**
latch A1	-1268	-203	**1065**	-1268	186	**1454**
latch A2	-1020	-800	**219**	-1020	-791	**229**
latch A3	-953	-615	**338**	-953	-390	**563**
latch A4	-897	-78	**819**	-897	356	**1253**
latch A5	-848	-319	**529**	-848	-278	**570**
latch A6	-690	-690	**0**	-690	395	**1085**
latch A7	-645	-645	**0**	-645	-19	**626**
latch A8	-633	-633	**0**	-633	290	**923**
latch A9	-610	67	**677**	-610	262	**872**
avg	*-904*	*-444*	*460*	*-904*	*4*	*908*

TABLE III
COMPARISON OF RUMBLE'S LP TO A SLACK-WEIGHTED CENTER-OF-GRAVITY TECHNIQUE.

latch placement. These new solutions are rejected by the driver so as not to make the design worse.

- On average, COG improves slack by 20.9% of the 2.2ns cycle time, whereas RUMBLE improves slack by 41.3%. This shows that one must incorporate slack constraints on cells incident on the latch to achieve the most balanced solution.

D. Optimizing Multiple Gates Simultaneously

For this experiment, we show how an even better solution can be obtained when one allows cells close to the latch to move. We show the effectiveness of this technique on two sets of circuits.

- **One-hop** subcircuits include every gate (while ignoring buffers and inverters) incident to the latch of interest that shares an incident net with the latch. Typically this results in 4 or 5 gates being moved.
- **Two-hop** subcircuits in addition include all non-buffer and inverter cells incident to cells in the one-hop neighborhood. These subcircuits range from 11 to 18 movables with a mean of 14.8 movables.

We compare this technique to iterated single-move RUMBLE, where we pick each cell in the neighborhood and solve the LP for that particular cell, fix it, and then move to the next cell. The experiment is designed to show that multiple cells need to be optimized simultaneously to obtain the best results.

To measure the improvement one must now consider the slacks of all cells that may be moved, and the objective

becomes to improve the worst slack of the entire subcircuit. However, when one cannot improve the most critical path, the other paths may have room for improvement. We use FOM to measure the total improvement of all the slacks in the subcircuit.

Tables IV and V compare iterating RUMBLE over each gate one at a time versus RUMBLE moving multiple gates simultaneously. Columns 2-4 show the original and final slack, and the slack improvement for iterated single-move RUMBLE, while columns 5-7 show the corresponding FOM measurements for a zero-slack threshold. Columns 8-13 show the same measurements for multi-move RUMBLE. We make the following observations:

- Multi-move RUMBLE is clearly more effective than iterative RUMBLE both for one- and two-hop neighborhoods. In fact, for six out of ten one-hop subcircuits and for seven out of ten two-hop circuits, multi-move actually brought the FOM down to zero, meaning it fixed all the timing violations. Iterative single move was able to fix two and four respectively.
- On average, the worst-slack improvements were 849ps and 673ps respectively for one- and two-hop subcircuits. The diminished improvement for larger subcircuits is likely because we are including more nets, some of which cannot be improved as much as those connected to the imbalanced latch (Figure 6 has an example).
- Solving the LP takes 53ms for one-hop subcircuits and 325ms for two-hop subcircuits, on average.

		Imb.	Imb. FOM	Crit.	Crit. FOM	FOM
ckt1	old	102768	-21855	7912	-2798	-22448
	new	93736	-19400	7775	-2644	-20511
	diff	-9032	2455	-137	154	1937
ckt2	old	12151	-3080	3206	-1783	-19211
	new	11037	-2351	2997	-1667	-18170
	diff	-1114	271	-209	116	1041

TABLE VI
RUMBLE DEPLOYED IN A PHYSICAL DESIGN FLOW ON CIRCUITS THAT HAVE PIPELINE LATCH PLACEMENT PROBLEMS. CKT1 HAS 2.92M OBJECTS AND 629K LATCHES AND CKT2 HAS 4.74M OBJECTS AND 247K LATCHES. "OLD" REPORTS VALUES BEFORE RUMBLE "NEW" REPORTS RESULTS AFTER AND "DIFF" REPORTS THEIR DIFFERENCE. FOM IS REPORTED IN NANOSECONDS.

E. RUMBLE in a physical design flow

In the experiments presented so far, we have compared the effects of RUMBLE to those of other techniques on the most critical latches of the design. Due to the high runtime of buffering all of the nets in multi-move subcircuits, multi-move RUMBLE for every critical latch in the design is expensive. Consequently, in this subsection, we demonstrate the cumulative effect of single-move RUMBLE when deployed in our physical synthesis flow on *all latches with a critical pin*. Table VI-D shows two circuits that each contain a significant number of poorly placed latches. For each circuit, we report 5 statistics. An imbalanced latch is defined as one that has slack on the input pins that is greater than the slack threshold, T_s (see Section IV-D), and slack on the output pins that are lower than T_s, or vice versa. The Imb. column reports the number of imbalanced latches found in the design. Let the set of imbalanced latches be I, and for each latch l let $ws(l)$ be the worst slack of any pin on l. We define imbalance FOM to be

$$\sum_{l \in I} T_s - ws(l) \tag{18}$$

The Imb. FOM column reports this value. A critical latch is defined as one that has pins on both sides that are below T_s. The Crit. column reports the number of critical latches found in the design. Similarly to imbalance FOM, if C is the set of critical latches and for each latch c let $ws(c)$ be the worst slack of any pin on c, then we define the critical FOM to be

$$\sum_{c \in C} T_s - ws(c) \tag{19}$$

The Crit. FOM column reports this value.

Finally, the FOM column reports the FOM for the entire design. We make the following observations:

- RUMBLE reduces the number of imbalanced latches by 8.8% and 9.2% on ckt1 and ckt2, respectively.
- RUMBLE has a harder time optimizing the critical latches than the imbalanced ones.
- RUMBLE reduces circuit FOM by 8.6% and 5.4% on ckt1 and ckt2, respectively.
- RUMBLE improves the characteristics of all columns, and does no harm to the circuit metrics.

In addition to these observations, we point out that the two most common reasons for being unable to fix a particular latch are 1) there is a high-fanout net in the subcircuit, which would degrade the performance of buffering, and we therefore skip this case or 2) the gates are moved to a fixed endpoint, which indicates that RUMBLE does not have enough freedom to solve the problem entirely. The addition of RUMBLE to our design flow adds about 4% to the total runtime in these experiments.

VII. CONCLUSIONS

In this work we observe that wirelength-driven placement leads to particularly poor timing of "pipeline latches" in modern physical design flows, which is especially problematic at sub-130nm technology nodes. To address this challenge, we developed RUMBLE — a linear-programming based, incremental physical synthesis algorithm that incorporates timing-driven placement and buffering. The latter justifies RUMBLE's linear-delay model which exhibited a 97% correlation to the reference timing model in our experiments. Empirically this delay model is accurate enough to guide optimization; RUMBLE improves slack by 41.3% of cycle time on average for a large commercial ASIC design.

The linear program (LP) used in RUMBLE is general enough to optimize multiple gates and latches simultaneously. However, when moving multiple gates considering only the slack objective, we encountered two challenges: placement stability and FOM degradations. We present our extensions to address these problems directly in our LP objective. With these additions, moving several gates simultaneously improves upon RUMBLE used iteratively on the same movables.

REFERENCES

[1] C. J. Alpert, C. Chu, and P. G. Villarrubia, "The Coming of Age of Physical Synthesis," *ICCAD*, 2007, pp. 246-249.

[2] C. J. Alpert et al., "Fast and Flexible Buffer Trees that Navigate the Physical Layout Environment," *DAC*, 2004, pp. 24-29.

[3] C. J. Alpert et al., "Accurate Estimation of Global Buffer Delay Within a Floorplan," *TCAD* 25(6), 2006, pp. 1140-1146.

[4] C. J. Alpert, et al., "Techniques for Fast Physical Synthesis," *Proc. IEEE* 95(3), 2007, pp. 573-599.

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

[6] A. Chowdhary et al., "How Accurately Can We Model Timing In A Placement Engine?," *DAC*, 2005, pp. 801-806.

[7] J. Cong, L. He, C.-K. Koh and P. H. Madden, "Performance Optimization of VLSI Interconnect Layout," *Integration: the VLSI Journal*, 1996, vol. 21, pp. 1-94.

[8] P. Cocchini, "Concurrent Flip-Flop and Repeater Insertion for High Performance Integrated Circuits," *ICCAD*, 2002, pp. 268-273.

[9] B. Halpin, C. Y. R. Chen and N. Sehgal, "Timing Driven Placement Using Physical Net Constraints," *DAC*, 2001, pp. 780-783.

[10] International Technology Roadmap for Semiconductors (2001). [Online]. Available: http://public.itrs.net.

[11] M. A. B. Jackson and E. S. Kuh, "Performance-driven Placement of Cell Based IC's," *DAC*, 1989, pp. 370-375.

[12] A. B. Kahng and I. L. Markov, "Min-max Placement for Large-scale Timing Optimization," *ISPD*, 2002, pp. 143-148.

[13] T. T. Kong, "A Novel Net Weighting Algorithm for Timing-driven Placement," *ICCAD*, 2002, pp. 172-176.

[14] T. Luo, D. Newmark and D. Z. Pan, "A New LP Based Incremental Timing Driven Placement for High Performance Designs," *DAC*, 2006, pp. 1115-1120.

[15] M. Marek-Sadowska and S. P. Lin, "Timing driven placement," in *ICCAD*, 1989, pp. 94-97.

[16] R. Nair, C. Berman, P. Hauge and E. Yoffa, "Generation of Performance Constraints for Layout," *TCAD* 8(8), 1989, pp. 860-874.

[17] R. Otten, "Global Wires Harmful?," *ISPD*, 1998, pp. 104-109.

[18] S. L. Ou and M. Pedram, "Timing-driven Placement Based on Partitioning with Dynamic Cut-net Control," *DAC*, 2000, pp. 472-476.

[19] D. A. Papa et al., "RUMBLE: An Incremental, Timing-driven, Physical-synthesis Optimization Algorithm," *ISPD*, 2008, pp. 2-9.

[20] H. Ren et al., "Hippocrates: First-Do-No-Harm Detailed Placement" *ASPDAC*, 2007, pp. 141-146.

[21] S. Sapatnekar, "Timing," Springer-Verlag, New York, 2004.

[22] P. Saxena, N. Menezes, P. Cocchini and D. A. Kirkpatrick, "Repeater Scaling and Its Impact on CAD," *TCAD* 23(4), 2004, pp. 451-463.

[23] L. Trevillyan et al., "An Integrated Environment for Technology Closure of Deep-submicron IC Designs," *IEEE Des. Test Comput.*, 2004, vol. 21, no. 1, pp. 14-22.

[24] Q. Wang, J. Lillis and S. Sanyal, "An LP-Based Methodology for Improved Timing-Driven Placement," *ASPDAC*, 2005, pp. 1139-1143.