A Difference Logic Formulation and SMT Solver for Timing-Driven Placement

Andrew Mihal
Tabula Inc., Santa Clara CA 95054, USA
amihal@tabula.com

Abstract
This paper presents a difference logic constraint satisfaction formulation and a custom SMT solver for programmable logic detailed placement problems. This problem domain is characterized by a large solution space with high space and time costs for generating constraints. To handle these problems efficiently, our solver features a dynamic clause generation callback interface to allow clauses to be added on demand during the search. The formulation and the solver both utilize concepts from static timing analysis to handle timing constraints. We show that the SMT approach provides better runtime and better quality of results than our previous Boolean SAT encoding.

1 Introduction

Semiconductor design automation tools have long made use of Boolean SAT for logic optimization, verification, test generation, and routing [13,10,5,15,21]. SAT-based placement has attracted less research interest. In the context of programmable logic, placers assign components from a netlist graph onto discrete sites on a programmable fabric. The result must satisfy a variety of constraints such as timing and routability. Devadas described placement via SAT-based bipartitioning in [4] but concluded that SAT solvers of the time were not yet powerful enough to solve more general 2-D placement problems.

Simulated annealing has been used effectively for placement [1], but this technique has disadvantages that can be addressed by a constraint satisfaction approach. Annealers make small perturbations to an initial placement (e.g. swapping the positions of two components) and accept or reject changes based on the total delta cost computed by a set of cost functions.

The simplicity of the move set makes it difficult for annealers to solve problems that require a coordinated change involving many netlist components. Such problems must be solved using a series of independent moves. The cost functions must be carefully crafted to ensure that partial progress towards the final goal is seen as gradual improvement. This property can be difficult to arrange, especially if the problem requires moving unrelated, non-violating components out of the way in order to make room for the components that are actually violating constraints.

A constraint satisfaction approach can directly address this weakness. Instead of searching for small changes that gradually approach an acceptable solution, a placer based on constraint satisfaction could rearrange a large number of components simultaneously such that the result satisfies all of the constraints. This strategy has the potential to outperform simulated annealing if it can be made efficient.

Scalability is a major technical obstacle to building a practical placer based on constraint satisfaction. In order for a sequential circuit to run at a specified clock rate, every state-to-state path in the netlist must have a total delay less than or equal to the target clock period. The number of such paths can be exponential in the size of the netlist.

Furthermore, each netlist edge has a number of placement options equal to the product of the number of candidate sites for the source and sink components. The routing delay between the source and sink sites on the fabric determines the edge delay, which in turn contributes to the path delays. Computing all of the fabric routing delays and generating the edge and path constraints is too expensive in runtime and memory to build a practical placer.
To solve this scalability problem, we use a two-part solution. First, concepts from static timing analysis are used to address the worst-case exponential scaling of path-based timing constraints. This formulation is presented in Section 3. Second, we use a dynamic clause generation approach to construct clauses lazily. We find that the solver is able to find a SAT or UNSAT result after exploring only a small fraction of the total search space, and therefore a majority of the clauses can be omitted entirely. This technique is presented in Section 4.

In the development of our detailed placer, our first approach was to use a purely Boolean formulation and solve it using a regular SAT solver. We explain this translation in Section 5 and discuss the shortcomings that motivated moving to an SMT solver that accepts difference logic constraints directly.

In Section 6, we present a custom difference logic SMT solver for the detailed placement problem domain. This solver specifically handles dynamic clause generation and the types of difference logic constraints that arise from our formulation of timing constraints based on static timing analysis. Section 7 gives experimental results comparing the difference logic formulation to our original purely Boolean SAT formulation. We also show the runtime improvement from dynamic clause generation. To get started, the next section briefly introduces the detailed placement problem for programmable logic.

2 Programmable Logic Detailed Placement

Programmable logic devices are implementation platforms for digital electronics that offer low up-front design costs and bypass the complexities of nanometer-scale transistor design. The general idea is that a prefabricated chip can be configured to implement any desired digital circuit by setting programmable bits on the chip.

Figure 1 shows an array of \( n \)-input lookup tables interconnected by a rich network of multiplexers. Each lookup table can implement any combinational function of \( n \) inputs by programming the \( 2^n \) bits of memory contained therein. Lookup tables are connected together by programming memory bits that drive the select signals of the multiplexers. A modern device may contain more than one million such lookup tables and often includes other resources such as flip-flops and memories. Kuon et al. [11] provide a survey of modern architectures.

Engineers create a design for implementation on programmable logic by writing a register-transfer level circuit description in Verilog or VHDL. A suite of design automation tools provided by the programmable logic vendor compiles the design and produces the programmable bit values for the chip. The compilation process has four major phases: synthesis, global placement, detailed placement, and routing. Synthesis compiles the circuit description into an optimized netlist of lookup tables. Placement assigns the lookup tables in the netlist onto lookup tables on the physical chip. Routing configures the multiplexers so that the connections specified in the netlist are made on the chip. Each phase poses intriguing optimization challenges. Chen et al. [2] give a broad survey of the techniques commonly used.

This paper focuses on the detailed placement problem, and specifically on timing optimization. Detailed placers assume that the netlist already has a placement that satisfies some global optimization criteria but still has problems of a more local nature that need to be repaired. For example, a global placer could use an analytical algorithm with a continuous coordinate system and an abstract geometric model of routing delay. The detailed placer refines this placement by snapping components to overlap-free discrete placement sites using a more accurate routing delay model based on actual paths through the interconnect network.

In addition to providing a Verilog or VHDL circuit description, users also specify that the sequential logic must run at a particular clock frequency. To meet this constraint the placer must ensure that all state-to-state paths in the netlist are placed such that the worst path delay is less than or equal to the target clock period \( \tau \).
3 Problem Formulation

In this section we describe how concepts from static timing analysis can be used to efficiently encode path-based timing constraints. We begin with variables that encode the placement of netlist components.
Timing-Driven Placement

onto sites. The Boolean variable $V_{AX}$ has the meaning that component $A$ is placed on site $X$. These placement variables can be arranged into a sparse matrix where the components are the rows and the sites are the columns as follows:

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$V_{AW}$</td>
<td>$V_{AX}$</td>
<td>$V_{AY}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>$V_{BX}$</td>
<td>$V_{BY}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>$V_{CY}$</td>
<td>$V_{CZ}$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1)

For each row in the matrix, an exactlyOne constraint (a standard atMostOne cardinality constraint combined with a standard Boolean clause) is added to ensure that each component gets a placement. For each column in the matrix, an atMostOne constraint is added to prevent placement overlaps.

Timing constraints for netlist edges follow this pattern:

$$(V_{AX} \land V_{BY}) \rightarrow (D_{AB} = d_{XY})$$

(2)

This formula says that if netlist component $A$ is placed on site $X$ and component $B$ is placed on site $Y$, then the delay on netlist edge $AB$ is equal to the routing delay between sites $X$ and $Y$. We assume there exists a timing model of the target architecture which provides these $d_{XY}$ numbers. The number of clauses for each edge is equal to the product of the number of candidate sites for the source and sink components.

Timing constraints for netlist state-to-state paths follow this pattern:

$$D_{AB} + D_{BC} + D_{CD} + \ldots \leq \tau$$

(3)

To avoid making a constraint of this type for every path in the netlist, we rewrite equations 2 and 3 using concepts from static timing analysis. Static timing analysis computes an arrival time and a required time at each component in the netlist graph [9]. The arrival time is the maximum path length to state elements backwards through the transitive fanin of a component. This value represents the time it takes from the beginning of the clock cycle for the data to propagate through the circuit and to become valid at the output of the component. The arrival time at the output of a state element is defined to be zero.

The required time is the clock period $\tau$ minus the maximum path length to state elements forwards through the transitive fanout of a component. This value is the latest time at which the component output can become valid and still make it to the downstream state elements before the end of the clock cycle.

The required time at the input of a state element is defined to be $\tau$.

The slack of a component is the required time minus the arrival time. A component with negative slack is on a path that fails timing. Instead of making a separate timing constraint for each state-to-state path in the netlist, one can simply constrain each netlist component to have non-negative slack with a clause $Arr_A \leq Req_A$.

The arrival time $Arr_A$ and required time $Req_A$ are defined in terms of the immediate fanin and fanout components:

$$Arr_A = \max_{\text{fanin} F_i} (Arr_{F_i} + D_{F_iA})$$

(4)

$$Req_A = \min_{\text{fanout} F_o} (Req_{F_o} - D_{AF_o})$$

(5)

Using these formulas, Equation 2 can be rewritten as:
\[(V_{AX} \land V_{BY}) \rightarrow (\text{Arr}_B - \text{Arr}_A \geq d_{XY}) \]
\[\land (\text{Req}_A - \text{Req}_B \leq -d_{XY}) \quad (6)\]

The result is a difference logic formulation. The only non-Boolean variables are the arrival and required times \(\text{Arr}_A\) and \(\text{Req}_A\) for each netlist component.

This formulation does not require enumerating all state-to-state paths and avoids the worst-case exponential number of constraints. However, the number of constraints of the form of Equation 6 still grows with the product of the number of placement options for the source and sink components for each netlist edge. This is still an impractically large number of constraints. We address this scalability issue using dynamic clause generation.

4 Dynamic Clause Generation

Observe that the placement variable matrix (1) is mostly false due to the \textit{exactlyOne} constraint on each row. Also, each clause \((V_{AX} \land V_{BY}) \rightarrow (\ldots)\) starts with two negative literals \((V_{AX} + V_{BY} + \ldots)\). A majority of the clauses are therefore trivially satisfied during the search.

Furthermore, the detailed placer is responsible for repairing only local constraint violations and does not attempt to make global changes to the initial placement. It is expected that only a minority of netlist components will have to be moved to accomplish this task. The solver is likely to find a SAT or UNSAT solution after attempting only a small fraction of the placement options \(V_{AX}\) for each component.

Therefore, while all of the clauses are theoretically necessary for correctness, in practice most of them do not affect the search. We can take advantage of this fact to reduce the working size of the problem. Clauses of the form \((V_{AX} + V_{BY} + \ldots)\) can be left out until the search actually enters a subspace where \(V_{AX}\) and \(V_{BY}\) are both true.

Our solver provides a callback method \textit{decisionCallback} that is invoked after assigning a placement variable \(V_{AX} = T\) and after all unit propagations have completed without conflicts. The placer adds clauses starting with \((V_{AX} + \ldots)\) during this callback. All fanin and fanout components of \(A\) are checked to see if they are placed in the solver state at the time of the callback. Then clauses related to fanin edges \(FiA\) and fanout edges \(AFo\) are generated.

The solver then processes the incoming clauses and makes its internal state consistent using a minimal amount of backtracking when necessary. The search can then continue in the subspace under \(V_{AX} = T\) as normal. The resulting behavior is the same as if the dynamic clauses had always been present, but with substantial runtime savings since most clauses are never generated.

Our approach is complementary to the two-watched-literal technique [14] in that it addresses clause generation runtime and memory footprint in addition to search runtime. Quantitative results are given in Section 7.

4.1 Related Dynamic Approaches

Eén and Sörensson describe a dynamic clause generation approach in [7] wherein the solver is restarted with additional clauses after a complete solution has been produced and examined. Our approach adds clauses while the solver is running and not only between invocations of an incremental solver.

The Lynx SAT solver includes a similar callback method for adding clauses after each propagation step [8]. That callback method examines the solver trail and adds clauses that conflict with the current assignment. In comparison, our approach allows clauses that do not conflict with the current assignment to be added as well.
Ohrimenko et al. [18] describe a purely Boolean encoding of difference logic constraints that includes lazy clause generation to reduce the number of Boolean clauses created for difference logic propagation. Our approach performs dynamic clause generation on a coarser level of abstraction. This is possible due to the natural subdivisions of the problem space and the low likelihood of actually searching the majority of these subspaces.

5 Boolean Formulation

Our first approach to constructing a detailed placer based on constraint satisfaction was to convert the difference logic timing constraints into a purely Boolean SAT problem [12]. We then added support for dynamic clause generation to MiniSAT version 1.12b to solve the instances [6]. This formulation is briefly described here because it is used as a baseline for measuring the benefits of our difference logic SMT solver.

The Boolean formulation uses a small-domain encoding variation due to Ohrimenko et al. [18]. The key idea is to define Boolean variables that represent upper and lower bounds on the difference logic variables instead of exact values of the variables. This approach is a natural fit for modeling arrival and required times and detecting negative slack.

For each netlist component $A$ we create a number line subdivided into $T$ discrete values representing times within the clock period $\tau$. Each subdivision has an associated non-decision Boolean variable $E_{At}$. When true, this variable has the meaning that $\text{Req}_A \leq t_{\tau T}$. When false, this variable indicates that $\text{Arr}_A > t_{\tau T}$.

The clauses shown in Equation 6 are then rewritten using these Boolean variables:

\[(V_{AX} \land V_{BY}) \rightarrow \bigwedge_t (E_{At} \rightarrow E_{B(t+\Delta)}) \text{ where } \Delta = d_{XY} \frac{T}{\tau} \quad (7)\]

If the solver makes placement decisions that result in an arrival time becoming larger than a required time at any netlist component, then some variable $E_{At}$ will be assigned to both true and false. Timing violations are therefore detected as ordinary Boolean conflicts.

A major drawback of the purely Boolean encoding of timing constraints is the quantization of time. Arrival and required times must be rounded conservatively to discrete number line variables $E_{At}$. On a long state-to-state path through the netlist, these rounding errors accumulate and over-constrain the problem. This is especially problematic when the target clock period $\tau$ approaches the maximum frequency supported by the target architecture. The conservative rounding forces the placer to find a solution that exceeds the actual target, but the architecture does not have interconnect routes that are fast enough. Consequently, placements that would actually be acceptable are rejected. The quantization of time can be partially addressed by increasing the number of subdivisions $T$, but this additional accuracy comes at the cost of increasing the number of clauses in the formulation.

The Boolean encoding is also computationally expensive. Boolean constraint propagation is used to mimic difference logic constraint propagation (e.g. addition and comparison of numerical variables). Both of these drawbacks can be addressed by using a difference logic SMT solver instead of a Boolean SAT solver.

6 A Dynamic Difference Logic Solver

Since dynamic clause generation is an essential component of our approach, we constructed a custom difference logic SMT solver instead of using a publicly available solver. Our design decisions for im-
plementing the solver are influenced by our experience building static timing analysis tools and prior work on DPLL(T) SMT solvers and difference logic solvers [3, 20, 16, 17].

The general software architecture of our solver follows MiniSAT but is entirely new code. Boolean literals are encoded as positive integers, with even numbers used for positive literals and odd numbers for negated literals. Difference logic literals are encoded as negative integers, with even numbers used for lower bounds and odd numbers used for upper bounds: The difference logic portions of the solver use floating-point math.

\[
\begin{array}{cccccccc}
& -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & \cdots \\
\cdots & \overline{DL_1} & (DL_1) & \overline{DL_0} & (DL_0) & B_0 & B_0 & B_1 & B_1 & \cdots \\
\end{array}
\]

(8)

The placer creates one difference logic variable for each netlist component and one difference logic variable for each netlist edge. The solver is informed which variables represent components and which represent edges, and is also provided the graph connectivity of these variables. This graph structure is used during the propagation phase to perform t-propagations and deduce t-consequences.

Netlist edge delay clauses are of the form:

\[(V_{AX} \land V_{BY}) \rightarrow (D_{AB} \geq d_{XY})\]  

(9)

**Propagation**

The propagation phase is divided into three parts which are executed in priority order. First, all Boolean unit propagations are made until nothing else can be propagated. If any conflicts are encountered, the solver performs conflict analysis and nonlinear backtracking as in MiniSAT.

After Boolean unit propagations are finished, the solver performs all difference logic propagations in a single batch t-propagation step. This is the “lazy” approach described in [3].

The difference logic propagation mirrors the arrival and required time propagation used in static timing analysis. If any netlist component has a new lower bound, an update is scheduled to recompute lower bounds on all of its fanout components. Similarly, any component with a new upper bound schedules upper bound updates for all of its fanin components. A new lower bound on a netlist edge schedules lower bound updates on fanout components and upper bound updates on fanin components.

Upper and lower bounds are propagated forwards and backwards throughout the netlist graph until the system converges or until the Bellman-Ford iteration limit is reached which indicates that a positive-weight cycle has been discovered. If such a cycle is found or if at any time some difference logic variable has a lower bound greater than its upper bound, the propagation phase stops and the solver proceeds to conflict analysis.

The result of the difference logic propagation phase is that all netlist components have updated arrival and required times that are consistent with the known netlist edge delays. The propagation also computes new upper bounds on the delays of edges:

\[(\text{Arr}_A \geq a \land \text{Req}_B \leq b) \rightarrow (D_{AB} \leq b - a)\]  

(10)

The solver uses these upper bounds to rule out future placement decisions that would result in edge delays that are too large. This feedback is one example of how t-consequences guide the Boolean part of the search.

Finally, after all Boolean and difference logic propagations have completed, the solver performs dynamic clause generation. The decisionCallback function is called for all new Boolean variables that have been assigned since the last round of dynamic clause generation. The placer adds clauses that are now relevant to the search.
The strict priority order of Boolean propagation, difference logic propagation, and dynamic clause generation reflects the runtime cost of each of these steps. The solver attempts to find conflicts using existing clauses before asking the placer to add new clauses. Thus no computation is wasted when the solver is destined to backtrack out of the current subtree. Also, the solver attempts to discover conflicts resulting from Boolean propagations before performing a more expensive t-propagation step.

Conflict Analysis

Conflicts are analyzed and learned clauses are generated using the 1-UIP approach as in MiniSAT. For each difference logic variable, the solver maintains two stacks of \text{\{value, reason, level\}} tuples that store monotonically tightening upper and lower bounds on the variable’s value. These stacks are searched to find the earliest decision level at which a difference logic literal became true or false. The matching stack entry contains the reason data necessary to build the implication graph and construct the 1-UIP learned clause.

The reason for a difference logic literal to be true or false can be one of two possibilities. If the literal was assigned as a result of a clause becoming unit, the reason field will refer to that clause. The conflict analysis algorithm can then continue unwinding that clause.

If the literal was assigned during difference logic propagation, the reason field will refer to a pair of difference logic node and edge literals that are responsible for the propagated value. For example, consider a netlist edge $AB$ with delay $D_{AB} = 10.0$ and source node $A$ with $\text{Arr}_A \geq 20.0$. The difference logic propagation would enqueue $\text{Arr}_B \geq 30.0$. If that literal is later involved in a conflict, the reason will appear to be the clause $(\text{Arr}_A \geq 20.0 + D_{AB} \geq 10.0 + \text{Arr}_B \geq 30.0)$. This clause is not in the clause database. It is a temporary object that is made so that the conflict analysis algorithm can unwind clausal implications and t-consequences identically.

7 Results

To measure the improvement of the difference logic formulation over the purely Boolean formulation, we ran a series of experiments placing netlists from the OpenCores database \cite{19} using both formulations. Our custom difference logic solver is used in both cases. The purely Boolean problems use the formulation described in Section 5 which uses no difference logic variables.

These results are based on the complete detailed placement tool. In addition to the timing constraints described in this paper, our placer also considers placement legalization constraints and routability constraints. The decision variables are also more extensive than described here. The placer is able to explore lookup table pin permutations and alternative routing paths for netlist edges between the same source and sink component sites. Retiming can also be applied to move logic across state elements.

A search-and-repair strategy is used that breaks the entire placement problem into smaller subproblems that each try to repair a specific timing violation. Each subproblem contains a subset of the netlist with approximately 100 components in the neighborhood of a timing violation. This subproblem size is sufficient to solve complex violations that require coordinated motion of dozens of components in order to repair. Each SAT result repairs not only the targeted timing violation but also other violations that are coincidentally included in the neighborhood. The placer creates and solves a series of subproblems until all of the timing violations in the netlist have been repaired.

The runtime and frequency numbers in Table 1 are normalized to the Boolean formulation results and represent the total runtime of the tool. The difference logic formulation achieves higher frequency placements due to the increased accuracy of representing delays using floating point numbers in the solver. The pessimistic rounding required by the Boolean formulation over-constrains the problem. In some cases, the difference logic formulation is able to find a placement that achieves the highest
frequency possible on the fabric. This happens when the entire critical path in the netlist is placed using the shortest possible fabric connections.

For the difference logic formulation, the table lists an RT Match result that indicates how long it took to attain the same frequency as the Boolean formulation, and a RT Best runtime for the maximum achieved frequency. The difference logic formulation is able to attain an equivalent result in less runtime, and goes on to achieve frequencies that are out of reach of the Boolean formulation.

The difference logic formulation is also more efficient in the number of clauses and variables. Table 1 lists the average number of clauses and variables over all of the subproblems in each netlist, which are about the same size independent of the total netlist size. In the Boolean formulation, equation 7 expands into a large number of clauses that mimic arrival and required time propagation. These are unnecessary in the difference logic formulation, as are the Boolean variables used to represent quantized times.

Table 2 shows the average per-instance runtime and clause count with and without the dynamic clause generation technique. Due to the long runtimes, results are shown only for small netlists. For both the Boolean and difference logic formulations, dynamic clause generation leads to approximately an order of magnitude reduction in runtime for constructing and solving each instance.

8 Conclusion

Programmable logic detailed placement is a challenging problem domain that has not been previously solved with a constraint satisfaction approach. Using the techniques described in this paper, our formulation can be solved efficiently and used to construct a practical placement tool.

One of the keys to making the approach scalable is to take advantage of the natural subdivisions of the problem space and the fact that the solver is able to determine a SAT or UNSAT result without visiting a majority of the subspaces. This characteristic makes the problem domain amenable to on-demand dynamic clause generation. Other problem domains with the same characteristic should be able to enjoy the same benefits.

The SMT-based placer is currently deployed in a production placement tool at Tabula. It replaces the purely Boolean formulation that was solved with a version of MiniSAT extended to support dynamic
clause generation. That approach, in turn, replaced a placement tool based on simulated annealing. The constraint satisfaction approaches have provided considerable runtime and quality benefits and we are encouraged to explore this technology further.

References