Speeding Up Physical Synthesis with Transactional Timing Analysis

David Papa
University of Michigan, Ann Arbor, and IBM Austin Research Lab

Igor L. Markov
University of Michigan, Ann Arbor

Michael D. Moffitt and Charles J. Alpert
IBM Austin Research Lab

Modern physical-synthesis flows operate on very large designs and perform increasingly aggressive timing optimizations. Traditional incremental timing analysis now represents the single greatest bottleneck in such optimizations and lacks the features necessary to support them efficiently. This article describes a paradigm of transactional timing analysis, which, together with incremental updates, offers an efficient, nested `undo` functionality that avoids significant timing calculations.

Achieving timing closure for large modern ASIC designs requires the use of physical synthesis—a series of performance-driven optimizations that simultaneously alter the layout, the netlist, and the electrical parameters of logic gates. Physical synthesis tightly couples analysis with optimization in an automated flow that iteratively improves design parameters. Such flows rely on *static timing analysis* (STA) in two essential ways. First, STA identifies the sections of the design that are most critical to overall performance. Second, STA assesses the impact of every potential change on circuit performance before the change is committed. Circuit optimizations are bundled into transforms that implement common operations such as relocating a gate and buffering a net.[1] State-of-the-art physical design tools use compound transforms that simultaneously perform many simpler transforms that would not have improved overall performance if applied individually.[2]

Advanced technology nodes require complex timing models that cannot be captured analytically with sufficient accuracy, often making timing analysis the single major bottleneck in physical synthesis. Therefore, we take a closer look at the conceptual role of STA and its interfaces with optimization. Mathematically, circuit optimizations often interact with STA by obtaining arrival times and required arrival times at timing points throughout the design.[3] However, running STA on the entire design to evaluate each potential change is impractical. Therefore, STA can be used

- in batch mode to evaluate the compound impact of many changes;
- in incremental mode, where the impact of a single change is efficiently propagated through the netlist; and
- with lazy updates, where timing data are propagated only in response to queries, essentially batching the changes that occur between queries.

Multiobjective optimizations now increasingly rely on do-no-harm methodologies that carefully evaluate each change and commit only those that provide tangible improvements.[4-6] The more aggressive algorithms have very high rejection rates in this loop, making the speed of incremental STA a major factor in improving physical synthesis. However, batched mode and lazy updates are of limited use when evaluating the individual impact of multiple candidate changes.

 0740-7475/10/$26.00 © 2010 IEEE

The major impact of STA on overall runtime tempts physical-synthesis developers to assume the responsibility for some aspects of timing analysis and to use handcrafted local-delay models instead of STA engines, which offer significant opportunities for runtime improvement. However, this practice risks subtle timing mistakes and also increases the development effort by lowering reuse. Therefore, we propose improvements to reusable STA engines that better account for the bounded scope of physical-synthesis transforms.

We present an extension to the interface of STA to accommodate transactioning. Our technique employs a timing-change-history data structure that stores changes to the state of the timing graph so that it can be efficiently restored to a previous state in the event of a retraction. This experimental approach has been specifically designed to allow nesting events that spur timing changes. To further improve worst-case complexity, we have limited changes to the timing graph by way of bounded timing analysis, an enhancement that works in conjunction with transactional timing analysis to allow for the rapid exploration of circuit search space. Finally, we provide an empirical evaluation of bounded transactioning for both classical and lazy STA, demonstrating an improvement in performance by up to two orders of magnitude.

Background

Timing analysis and its integration into the physical-design flow have long been key topics in design automation. Modern STA engines are products of sophisticated engineering, and they have evolved substantially over recent decades. Yet, dramatic changes to basic timing models continue to drive the need for further innovation. For instance, multimode timing—wherein several timing points are maintained at each node of the global-timing graph, each corresponding to a different corner of design operation—has become increasingly popular. Although these corners enable modern optimization techniques to evaluate the effect of their actions on many scenarios at once, they also serve as a multiplier of basic computations that the timing engine must perform. Statistical STA engines that reflect the variance of design performance require the maintenance of complex distribution models that also significantly expand the amount of work placed on the timing engine. These elaborate models, in conjunction with a stronger emphasis on local transform-driven operations, have increased the responsibility of timing engines to provide a much higher degree of incremental maintenance of internal timing state.

Previous work

Drumm et al. explored the problem of updating only a subset of timing-analysis values in response to a local change.[7] In this approach, a depth-first propagation of timing values is executed until no change is observed. To reduce the amount of incremental recalculation needed, this process was later refined.[8] Lee and Tang presented a distinction between the propagation cost of positive and negative delay changes,[9] demonstrating that the expense of executing an operation is distinct from the execution of its inverse. The algorithm, described by Sapatnekar,[10] avoids excessive computation by propagating only along paths that are influenced by altered inputs. Mondal and Mandal proposed a query language based on temporal logic,[11] along with an algorithm to efficiently retrieve answers to those queries. Das et al. explored the application of STA for coupling and exploited circuit structure to determine an effective node ordering during incremental iterative analysis.[12]

Relatively little attention has been given to the explicit support for the retraction of local design changes, though some exceptions exist. Recent work by Kazda et al.,[13] for instance, provides support for transactional operations such as `begin`, `commit`, and `undo`. However, these operations are restricted only to the restoration of previously cached routing data, and are not communicated to the timing engine. Indeed, the decision to revert one or more timing properties to their original state is typically cast as just another sequence of incremental changes to the system; this forces the wasteful recomputation of timing data, which may be exacerbated due to the difference in expense between executing an operation and executing its inverse.[9] Other choices in the design flow—such as the decision to compute Steiner trees for delay estimation—also compound the effort required to restore timing information to a previously known state. The savings that can be achieved by efficiently rolling back recent changes are likely to escalate in coming years, as compound transforms become increasingly important in physical synthesis and routinely thrash the timer with multiple hypothetical changes.

Incremental STA

Early STA engines always processed an entire design, which is impractically expensive for evaluating optimization transforms. This expense can be avoided by using stale timing information or crude estimations, neither of which are acceptable in modern high-precision physical synthesis.[5] Another alternative is to maintain accurate timing information throughout the automated flow, but to do so in an incremental fashion. Research in incremental STA aims to provide efficient techniques for updating values within a timing network in response to local and partial modifications. Several varieties of incremental STA have appeared over the past decade, and they are responsible for decreasing timing runtime from hours to minutes.[14] Further extensions to incremental analysis include level- and dominance-limited schemes to reduce the amount of work performed.[15] *Lazy evaluation*,[8] in which propagation is delayed until triggered by a relevant query, represents a particularly important improvement in throughput from STA engines.

The boost in throughput offered by incremental analysis allows an optimization algorithm (as well as a designer) to explore several hypothetical (or what-if) scenarios, a task unaffordable in earlier tools.[14] Such hypothetical scenarios are typically communicated to the timing engine as if committing changes. If the results are unacceptable and the scenarios are rejected, another set of changes must be committed. This requires new timing calculations, even though the needed timing values have previously been known. Although a single layer of what-if support can be added to STA easily, it is insufficient for handling the evaluation of multiple nested scenarios and their retraction, which we will discuss in this article with detailed use cases.

Transform-driven optimization

Timing optimization during physical synthesis is typically accomplished, as Figure 1 illustrates, by gradually modifying and refining an initial netlist and placement image.[16] It is useful to distinguish between *drivers* and *transforms*; a driver selects (and orders) a sequence of nets or gates to refine, whereas a transform applies a given local optimization to the objects specified by the driver. For instance, IBM's Placement Driven Synthesis tool makes use of several transform templates, including buffering, connection reordering, and cloning (in addition to traditional techniques such as movement and repowering).[1]

The ultimate goal of timing-driven placement and synthesis is to obtain a set of nonoverlapping locations for all cells such that the performance of the design meets objectives. Not surprisingly, each transform typically makes several queries to the timer, not only to construct a basic model of the neighboring region (with appropriate arrival times and delays), but also to issue a query after optimization is complete to verify that an improvement in timing has been realized. Incremental propagation and lazy evaluation are well-known techniques that avoid work in some common use cases, thus saving considerable runtime. However, more sophisticated optimization techniques highlight use cases in which the timer performs unnecessary computations in the runtime bottleneck.

Table 1 summarizes several notable use cases. These cases are not meant to be mutually exclusive, but rather represent common strategies employed by physical-synthesis transforms. In this article, we focus on the use of retraction by these strategies.

Case 1: Fallible transforms

The simplest transforms rely exclusively on the timer to analyze the quality of their results. For example, a movement transform may blindly attempt several nearby locations for a critical latch, or a repowering transform may bind a critical gate to every possible power level. In either case, a timing query must be executed for each solution, so that the transform may keep the location (or power level) that exhibits the best slack. We refer to such techniques as *bind-and-test* transforms.

Alternatively, some transforms may attempt to predict the impact of their changes in advance, and then use the timer only to verify improvement. In the case of a repowering transform, the slew of input pins and the capacitance of output nets and sink pins (all known quantities) are enough information to determine a rough estimate of the slack on the output pin for each power level. Such a transform could compute the best power level, bind it, then verify that the new slack is better than the old one. Clearly, an approximation-based repowering transform may calculate a new power level that is inferior to the existing one, requiring it to restore the original power level to prevent degrading the circuit performance. In other words, such transforms are frequently fallible (i.e., prone to failure), and as a result, `undo` is often needed for error correction.

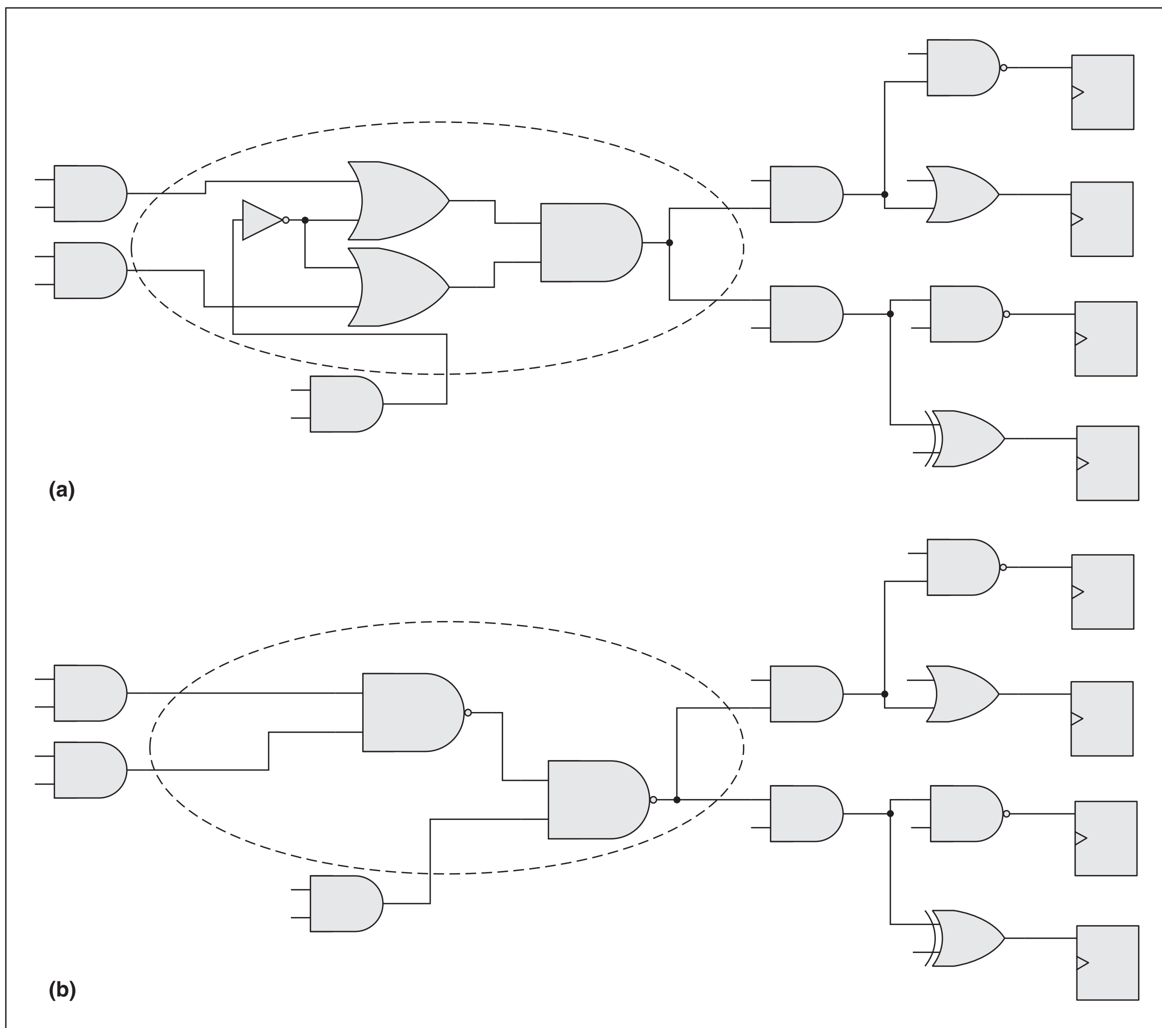

Figure 1. A physical synthesis transform improves the subcircuit (a) by resynthesizing the logic, resulting in the circuit shown in (b). The traditional way of evaluating the timing impact of such transforms can be improved considerably.

Table 1. Types of transforms with embedded retraction, the reasons they need retraction, and in what situations a retraction is applied. Representative values of `undo` frequency shown here are the expected number of retractions per transform application, and illustrate order-of-magnitude differences between successive cases.

Transform type	`Undo` purpose	`Undo` context	`Undo` frequency per transform application
Fallible	Error correction	Upon degradation	0.1
Candidate	Measure independent changes	For each candidate	1
Compound	Model nested changes	Search backtrack	10

Though simple, both fallible and bind-and-test transforms are inherently slow because of the propagation required to accommodate their repeated changes and timing queries. In the example of our bind-and-test repowering algorithm, each power level triggers timing updates for the fan-in and fan-out cones of the gate. Admittedly, much of this propagation could be deferred if the timing

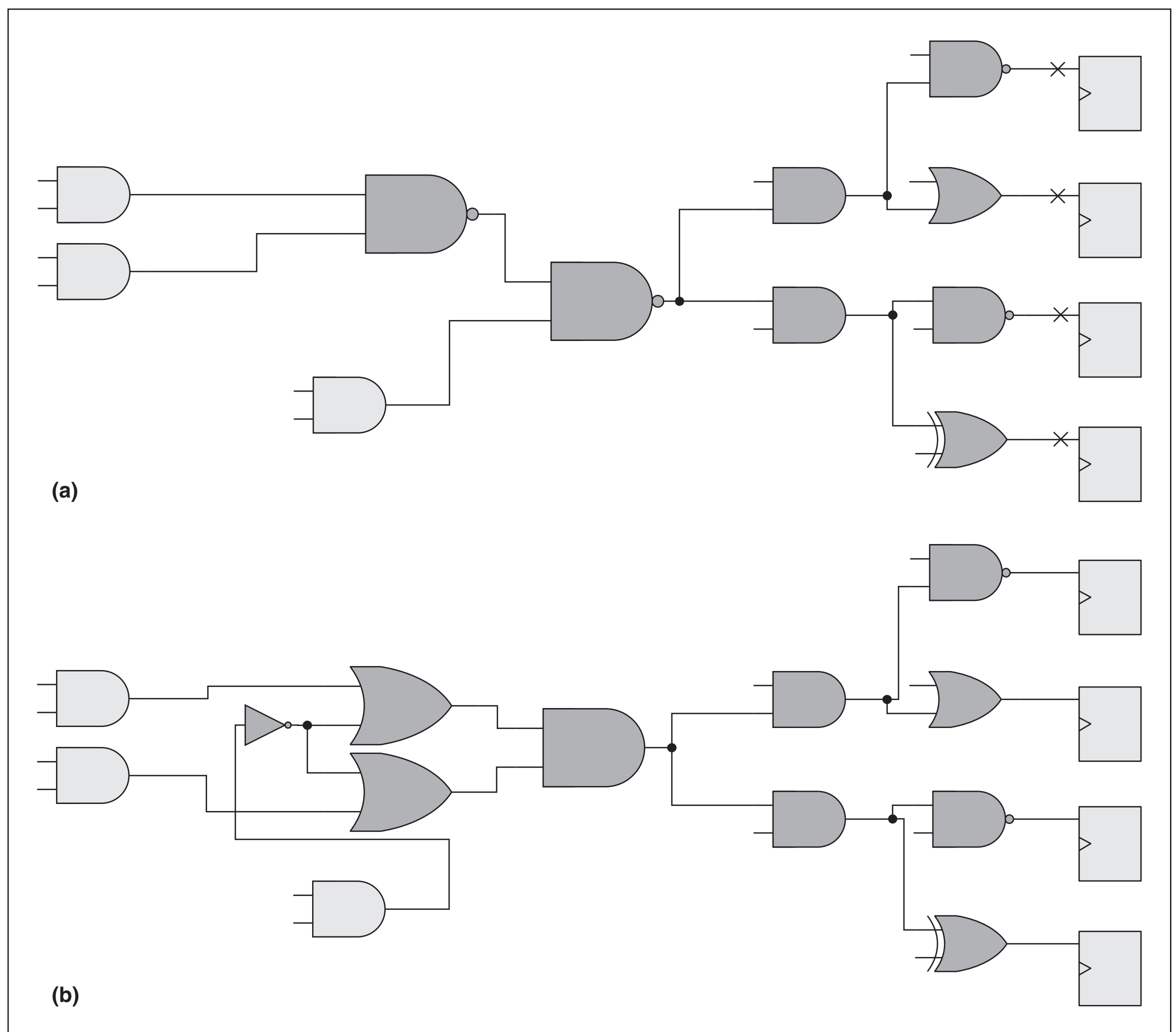

Figure 2. Evaluating the timing impact of the physical-synthesis transform in Figure 1b. Traditional STA with lazy evaluation will mark the fan-out cone of the change "dirty" (a). If the change is found to have a negative impact on timing, it will be reversed (b). This reversion will be treated as another change, and the fan-out cone will be marked "dirty" for a second time.

engine adopts a philosophy of lazy evaluation, as do many STA engines. In such a system, the arrival times of the nodes in the fan-out cone would remain uncomputed, but would instead be marked as "dirty" to indicate their staleness. Timing propagation would be invoked only in the event of a query (and even then, only to the portion of logic needed to answer the query). However, if the original location (or power level) is optimal—as it is likely to be if detailed placement has done its job properly—the demarcation of these cones as dirty is unnecessary because the original arrival times stored within these cones are in fact a correct representation of the current state. Figure 2 illustrates the amount of work performed by traditional STA with lazy evaluation when a circuit transform is retracted.

Case 2: Candidate selection transforms

Candidate selection transforms are those that employ multiple strategies to generate several alternatives, or *candidate solutions*, for a given gate. In so doing, they run each optimization and select the best candidate. Such transforms leverage the fact that different strategies work well in different contexts. For example, consider a transform that generates candidates by repowering as well as moving a gate. Typically, moving a gate can have a higher impact, but if the design has too little white space, there may be no open location where the gate can move

to improve timing. Instead, a higher power level may be available for the same footprint, or white space may be available nearby to increase the footprint.

Although fallible transforms may occasionally encounter `undo` for correction (e.g., when they degrade circuit performance due to approximation inaccuracy), a candidate selection transform requires `undo` by construction; after each candidate is computed, the initial state must be restored so that the next candidate can be generated independently on the basis of the initial conditions. In the example of repowering or moving a gate, retraction must restore the gate to its original power level after repowering so that the movement decision can be based on the timing of the initial power level. Timing queries for interrogating initial conditions of each candidate generation strategy can avoid the unnecessary work of timing updates if `undo` can restore the initial timing state.

Case 3: Compound transforms

Compound transforms not only consider multiple strategies for generating candidates but also do so for multiple objects. Such transforms may even consider composing optimizations to generate a single candidate. For example, they may consider simultaneously moving and/or repowering two connected gates in a discrete domain.[4,2] In this situation, there is a combinatorial number of solutions to evaluate, where each successive decision may depend on the previous.

Compound transforms stress timing analysis tools much more heavily than other use cases, in that the construction of a local model requires the search of a large, conditional solution space. Modifications are typically made in nested sequences to generate appropriate timing arcs; indeed, Moffitt et al. observed that the expense of generating their disjunctive timing graph is often more costly than the branch-and-bound search used to solve it optimally[2]—a consequence of the propagation incurred by the timer. When `undo` can efficiently restore the previous timing state, combinatorially many timing updates can be saved in compound transforms.

Transactional timing analysis

In the presence of retractions, state-of-the-art timing analysis performs a large amount of unnecessary work, as we have argued thus far. We now take a look at the details of bounded transactional timing analysis, which serves to substantially reduce the computation needed to support `undo`. We consider its application to both classical STA and the more popular version that supports lazy evaluation.

Incorporation of transactions

By definition, a retraction restores the design to a previously known state. Current techniques (which view retraction as nothing other than a separate incremental change) discard the original timing values during propagation. In transactional timing analysis, the key is to cache all timing data that becomes invalidated during the execution of a change.

Specifically, when a modification is made to the design, the timer is notified through a monitoring mechanism that the delay at a particular timing point has changed. That notification triggers a corresponding propagation to the transitive fan-in and fan-out cones. During transactional-timing-analysis propagation, prior values are not simply overwritten (as is commonly done within STA engines), but are rather stored in a change stack as new values are written in their place. Therefore, if and when change is retracted, the old values may be restored by replaying the timing updates in reverse.

If a sequence of nested transactions are executed (as may occur with compound transforms), each individual change stack serves as a distinct checkpoint of the design state. These checkpoints are themselves stored on a transaction stack of all change stacks. A new change stack is pushed onto the transaction stack when a transform requests a new checkpoint. The current state of timing is stored in the timing graph as usual. When a transform backtracks and retracts its circuit modifications, changes to the timing graph may be rolled back to the most recent checkpoint by copying all values in the current head of the transaction stack back into the timing graph. Changes may be committed simply by clearing the transaction stack.

Figure 3 shows one possible implementation of transactional timing analysis. We stress that many variations on this code are possible; for instance, if a change is likely to have significant impact on the state of the design, the caching of old timing values could be performed once prior to (rather than during) propagation. To facilitate integration, a more complex interface must exist between transforms and timing engines. Transforms are required to communicate their intent (e.g., whether their modification imposes a new condition or, instead, reflects a

```
Changed-Delay
       > Input: arc → timing arc that changed
1.     Propagate-Forward(arc.input)
2.     Propagate-Forward(arc.input)

Undo-Changes
1.     AATStack = ChangeHistory.top().AATStack
2.     WHILE NOT AATStack.empty()
3.        AATStack.top().node.aat = AATStack.top().aat
4.        AATStack.pop()
5.     RATStack = ChangeHistory.top().RATStack
6.     WHILE NOT RATStack.empty()
7.        RATStack.top().node.aat = RATStack.top().aat
8.        RATStack.pop()
9.     ChangeHistory.pop()

Commit-Changes
       > Existing changes no longer need to be tracked
1.     ChangeHistory.clear()

Propagate-Forward
       > Input: timing-point
1.     FOREACH successor succ of timing-point
2.        IF Update-AAT (timing-point, succ)
3.          Propagate-Forward (succ)

Push-Changes
1.     ChangeHistory.push_new_layer()

Update-AAT
       > Input: pred, succ timing points
1.     delay = compute-delay(pred, succ)
2.     IF (succ.aat < pred.aat + delay)
3.        ChangeHistory.top().AATStack.push(TC(succ,
               succ.aat))
4.        succ.aat = pred.aat + delay
5.        RETURN TRUE
6.     RETURN FALSE

Propagate-Backward
       > Input: timing-point
1.     FOREACH predecessor pred of timing-point
2.        IF Update-RAT(pred, timing-point)
3.          Propagate-Backward(pred)

Update-RAT
       > Input: pred, succ timing points
1.     delay = compute-delay(pred, succ)
2.     IF (pred.rat > succ.rat - delay)
3.        ChangeHistory.top().RATStack.push(TC(pred,
               pred.rat))
4.        pred.rat = succ.rat - delay
5.        RETURN TRUE
6.     RETURN FALSE
```

Figure 3. One possible implementation of transactional timing analysis. The functions `Propagate-Forward` and `Propagate-Backward`, shown here using recursion for brevity, are best implemented without recursion. `AAT` and `RAT` refer to actual arrival time and required arrival time respectively.

restoration), so that the appropriate action is taken on behalf of the timing engine.

Extensions

As noted earlier, it is common for STA engines to defer timing updates until needed by a relevant timing query. In many cases, this avoids work when timing values are invalidated multiple times before they are actually used. The notification of a change in delay during such lazy execution will not trigger timing propagation. Instead, the fan-in and fan-out cones of a modified edge are simply marked dirty, indicating that they must be recomputed.

To accommodate transactional timing analysis with lazy execution, dirty bits must also be considered as part of a timing point's state. In the event of a retraction, traditional STA engines allow the affected nodes to remain dirty, whereas bounded timing analysis will revert them back to their state prior to the change. Though not shown in Figure 3, the extension is relatively straightforward: all actions that alter the dirty bit of a timing point are recorded and are subsequently restored if the transform issues a retraction.

Finally, support for transactioning in the presence of logic changes (such as the pin swapping of our original example) requires careful caching of topological modifications to the graph itself (in addition to the timing values associated with these elements). The creation, deletion, and modification of graph connectivity can be achieved through a reference labeling of timing points; changes to structural elements, such as edges and nodes, are recorded with respect to these unique identifiers and thus may subsequently be restored. Although the implementation required to properly maintain this bookkeeping is complex and nuanced, it introduces no substantial intellectual novelties beyond the original framework.

Bounded timing windows

When evaluating the impact of a transform, it is common to query timing at specific relative locations to the change. For example, we can query the slack of a gate's output pin after repowering, or the slack of an input pin at the next circuit level after moving a gate. When it is known beforehand how far the scope of a change extends, we can limit timing analysis (and, for our application, the amount of data stored per transaction) to be correct only within that range. We call this local region a

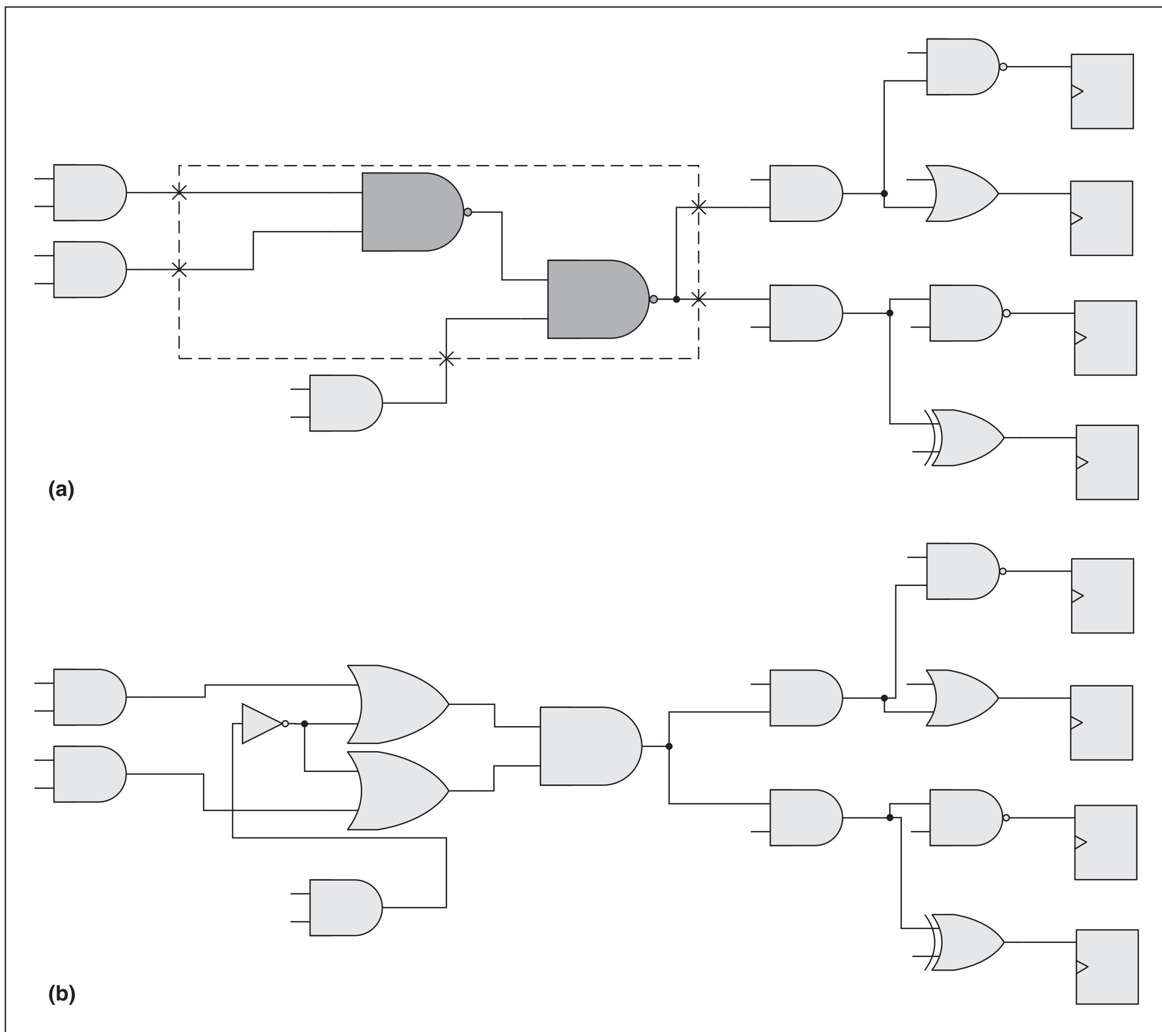

Figure 4. Evaluating the timing impact of the physical-synthesis transform in Figure 1b. Bounded transactional timing analysis will not propagate the change outside of a specified window (a). In the event of a reversion, gates with dirty timing will have their timing data restored (b).

bounded timing window. (Some STA engines—such as IBM's EinsTimer—provide similar level-limiting features that serve to circumscribe the scope of local changes; they are not, however, integrated with any form of transaction management.) Limiting propagation to such windows provides runtime savings, as it is only necessary to propagate arrival times (and/or dirty bits, in the case of lazy evaluation) to the window boundaries. Likewise, in the event of a rollback, the data required to restore the graph to its original state is also reduced. Since immediate timing queries are assumed to be made within the timing window, all values outside the region are considered to be fixed timing endpoints. Figure 4 shows bounded transactional timing analysis for an example transform.

Selecting an appropriate window size for a particular transform can require some care. The effect on timing of an optimization depends on the nature of the optimization; therefore, choosing a static-window size is best done when the transform is designed and tested. In particular, differences in slew can greatly affect timing for the whole path in ways that are difficult to predict while considering only slack.[17] For this reason, timing-analysis tools support a mode to limit slew propagation to a constant number of levels. This mode provides a convenient way to limit the scope of timing changes and improves the speed of timing analysis in physical-synthesis tools. Any window larger than the scope of slew propagation can provide faster queries with no accuracy loss. Furthermore, in the context of bounded transactional

timing analysis, timing queries are required only to decide if a retraction is necessary. Typically, the effect of an optimization on a path's timing is known with enough accuracy to make a decision whether or not to retract after a signal is propagated through only a few levels of logic. An additional dynamic approach runs a few trial transform applications and samples several window sizes to determine how much accuracy is lost for various window sizes. Such an approach then chooses the smallest window size with tolerable error to be used on the majority of transform applications.

Facilitating parallelism

Because a bounded timing window delimits the scope of a local change, it also provides a guarantee on the mutual independence of disjoint timing islands. This independence meets the requirements set forth for distributed STA,[18] and it could, in theory, be exploited to easily decompose timing optimization into several parallel processes.

Although we did not evaluate such a parallel architecture in our experimental setup, we stress that significant computational savings could be gained if properly implemented and integrated with other components of the automated flow (e.g., the placement engine, the data model, etc.).

Empirical results

To evaluate the computational benefits of bounded transactional timing analysis, we implemented the aforementioned techniques in a new STA tool that supports both classical STA (i.e., the academic variety that immediately performs propagation of modified timing values) and lazy evaluation (e.g., the more popular variety that performs propagation only on demand). For evaluation of the former, we discounted the runtime required for initial propagation of a change, as that time is shared by "with-transaction" and "without-transaction" runs. All incarnations of our timing engine employ some form of incremental propagation.

We modified a simple timing-driven gate movement transformation within a state-of-the-art industrial physical-synthesis flow to query our static-timing analyzer when deciding whether or not to retract the change. Changes to delay values in the timing graph of an actual 65-nm design were simulated and profiled to determine the runtime incurred by STA. Two parameters were adjusted in these experiments: the probability that a delay change is retracted (P(`undo`)) and the size of our bounded timing window (where a size of ∞ indicates the absence of this technique). Because the frequency of finding timing-driven placement improvements strongly depends on the circuit and the state of optimization, our experimental transform used the P(`undo`) parameter to determine if and when to retract the change. Thus, we could vary P(`undo`) independently to study the impact on runtime of any frequency of retraction.

Table 2 presents the results of these tests. As would be expected, no benefit is observed when retractions are never performed; in fact, the overhead involved in recording transaction state puts transactioning at a slight disadvantage. However, even with a moderate amount of `undo`, the computational savings can be substantial. For classical STA, an improvement in retraction speed of over 200× is observed. For lazy evaluation, a factor of up to 5.2× is achieved, confirming that although lazy evaluation alone does indeed prevent a fair amount of thrashing, it can be further improved by transactioning.

It can also be observed that bounded timing windows (which can be exploited independently of transactioning) are generally effective at reducing runtime. We expect that most automated flows should be able to make use of the combined benefits of lazy evaluation, transactioning, and bounded timing windows.

IN THE SYSTEM we have described, our work was motivated primarily by deficiencies in STA that cause it to behave poorly for a wide range of physical-synthesis operations. The incremental-timing concepts we have presented are not unique to physical synthesis; they are equally applicable to the efficient support of logic synthesis transforms, and some of them may have been in use for this purpose since the mid-1990s. However, conventional logic synthesis does not stress timing infrastructure as much as modern physical synthesis does; therefore, relevant techniques were not given as much attention in timing analysis literature—and, to this day, remain poorly documented. As transform-driven optimizations in physical synthesis continue to increase in complexity, the need to efficiently accommodate hypothetical timing queries is likely to grow. Our future work pertains to researching and developing new compound transformations that leverage the timing analysis techniques we have described in this article. ■

Table 2. Empirical results of bounded transactional timing analysis with classical STA and with lazy STA.

		Nodes expanded				Runtime (seconds)			
		Classical STA		Lazy STA		Classical STA		Lazy STA	
P(undo)	Window size	Without trans.*	With trans.	Without trans.	With trans.	Without trans.	With trans.	Without trans.	With trans.
0%	∞	12,638	12,638	22,905	22,905	0.28	0.33 (0.8x)	2.09	2.36 (0.8x)
	40	12,638	12,638	19,356	19,356	0.32	0.33 (0.9x)	1.65	1.86 (0.8x)
	20	10,186	10,186	7,400	7,400	0.25	0.26 (0.9x)	0.41	0.47 (0.8x)
	10	3,641	3,641	1,967	1,967	0.08	0.08 (1.0x)	0.10	0.11 (0.9x)
10%	∞	346,170	12,646	22,895	22,605	14.5	0.32 (45.3x)	2.07	2.29 (0.9x)
	40	202,821	12,646	19,346	19,056	6.65	0.32 (20.7x)	1.69	1.82 (0.9x)
	20	41,251	10,194	7,380	7,013	1.3	0.25 (5.2x)	0.41	0.43 (0.9x)
	10	5,957	3,649	1,955	1,793	0.14	0.07 (2.0x)	0.09	0.10 (0.9x)
30%	∞	1,124,067	12,693	22,888	21,960	46.66	0.32 (145.8x)	1.98	2.28 (0.8x)
	40	510,642	12,693	19,339	18,320	14.84	0.32 (46.3x)	1.63	1.76 (0.9x)
	20	75,128	10,233	7,353	6,282	2.13	0.25 (8.5x)	0.39	0.37 (1.0x)
	10	8,716	3,649	1,948	1,599	0.19	0.07 (2.7x)	0.10	0.09 (1.1x)
50%	∞	1,733,287	12,693	22,886	9,939	73.11	0.32 (228.4x)	2	0.67 (2.9x)
	40	799,207	12,693	19,335	9,405	24.50	0.32 (76.5x)	1.62	0.63 (2.5x)
	20	105,003	10,233	7,351	4,012	3.12	0.25 (12.4x)	0.41	0.25 (1.6x)
	10	11,570	3,649	1,944	1,085	0.26	0.08 (3.2x)	0.09	0.06 (1.5x)
70%	∞	1,855,924	12,705	22,872	6,483	76.47	0.31 (246.6x)	2.02	0.48 (4.2x)
	40	913,461	12,705	19,321	5,848	27.52	0.32 (86.0x)	1.65	0.44 (3.7x)
	20	133,800	10,245	7,339	1,882	4.12	0.25 (16.4x)	0.40	0.14 (2.8x)
	10	15,257	3,661	1,932	397	0.34	0.07 (4.8x)	0.10	0.03 (3.3x)
90%	∞	1,947,548	12,705	22,850	5,551	76.81	0.33 (232.7x)	2.11	0.40 (5.2x)
	40	995,711	12,705	19,299	4,769	29.02	0.33 (87.9x)	1.62	0.36 (4.5x)
	20	157,328	10,245	7,315	1,078	4.51	0.25 (18.0x)	0.40	0.07 (5.7x)
	10	17,019	3,661	1,910	173	0.37	0.07 (5.2x)	0.09	0.02 (4.5x)

* trans. = transactioning

References

1. L. Trevillyan et al., "An Integrated Environment for Technology Closure of Deep-Submicron IC Designs," *IEEE Design & Test,* vol. 21, no. 1, 2004, pp. 14-22.
2. M.D. Moffitt et al., "Path Smoothing via Discrete Optimization," *Proc. 45th Design Automation Conf.* (DAC 08), ACM Press, 2008, pp. 724-727.
3. L.N. Kannan, P.R. Suaris, and H.-G. Fang, "A Methodology and Algorithms for Post-Placement Delay Optimization," *Proc. 31st Design Automation Conf.* (DAC 94), ACM Press, 1994, pp. 327-332.
4. D.A. Papa et al., "Rumble: An Incremental, Timing-Driven, Physical-Synthesis Optimization Algorithm," *Proc. Int'l Symp. Physical Design* (ISPD 08), ACM Press, 2008, pp. 2-9.
5. H. Ren et al., "Hippocrates: First-Do-No-Harm Detailed Placement," *Proc. Asia and South Pacific Design Automation Conf.* (ASPDAC 07), IEEE CS Press, 2007, pp. 141-146.
6. K.-H. Chang, I.L. Markov, and V. Bertacco, "Safe Delay Optimization for Physical Synthesis," *Proc. Asia and South Pacific Design Automation Conf.* (ASPDAC 07), IEEE CS Press, 2007, pp. 628-633.
7. A.D. Drumm, R.C. Itskin, and K.W. Todd, *Method and Apparatus for Performing Timing Correction Transformations on a Technology-Independent Logic Model during Logic Synthesis,* US patent

5,003,487, to IBM Corp., Patent and Trademark Office, 1991.
8. R.P. Abato et al., *Incremental Timing Analysis,* US patent 5,508,937, to IBM Corp., Patent and Trademark Office, 1996.
9. J. Lee and D.T. Tang, ''An Algorithm for Incremental Timing Analysis,'' *Proc. 32nd Design Automation Conf.* (DAC 95), ACM Press, 1995, pp. 696-701.
10. S.S. Sapatnekar, ''Efficient Calculation of All-Pairs Input-to-Output Delays in Synchronous Sequential Circuits,'' *Proc. Int'l Symp. Circuits and Systems* (ISCAS 96), IEEE Press, 1996, pp. 724-727.
11. A. Mondal and C.A. Mandal, ''A New Approach to Timing Analysis Using Event Propagation and Temporal Logic,'' *Proc. Design, Automation and Test in Europe Conf.,* (DATE 04), IEEE CS Press, 2004, pp. 1198-1203.
12. D. Das et al., ''FA-STAC: A Framework for Fast and Accurate Static Timing Analysis with Coupling,'' *Proc. Int'l Conf. Computer Design* (ICCD 06), IEEE Press, 2006, pp. 43-49.
13. M.A. Kazda et al., *System and Method for Sign-Off Timing Closure of a VLSI Chip,* US patent 7581201, to IBM Corp., Patent and Trademark Office, 2009.
14. D. Bronnenberg, ''Static Timing Analysis Increases ASIC Performance,'' *Integrated System Design,* June 1999.
15. L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC Implementation, Circuit Design, and Process Technology* CRC Press, 2006.
16. W.E. Donath et al., ''Transformational Placement and Synthesis,'' *Proc. Design, Automation and Test in Europe Conf.* (DATE 00), IEEE CS Press, 2000, pp. 194-201.
17. J. Vygen, ''Slack in Static Timing Analysis,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 9, 2006, pp. 1876-1885.
18. W.E. Donath and D.J. Hathaway, *Distributed Static Timing Analysis,* US patent 6,557,151, to IBM Corp., Patent and Trademark Office, 2003.

David Papa is a research scientist at IBM's Austin Research Lab and has recently defended his PhD dissertation in computer science and engineering at the University of Michigan, Ann Arbor. His research interests include physical design, especially physical synthesis and placement. He has an MSE in computer science and engineering from the University of Michigan, Ann Arbor. He is a student member of IEEE.

Michael D. Moffitt is a research scientist at IBM. His research interests include constraint satisfaction, combinatorial optimization, and their application to large-scale problems commonly found in artificial intelligence and CAD. He has a PhD in computer science and engineering from the University of Michigan, Ann Arbor.

Charles J. Alpert manages the Design Productivity Group at IBM's Austin Research Laboratory. His research interests include design automation tools and methodologies to improve designer productivity and reduce design cost. He has a PhD in computer science from the University of California, Los Angeles. He is an IEEE Fellow.

Igor L. Markov is an associate professor of electrical engineering and computer science at the University of Michigan, Ann Arbor. His research interests include computers that make computers (software and hardware), secure hardware design, and combinatorial optimization with applications to the design, verification, and debug of ICs, as well as in quantum logic circuits. He has a PhD in computer science from the University of California, Los Angeles. He is a senior member of the ACM and IEEE.

■ Direct questions and comments about this article to Igor L. Markov, University of Michigan, Dept of EECS, ACAL Lab, 2260 Hayward Ave – CSE, Ann Arbor, MI 48109-2121; imarkov@eecs.umich.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

Think You Know Software?

PROVE IT!

How well do you know the software development process? Rise to the challenge by taking the CSDA or CSDP Examination.

With more and more employers seeking credential holders, it's a great time to add this unique credential to your resume.

www.computer.org/getcertified