Constructive Multilevel Logic Synthesis Under Properties of Boolean Algebra

Victor N. Kravets and Karem A. Sakallah
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48104
{vkravets, karem}@eecs.umich.edu

Abstract

We describe a new constructive multilevel logic synthesis system that integrates the traditionally separate technology-independent and technology-dependent stages of modern synthesis tools. Dubbed M32, this system is capable of generating circuits incrementally based on both functional as well as structural considerations. This is achieved by maintaining a dynamic structural representation of the evolving implementation and by refining it through progressive introduction of gates from a target technology library. Circuit construction proceeds from the primary inputs towards the primary outputs. Preliminary experimental results show that circuits generated using this approach are generally superior to those produced by multi-stage synthesis.

I. Introduction and Motivation

In this paper we describe a new multilevel logic synthesis system, M32, that departs in several important respects from current practice in logic synthesis technology. The development of M32 was motivated by the oft-cited refrain that wires are starting to dominate active logic in determining the area and speed of deep submicron ICs, and that current synthesis flows are biased primarily towards optimizing gates. M32 was designed to address this bias by intertwining the traditionally separate phases of technology-independent Boolean optimization and technology-dependent mapping in a constructive synthesis strategy that is cognizant of the structural implications of optimization decisions.

To quantify the impact of wires on circuit area we conducted a controlled experiment that compared the layouts of combinational circuits that have the same active areas but different interconnect patterns. The layouts were generated using the Epoch [9] standard cell place and route tools for a two-layer $0.5\mu m$ CMOS IC process that allows over-cell routing. I/O pins were distributed around the perimeter of the standard cell block. The plot in Figure 1 shows the total routing area as well as delay per logic level as functions of topological complexity given by

$$\text{Topological complexity} = \frac{\sum_{n=1}^{\#\text{Edges}} L(n)}{\#\text{Edges}} \tag{1}$$

where $L(n)$ is the number of topological levels crossed by wire n and #Edges is the total number of wires in the circuit. This metric is similar to the fanout range suggested by Vaishnav and Pedram [30] for controlling routing complexity during technology-independent logic synthesis. As the figure clearly shows, routing area increases with increasing topological complexity, and begins to exceed active area when topological complexity is around 2. Similarly, signal delay per logic level increases with increased topological complexity. While these results may be specific to the particular IC technology and physical design system used in the experiment, they nevertheless confirm the general belief that wiring can be a significant contributor to area and delay.

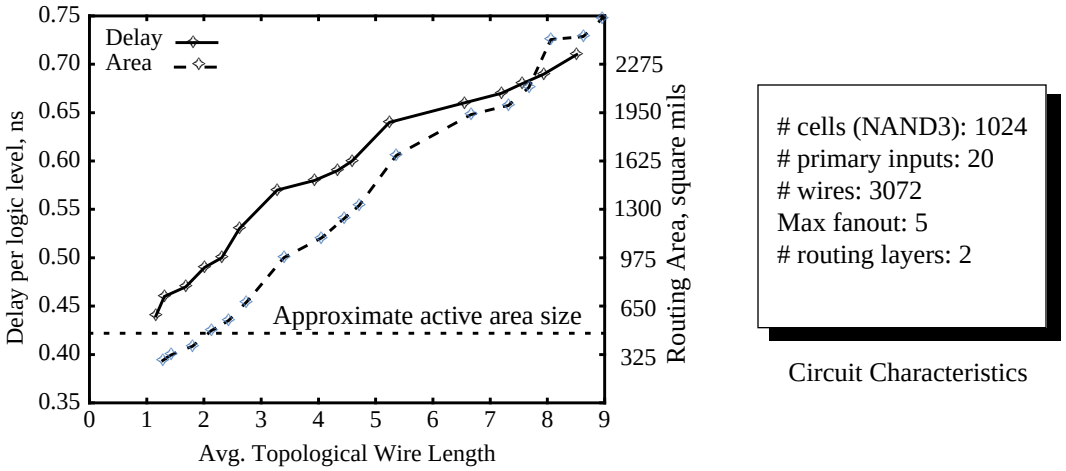


Fig. 1. Effects of wiring on circuit area and delay.

An algorithm which produces a technology specific network implementation from the initial phases of synthesis is one of the goals in this work. The motivation for this approach is to allow exploration of degrees of freedom which are usually lost in the split between technology-independent and technology-dependent phases. Such an approach enables the algorithm to account for technology specific functional characteristics (e.g. signal arrival time and vertex delay), and network structure (e.g. connectivity). Furthermore, this makes the task of assessing final design quality more accurate from the initial steps of synthesis, whose pre-determined properties can be annotated forward to the back-end tools. For example, physical location of gates in the final design can be determined at the early stages of synthesis in order to meet tighter delay constraints.

The remainder of this paper is organized as follows. Section II gives a brief description of prior work. A synthesis algorithm based on the constructive synthesis methodology is then presented in Section III. It overcomes the limitations of previous constructive approaches and benefits from recent advances in synthesis technology, allowing it to handle much larger circuits. Its prototype implementation in the M32 synthesis system has been demonstrated for combinational circuits containing several thousand gates. An example illustrating execution of the algorithm is given in Section IV. Experimental results discussed in Section V.

II. Prior Work

Combinational multilevel logic synthesis is the process of implementing a set of logic expressions using cells from a technology library, each with a prescribed function and physical characteristics [27]. Most of the current logic synthesis systems divide the logic synthesis process into technology-independent [5] and technology-dependent stages [17, 11]. The technology-independent stage focuses on partitioning the logic, whereas the technology-dependent stage chooses appropriate gates from the library to implement the partitioned logic. Such multi-stage approaches to complex optimization problems are common in electronic design automation (e.g. placement followed by routing) and are usually necessitated by the difficulty of solving these problems conjointly.

This “serialization” of the optimization process implies that decisions made in earlier stages must necessarily be based on loose estimates of what later stages can accomplish. At the same time, the solutions produced by early stages place limitations on the degrees of freedom to improve final implementation of a design. For two-stage logic synthesis, decisions made during the technology-independent stage [6, 8, 7] significantly determine the structure of a circuit. They are made with no regard for the downstream technology. When the technology characteristics become available in the mapping stage it is often too late to augment the effects of these decisions to improve circuit quality.

The back-annotated approaches, which perform resynthesis after technology specific information is extracted from the mapped circuit, compensate partially for this problem. Given a *sign-off*, information these approaches would typically resynthesize the circuit through critical section correction [15, 32, 2, 22, 14, 28]. While this yields improvement in circuit quality, technology-independent and technology-dependent transformations still remain disconnected. In [20], authors address this problem by dynamically modifying the set of AND2/INV decompositions while deleting others based on the

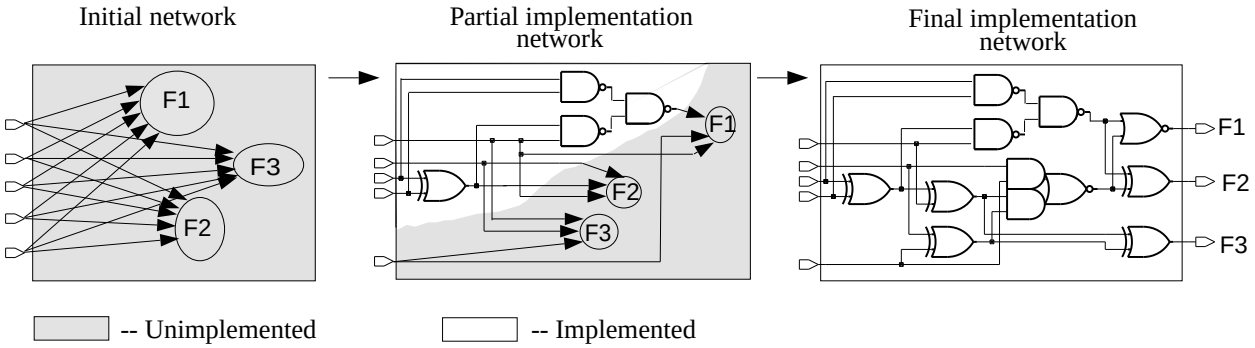


Fig. 2. Illustration of the constructive technology-dependent synthesis

actual cost function used in technology mapping. It allows technology-independent transformations to be part of the technology mapping. However, these transformations do not exhibit global knowledge about circuit structure and functionality

This problem becomes more important with advances in CMOS technology since interconnections are becoming a major concern in today's high-performance, high-density ASIC designs [18]. It is therefore critical for a synthesis tool to have accurate estimation of wiring from the initial stages of synthesis. In [30], the authors pursued this problem by relying on the traditional synthesis methods, accounting for routing area during logic decomposition using heuristics for minimizing the fanout ranges of a decomposed network. Another method of minimizing routing cost by deriving and maintaining an order among primary inputs of the circuit was proposed in [24, 1]. These approaches operate during the early phase of synthesis, and remain removed from the final function-realizing circuit. In [23] an attempt is made to account for the interconnections during the technology mapping phase of synthesis. The idea is to generate a "companion" placement solution for the circuit before it is mapped. This placement is then dynamically updated as the mapping process tries to evaluate the cost of a matching gate. The algorithm estimates the interconnection contribution to the area and delay by referring to the dynamically updated placement of the network. Using this technique, the authors are able to generate circuits with shorter wire length and smaller area.

Realizing the need for synthesis to account for the "physical" information of back-end tools, Synopsys makes it possible to choose an appropriate wire-load model [19]. The wire-load models specified in the Synopsys technology library are based on statistical data which is design and process technology-dependent. Thus, inaccuracies in wire-load models can lead to synthesized designs which are pessimistic, unroutable, or don't meet tight constraints after routing is performed. The synthesis process in Synopsys also relies on the methodology of technology-independent transformations, which is not suited to account for the final wire lengths of a design.

Several attempts to synthesize networks incrementally are reported in the literature. Davidson presented a branch-and-bound algorithm for NAND network synthesis [10]. The algorithm constructs a network realization incrementally starting from the primary outputs. In each iteration, the algorithm extends a partial solution by introducing a new NAND gate together with its fanin connections. Another incremental synthesis procedure for arbitrary gates was presented by Schneider and Dietmeyer [26]. In each step, their algorithm finds the circuit package (i.e. gate type in a technology library) that satisfies some goal, such as area and delay, while meeting fanin, loading and power constraints. Such approaches, however, did not yield practical synthesis tools: their exponential run time complexity rendered them useless except for very small circuits.

III. The M32 Logic Synthesis System

M32 is a multilevel technology-dependent constructive logic synthesis system. It relies on extended algebraic decomposition techniques which define a feasible space of transformations which can be applied to a sum-of-products expression. As an optimization objective the system considers the structure and size of a circuit. Constraints are implied by the gate library. The system is currently geared towards performance-oriented synthesis which also minimizes average topological wire length. In this section we first give an overview of the system, and then describe its main points in detail.

```
while exists f_i ∈ F such that f_i is not a single literal do {
    k ← SelectFunction( ) ;          // Determine largest f_k,      (1 ≤ k ≤ n)
    P ← GeneratateDivisor(f_k) ;     // Select atomic divisor P
    ẏ_v ← IntroduceGates(P) ;        // v is output vertex of subckt implementing P
    Substitute(P, ẏ_v) ;             // Re-express F in terms of the newly
                                     // implemented logic
```

Fig. 3. Synthesis loop in the M32 system

A. System Overview

The overall synthesis process in M32 is depicted in Figure 2. The state of the synthesis process is modeled by a *Boolean network* η [4] that captures an evolving structural representation of the functions being synthesized. Each vertex v in such a network has an associated Boolean function f_v that computes the logic value at the vertex's output in terms of the logic values on its inputs. A vertex is considered to be *implemented* if its function is equivalent to one of the functions in a given gate library L, and *unimplemented* otherwise. To insure an *onto* mapping from L to the vertices in η, the library must, at a minimum, have inverter and pass-through wire gates as well as any 2-input gate that makes it functionally complete.

Reading its functional specification F as a set of multi-output cubes, M32 constructs an initial *specification* network consisting of implemented vertices corresponding to the primary inputs and unimplemented vertices corresponding to the specified output functions $f_1, f_2, \ldots, f_n$. As the synthesis process evolves, the functions of unimplemented vertices are successively decomposed in terms of those of already implemented vertices, resulting in a series of *partial implementation* networks. The decomposition is closely tied with the given gate library, and leads to the creation of new implemented vertices, i.e. vertices that correspond to library gates. Each implemented vertex v introduces a new variable y_v which can now be used to simplify the functions of unimplemented vertices. The effect of these successive decompositions is an expansion of η from the PIs towards the POs as more implemented vertices are created and as the functional complexity of unimplemented vertices is reduced. This constructive creation process makes it possible to control the structural complexity of the evolving implementation. The synthesis process terminates when all vertices become implemented yielding a *final implementation* network.

The main loop in the M32 synthesis algorithm is shown in the pseudo-code of Figure 3. The algorithm treats F and η as globally accessible structures. In each iteration, the functions of unimplemented vertices are examined and one of them, f_k, is selected for a decomposition step. An *atomic divisor* P is extracted from f_k by an appropriate division procedure. The divisor is subsequently implemented by a small set of vertices corresponding to gates from L leading to the expansion of the implemented part of η. The decomposition step is completed by substituting the variables y_v of the newly created vertices into the functions of the unimplemented vertices. An unimplemented vertex becomes implemented when its function reduces to a single literal. Thus, the iteration stops, signalling completion of the synthesis process, when all the functions in F have been reduced to single literals.

This algorithm has several features that distinguish it from commonly-used synthesis methods:

- It interleaves functional decomposition and technology mapping throughout the synthesis process
- It considers the structural implications of candidate decompositions
- It selectively applies Boolean transformations to improve synthesis quality without adversely affecting run time efficiency

The remainder of this section is devoted to detailed descriptions of the four main routines of the algorithm. To facilitate these descriptions we introduce the following definitions. The depth of a literal, $\text{depth}(\dot{y}_v)$, is the topological depth of its corresponding vertex v in η; the depth of a primary input is defined to be 0. The depth of an expression E, or a set of literals, is the maximum depth of any of their literals. The set of literals appearing in E will be denoted $\text{support}(E)$. The num-

ber of times a literal $\dot{y}_v$ occurs in E will be denoted by occurrence$(\dot{y}_v, E)$, and size(E) will denote the number of literal occurrences in E.

B. SelectFunction

The order in which the functions of F are decomposed clearly affects the final synthesized implementation. In each iteration of the algorithm, the set of vertices that are candidates for decomposition is the subset of unimplemented vertices whose fanins are already implemented. The *SelectFunction* routine identifies the next function to decompose by greedily choosing the candidate vertex whose function has the largest number of literals in its cube representation. This choice is motivated by the expectation that larger, richer, functions yield better divisors. In this context, divisor P_1 is considered better than divisor P_2 if the vertices in its implementation subcircuit represent better opportunities for sharing among the unimplemented vertices. More sophisticated selection strategies can be easily envisioned, especially when the initial specification is a multi-level network.

C. GenerateDivisor

Like most modern synthesis algorithms, M32 relies on an efficient division procedure to decompose a function f into the form $pq + r$. The most commonly used division procedure is weak division [8] which is applied to an SOP expression for f and is equivalent to excluding all Boolean transformations except for the distributive law. While primarily motivated by the need for a fast division operation, weak division has been empirically shown to yield acceptable decompositions in practice. Still, the quality of the divisors, as measured by the total number of literals in the resulting factored form, can be improved by the judicious application of additional Boolean transformations. The division operation in M32 augments the distributive law with the annihilation ($a'a = \mathbf{0}$) and idempotency ($aa = a$) properties to generate better decompositions. This additional flexibility comes at a modest computational cost.

Example 3.1 Let $f = abcg + abe + acde + a'b'cd + a'b'e' + a'ce'g + cdg$. A possible algebraic decomposition of f obtained using weak division is $f = (ab + d + a'e')cg + abe + acde + a'b'cd + a'b'e'$ which has a literal cost of 21. Use of annihilation and idempotency yields the more compact decomposition $f = (ab + cd + a'e')(ae + cg + a'b')$ whose literal cost is only 12. Unlike the algebraic decomposition which yields factors with disjoint support, this decomposition produces factors that have joint support. ■

It is interesting to note that limiting allowable transformations of Boolean expressions to the above three properties (distributivity, annihilation, and idempotency) guarantees that any generated factored form will reproduce the original SOP cover under *flattening* [16]. This, in turn, implies that different initial SOP representations of a function can lead to different decompositions. Removal of this bias requires the deployment of the entire arsenal of Boolean transformations, i.e. operating in the unrestricted functional domain. In general, the attendant improvement in the quality of such unrestricted decomposition comes at a steep computational cost to be practical. M32 partially compensates for this bias by intertwining decomposition with mapping to a specific gate library while managing the structural attributes of the evolving implementation.

Divisor selection in M32 is accomplished through successive factorization, using the distributive law, of repeated literals from an SOP expression f. The annihilation and idempotency transformations are subsequently invoked to modify the resulting quotient and reduce the literal cost of the decomposition. This process is iterated until each literal appears only once in the factored form. To account for the structural implications of particular decompositions, literals are chosen based on a structural cost metric that is computed according to:

$$\text{cost}(\dot{x}) = \frac{\displaystyle\sum_{\dot{y}_v \in \text{ support}(f/\dot{x})} \frac{[\text{depth}(f/\dot{x}) - \text{depth}(\dot{y}_v)]}{\text{occurrence}(\dot{y}_v, f/\dot{x})}}{\text{size}(f/\dot{x})}. \tag{2}$$

where $\dot{x}$ is a candidate literal in expression f and $f/\dot{x}$ is the quotient resulting from algebraic division of f by $\dot{x}$. *GenerateDivisor* creates a divisor by successively selecting the next candidate literal with least cost according to (2). Signal arrival times were also factored into divisor selection.

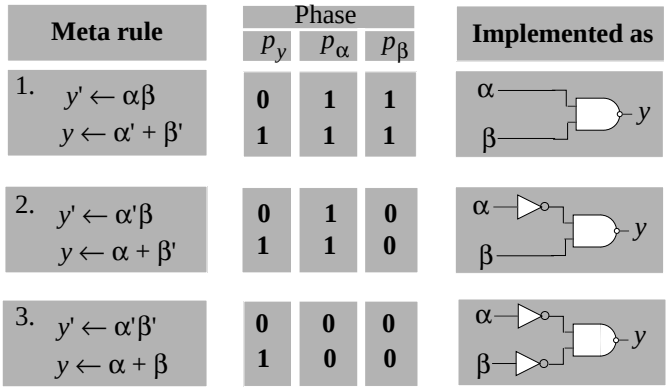


Fig. 4. Meta rules describing variable support selection and construction of the circuit

D. IntroduceGates

The technology mapping step in modern logic synthesis is based on graph covering of an intermediate Boolean network obtained through technology-independent functional decomposition. This "optimized" network is initially translated into a forest of trees each of whose vertices is an inverter or a 2-input NAND gate. The trees are subsequently mapped, in topological order, to the target technology library. This involves two steps: (1) pattern matching [11] or Boolean matching [25, 21]; and (2) gate assignment [17]. In the matching step all possible gate functions in the library that are logically equivalent subtrees rooted in a given vertex are considered. In the gate assignment step an optimal match is selected, and its corresponding subtree is implemented in terms of the library gate. These steps are applied recursively starting from the primary using dynamic programming [3].

Unlike conventional technology mappers that operate on an intermediate "optimized" Boolean network obtained in a prior technology-independent phase, M32 closely ties its creation of gates with functional decomposition. As soon as a divisor P is found by *GenerateDivisor*, *IntroduceGates* proceeds to map it to the given gate library. The mapping process is also different from those used in conventional synthesis tools: no intermediate *subject graph* is constructed. Similar to GenerateDivisor, this procedure is aware of the structural implications of its choices, and involves iterating the following steps until P is fully implemented by library gates:

1. A gate from the technology library with its suitable variable support in P is selected for instantiation as vertex v
2. The vertex v, along with its possible fanin inverters, is instantiated and added to η; variable y_v is associated with the new vertex v
3. The divisor P is re-expressed in terms of y_v

The implementation of *IntroduceGates* is currently limited to a small technology library defined by $L = \{wire, INV, NAND2\}$. Thus, the gate type selection step in the above procedure is unnecessary. Enhancements to *IntroduceGates* that are currently underway extend its capabilities to richer gate libraries. The selection of a suitable variable support in step 1 is also simplified by this library choice. Candidate variable subsets are determined by considering all associative groupings of literal pairs in P, and the best pair is selected. Again, quality in this context is estimated by a structural metric: a pair of literals which instantiates vertex v of least depth is greedily selected.

The depth of v in η is derived from the depth of vertices in the subexpression E of an associative grouping while accounting for the possible presence of an inverter on v's fanin lines. The presence of an inverter on the fanin lines is determined by matching subexpression E to the NAND2 function under input/output phase assignments. Due to the small number of possible matching combinations when using only NAND2 gates, we enumerate them as *meta rules* in Figure 4. To break ties between the candidate supports we have used estimated signal arrival times based on the following delay model:

$$delay = \tau_g + n \times \tau_o \times C_p \tag{3}$$

where τ_g is the intrinsic gate delay, n is the gate fan-out, τ_o is fanout delay, C_p is capacitance on a fanout pin. In the SIS-1.2. system this model is known as *library model*. In our experiments we have used nominal delay and capacitance was taken from the `MCNC` library.

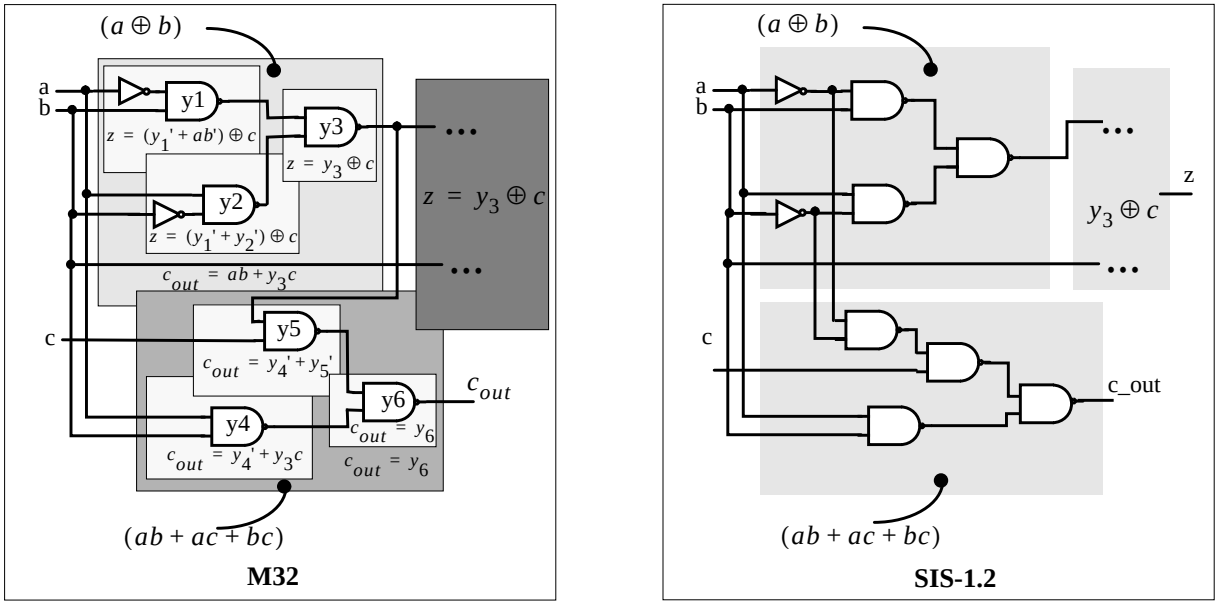


Fig. 5. Synthesis of the full adder in M32 and SIS-1.2

After a gate support and its phases are determined, it is introduced into the circuit by establishing its fan-in connections and placing an inverter (either a new one or reusing a previously introduced one) on each connection which has negative (0) input phase. No inverter is placed on the gate output regardless of the output phase.

Finally, expression P is re-expressed according to the newly implemented part of P. This is done by applying substitution (described below), which replaces the subexpression E with the new variable y_v using its complemented and non-complemented forms. Thus, each iteration in *IntroduceGates* reduces the size of P. The process continues until P is reduced to a single literal. This literal is then returned by the *IntroduceGates* routine, and is later used to substitute it for P in F.

E. Substitute

At each iteration of the algorithm routine *Substitute* re-expresses functions $f_1, \ldots, f_n$ of the unimplemented vertices in η in terms of the newly introduced vertices. The substitutions applied to the functions are based on the division operation. If $pq + r$ is the result dividing f by p then substitution re-expresses f as $\dot{y}_v q + r$, where $\dot{y}_v$ is a new literal. The $pq + r$ factored forms define feasible substitutions. At the substitution step they are currently based on the distributive, annihilation, and idempotency law of Boolean algebra applied to SOP forms. To allow more feasible decompositions during the substitution process we have also used cube reduction, which was implemented the using sharp product operation $A\#B$, and defined as $A\bar{B}$ [12]. Application of these operations is performed if it facilitates division with respect to a given divisor. Example in Section IV illustrates how the use of cube reduction can lead to the better design.

Substitute re-expresses $f_1, \ldots, f_n$ in two steps: (1) substituting literal $\dot{y}_v$ ($\dot{y}_v'$) for the P (P') function; and (2) substituting variables of newly introduced vertices for their gate functions, modulo the phase assignment on its inputs determined in *IntroduceGates*. These steps are applied to each of the functions $f_1, \ldots, f_n$ individually. The routine performs substitution selectively to minimize topological cost. Thus, quotients containing fewer cubes than possible may get selected. In the first step *Substitute* uses the literal returned by the *IntroduceGates* routine, which corresponds to the output vertex of a subcircuit implementing P. The second step of the routine implements substitutions of finer granularity. Substitution of gate functions in this step is performed for each of the vertices in their topological order.

Note that substitution using the distributive law only does not require step (1), since it is subsumed by step (2). We should also point out that the division under the used properties may not be unique. This is illustrated in the example below, where annihilation gives rise to two distinct decompositions:

Example 3.2 Suppose $f = abc'e' + a'bcd + ac'de' + a'cde$, and let $P = ac'e' + a'cd$ be a divisor of f. We then can have two different quotients $q_1 = b + c'd + ce$ and $q_2 = b + ad + a'e$. Thus either substitution $f = y_v(b + c'd + ce)$ or

is feasible. ■

IV. An M32 Synthesis Example

Synthesis of a full adder will be used to as an example of the M32 system. M32 first reads the functional specification

$$z = a \oplus b \oplus c = a'b'c + a'bc' + ab'c' + abc \tag{4}$$

$$c_{out} = ab + ac + bc \tag{5}$$

of the circuit to be synthesized. M32 expects it to be in the two level `pla` format [31]. Thus, $F = \{z, c_{out}\}$ is the set of unimplemented vertices to be synthesized. The algorithm then selects function z since it has more literals then c_{out}. A subexpression P selected from z by the $GenerateDivisor$ routine is $a'b + ab'$. *GenerateDivisor* selects this divisor since it has the least cost, with the assumption that signal c arrives later than two other input signals a and b. This expression is then implemented using *IntroduceGates* by introducing gates y_1, y_2, and y_3 through the following sequence of transformations of P:

$$P \equiv a'b + ab' \xrightarrow{2} y_1' + ab' \xrightarrow{2} y_1' + y_2' \xrightarrow{1} y_3 \tag{6}$$

where a number above each arrow indicates a matching meta rule from Figure 4.

The *Substitute* function would then substitute y_3 for $ab' + a'b$ (complement substitution is also tried) in both z and c_{out}, giving:

$$z = cy_3' + c'y_3 \tag{7}$$

$$c_{out} = ab + cy_3. \tag{8}$$

Note that $ab' + a'b$ is not an algebraic divisor of $ab + ac + bc$. Therefore substitution based on weak division alone would not bring any changes to c_{out}. The *Substitute* routine makes substitution in c_{out} possible due to the cube reduction based on the sharp product operation. The M32 system detects that the division become possible if cubes ac and bc are reduced to $a'bc$ and abc respectively. This is a valid transformation since the result of $ac\#ab'c$ and $bc\#a'bc$, which is in both cases cube abc, is covered by the cube ab. It is now easy to see that $a'b + ab'$ divides expression $ab + ab'c + a'bc$, representing the same carry-out function c_{out}: $ab + (ab' + a'b)c$.

The next step in the *Substitute* routine is to see if any of the local functions of vertices y_1, y_2 or y_3 can be used to re-express either z or c_{out}. No further substitutions are possible in this case. This completes the first iteration of the algorithm in Figure 3.

On the next iteration of the loop either the unimplemented vertex for c_{out} or z can be selected, since both of their functions have same number of literals. Figure 5 depicts execution of the algorithm with the assumption that c_{out} is selected. c_{out} is completely implemented on this iteration of the algorithm through the following sequence of transformations:

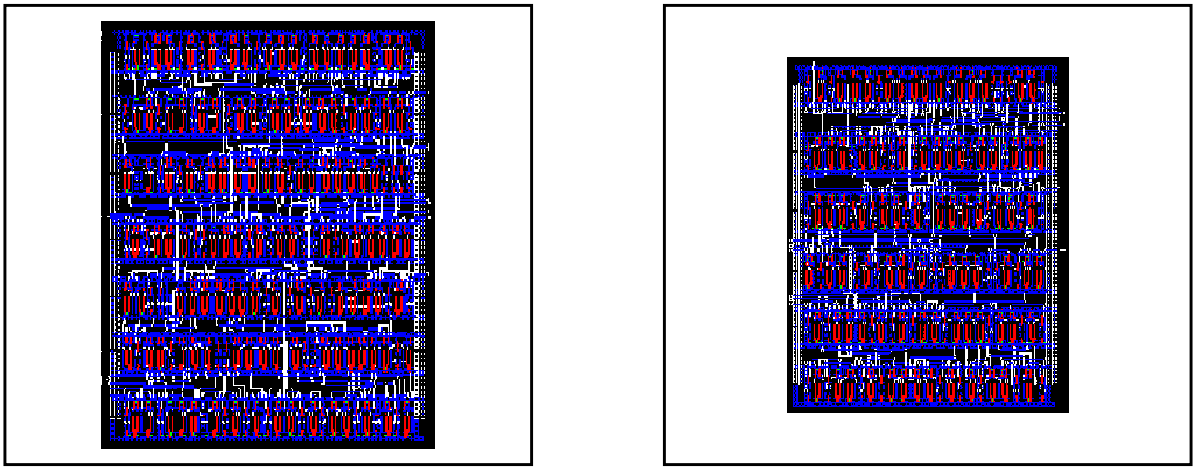
$$c_{out} \equiv ab + y_3c \xrightarrow{1} y_4' + y_3c \xrightarrow{1} y_4' + y_5' \xrightarrow{1} y_6. \tag{9}$$

The final implementation of the circuit has 9 NAND2 gates and 4 inverters, which is one NAND2 fewer than the equivalent SIS-1.2 implementation.

V. Experimental results

We evaluated the performance of M32 by synthesizing a set of circuits selected from the MCNC benchmarks [31] and comparing the results against SIS-1.2 [13]. The benchmarks were first minimized using ESPRESSO [5] prior to multilevel synthesis in M32 or SIS-1.2. We used the `delay` script in SIS-1.2 which is based on the *clustering script* proposed in [29] and is targeted towards technology-independent minimization of circuit delay. Both systems used the minimal gate library $L = \{wire, INV, NAND2\}$, and delay of the implementations obtained by both systems was estimated using the *library* delay model of SIS-1.2 and parameters from the `mcnc.genlib` library.

Table I compares the generated circuits in terms of the number of gates, levels of logic, topological complexity and esti-



SIS-1.2 (script.delay) M32

Fig. 6. `Cordic` relative layout areas (shown to the same scale)

mated pre-layout delay. The results in this table suggest the following observations:

- Even though minimization of gate count is not a primary objective in the M32 system, it generates implementations with fewer gates in all but two cases. In some cases the reduction in gate count is almost 50%.
- M32-generated circuits have consistently fewer logic levels, in several cases being almost half as deep as SIS-generated circuits.
- The topological complexity of M32-generated circuits is consistently lower than that of SIS-generated circuits. The topological complexity however, should not be interpreted as a final judge of circuit quality. Our experiments in Figure 1 were performed on synthetic benchmarks which belong to a restricted class of topologies, and which very significantly in their topological complexity. Typically, the average topological wire length of practical circuits does not vary as much as it does in Figure 1.
- The pre-layout circuit delays of M32-generated circuits are consistently lower than those of SIS-generated circuits, the average improvement in delay being about 30%.

The run times of M32 were comparable to or better than those of SIS-1.2 for all benchmarks suggesting that the use of more powerful decompositions and substitutions is computationally feasible.

To get a better indication of synthesis quality, the netlists produced from M32 and SIS-1.2 were laid out using the Epoch standard cell place and route tools [9] from Cascade Design Automation. The layouts were generated using cells in a $0.5\mu m$ CMOS process with two layers of metal, and allowing over-cell routing. I/O pins were distributed around the perimeter of the standard cell block. Delays were computed using the Epoch static timing analyzer TACTIC. These results, shown in Table II, indicate a 23% average improvement in total area, routing length and post-layout delay for M32-generated circuits. The layouts generated for a representative circuit, the `cordic` benchmark, are shown in Figure 6.

The `cordic` benchmark is also used, in Figure 7, to highlight the constructive nature of M32's synthesis algorithm and to illustrate its ability to dynamically adjust the implementation topology. The two variants shown in the figure were forced to diverge after the 28 iteration of the synthesis loop (1/3 of total iterations for the Figure 7 (a) solution). The gates marked with ● in both variants correspond to the common portion of the implemented schematics. The implementation in part (a) of the figure corresponds to the results shown in Table I and Table II, and reflects the incorporation of topological complexity constraints. The implementation in part (b) was generated by relaxing these constraints after the 28 iteration. This incremental synthesis capability can prove invaluable when the generated netlists marginally fail to meet specifications and must be fine tuned in the neighborhood of a given solution.

The last experiment was designed to assess the effect of using a richer gate library on the quality of the generated circuits. Since the only library that is currently supported by our prototype implementation of M32 is the simple NAND2/INV library, we had to resort to a less-than-ideal work-around to generate circuits based on other libraries. Using the SIS-1.2 `map` command, the NAND2/INV circuits produced by M32 were technology-mapped to the `mcnc.genlib` library. The

TABLE I: Pre-layout synthesis results

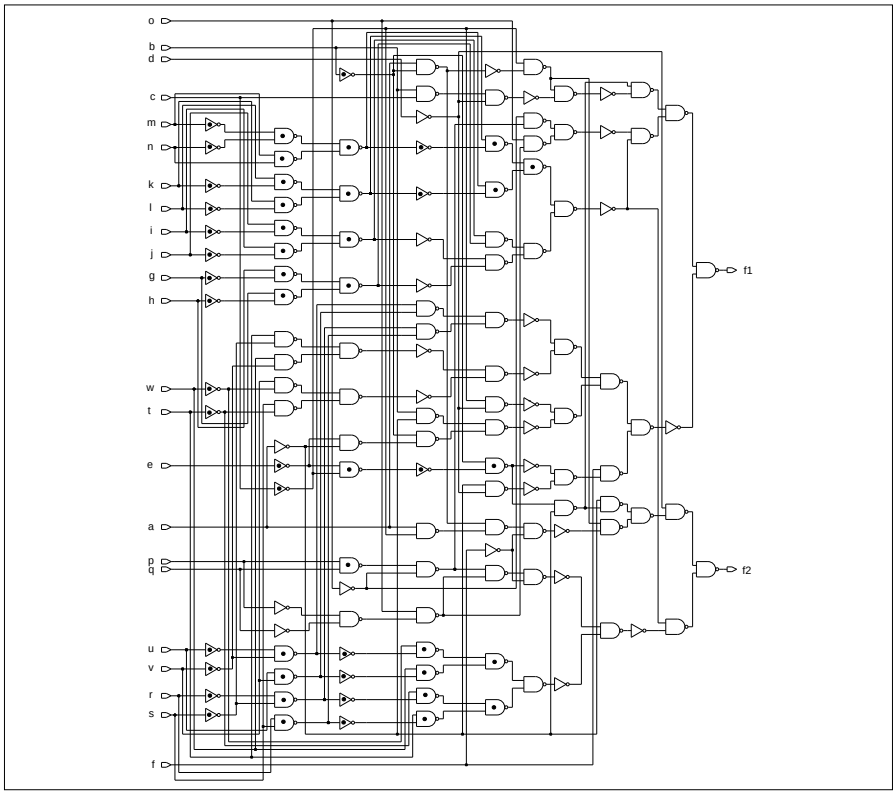
Circuit Name				SIS-1.2				M32				Norm.
	Inp.	Outp.	Cubes	Gates	Levels	Compl.	Delay	Gates	Levels	Compl.	Delay	Delay
z4ml	7	4	59	75	12	1.96	17.6	44	8	1.55	11.6	0.65
vda	17	39	793	1573	29	2.48	62.3	1115	16	1.56	34.3	0.55
inc	7	9	42	152	20	2.20	30.2	133	10	1.55	17.2	0.56
count	35	16	184	311	17	2.55	24.1	201	13	2.19	21.1	0.87
ldd	9	19	70	139	13	1.96	18.9	110	10	1.84	15.4	0.81
b9	41	21	141	176	12	1.69	17.8	177	10	1.60	14.8	0.83
ex4	128	28	620	690	20	1.73	28.0	665	15	1.43	19.3	0.68
cordic	23	2	1180	182	18	1.65	23.4	133	11	1.39	14.6	0.62
cps	24	109	654	2162	35	2.52	61.9	1625	17	1.93	41.3	0.66
duke2	22	29	120	743	20	2.17	35.0	534	13	1.68	24.8	0.70
vg2	25	6	110	239	16	1.88	21.6	152	11	1.36	15.5	0.71
apex2	39	3	438	564	33	2.92	44.6	484	18	1.83	25.4	0.56
sqrt8	8	4	88	79	14	1.84	18.3	76	12	1.72	17.6	0.96
bw	5	28	110	232	16	1.91	28.9	206	9	1.58	18.3	0.63
clip	9	5	167	240	26	2.55	31.3	309	14	1.75	17.5	0.55

TABLE II: Post-layout synthesis results

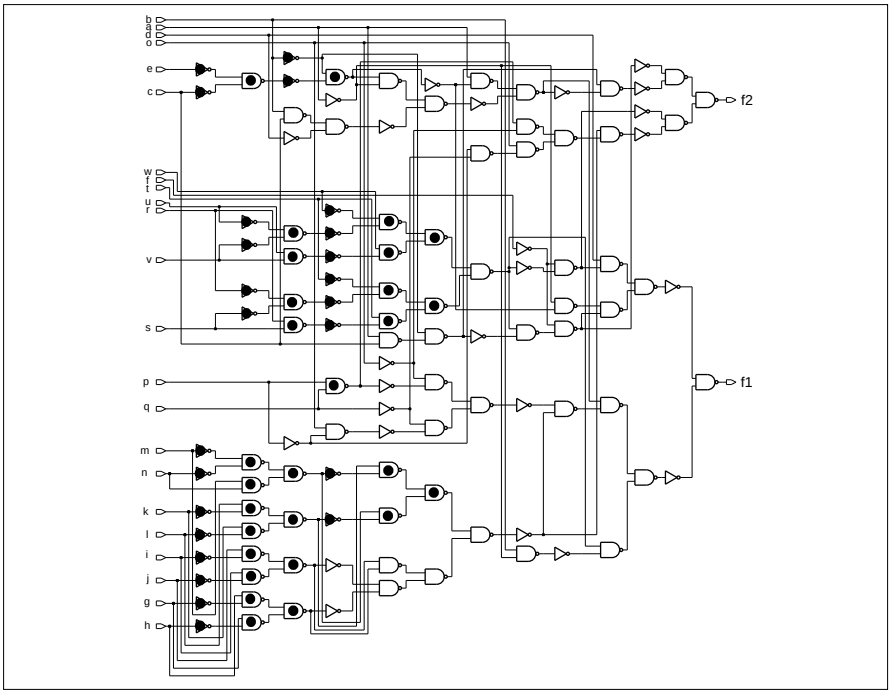
Circuit Name	SIS-1.2			M32			Improvement M32 / SIS-1.2		
	Total area (mil^2)	Routing length (μm)	Delay (ns)	Total area (mil^2)	Routing length (μm)	Delay (ns)	Total area	Routing length	Delay
z4ml	33.59	6592.0	2.66	21.55	4212.2	1.81	0.64	0.63	0.68
vda	1159.20	326400.4	7.45	715.642	189921.6	5.87	0.61	0.58	0.78
inc	77.58	17041.5	4.21	62.24	14039.6	2.52	0.80	0.82	0.59
count	137.7	28675.4	3.63	81.51	16941.6	3.62	0.59	0.59	0.99
ldd	60.73	12220.2	2.96	48.09	9123.3	2.18	0.80	0.74	0.73
b9	74.95	15136.6	2.88	76.93	15519.5	2.10	1.02	1.02	0.72
ex4	311.72	66272.5	4.16	294.92	60746.6	3.15	0.94	0.91	0.75
cordic	70.82	13996.6	3.55	54.59	10751.4	1.98	0.77	0.76	0.55
cps	1414.4	393693	8.19	1032.2	271116	5.74	0.72	0.68	0.70
duke2	414.28	106240.6	4.87	273.10	65001.5	3.80	0.65	0.61	0.78
vg2	100.47	20847.5	3.20	63.17	12526.6	2.15	0.63	0.60	0.67
apex2	283.02	68251.2	6.89	237.54	54155.8	4.67	0.90	0.79	0.67
sqrt8	34.44	7245.9	2.74	32.14	6640.8	2.41	0.94	0.91	0.87
bw	111.25	24798.6	3.71	99.46	22616.8	2.46	0.89	0.91	0.66
clip	116.79	25814.6	5.87	156.59	34502.7	3.71	1.34	1.33	0.63
						Average Improvement:	0.81	0.79	0.71

same mapping process was also applied to the NAND2/INV circuits generated by SIS-1.2. The results of this experiment are shown in Table III. The columns labeled “Area” and “Delay” record, respectively, the active areas and circuits delays of each implementation as reported by the mapper.

An examination of these results indicates that, overall, the circuits produced from M32 are still faster and smaller than those produced from SIS-1.2. However, the improvement is not as pronounced as it was for the NAND2/INV library. This outcome is hardly surprising since the above mapping process is decidedly antithetical to the constructive synthesis philosophy of M32 and undoes many of its gains. Specifically, mapping by tree covering is ill-suited to a highly optimized DAG



a) Synthesis with topological constraints: 133 gates, 11 levels



b) Synthesis with relaxed structural constraints:116 gates, 13 levels

Fig. 7. Two incrementally-different implementations of the `cordic` circuit

and we conjecture that exact DAG covering may have produced better results. It must be noted, however, that DAG covering is notoriously difficult and no published algorithms that can effectively handle large circuits have been demonstrated.

TABLE III: Results from mapping to a richer technology library

Circuit	SIS				M32			
	NAND2/INV		mcnc.genlib*		NAND2/INV		mcnc.genlib*	
Name	Area	Delay	Area	Delay	Area	Delay	Area	Delay
z4ml	142	17.6	131	12.0	84	11.6	85	11.2
vda	2731	62.3	2188	21.6	1870	34.3	1558	20.1
inc	268	30.2	223	13.8	236	17.2	189	10.8
count	521	24.1	394	14.4	340	21.1	277	17.2
ldd	237	18.9	198	12.7	194	15.4	167	12.1
b9	280	17.8	224	12.6	297	14.8	261	10.8
ex4	1149	28.0	888	18.5	1078	19.3	843	15.6
cordic	301	23.4	244	17.2	216	14.6	163	11.1
cps	3641	61.9	2862	29.5	2753	41.3	2368	20.0
duke2	1263	35.0	1039	18.6	898	24.8	779	15.6
vg2	394	21.6	297	12.9	239	15.5	174	14.1
apex2	963	44.6	797	28.9	797	25.4	649	21.5
sqrt8	133	18.3	111	12.4	130	17.6	110	11.5
bw	395	28.9	333	14.3	349	18.3	328	11.3
clip	413	31.3	338	23.9	521	17.5	437	15.7

* Technology mapping was done using the SIS-1.2 command `map -s -n 1 -AFG -p`

We believe that a more natural approach for solving this problem is the extension of the M32 algorithm to handle arbitrary gate libraries directly. This effort is currently underway.

VI. Conclusions and Future Work

The M32 synthesis approach outlined in this paper is a promising alternative to conventional multi-stage logic synthesis algorithms. Initial results from a prototype implementation are encouraging and suggest that further exploration of this method is worthwhile. We are currently examining a number of extensions and variants including:

- Experimentation with other structural complexity metrics
- Support of arbitrary gate libraries
- Synthesis of partially-specified functions
- Exploration of other functional representations, such as BDDs, to enable more powerful Boolean transformations

Ultimately, we would like to integrate physical optimization (placement and routing) with logic synthesis for better management of interconnect effects on both area and delay.

References

[1] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot. Multilevel synthesis minimizing the routing factor. In *27th Design Automation Conference*, pages 365–368, June 1990.

[2] K. Bartlett, W. Cohenand A. de Geus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Trans. CAD IC*, CAD-5, 1986.

[3] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] R. Brayton, E. Detjens, S. Krishna, P. McGeer, T. Ma and L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli. Multiple-level optimization system. In *IEEE Int. Conf. on CAD (ICCAD)*, Santa Clara, CA, November 1986.

[5] R. K. Brayton, G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli. A comparison of logic minimization strategies using ESPRESS0. In *Proc. IEEE International Symposium on Circuits and Systems*, pages 42–48, May 1982.

[6] R. K. Brayton and C. McMullen. Synthesis and optimization of multistage logic. In *Proc. Int. Conf. on Comp. Des. (ICCD-84)*, pages 23–28, Rye, 1984.

[7] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers for VLSI Synthesis, 1984.

[8] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 6:1062–1081, November 1987.

[9] *EPOCH User's Manual, ver. 3.2*. Bellevue, WA 98006, 1995.

[10] E. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Transactions on Computers*, C-18(12):1098–1109, December 1969.

[11] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Proceedings International Conference on Computer-Aided Design*, pages 116–119, Santa Clara, CA, November 1987.

[12] D. L. Dietmeyer. *Logic design of digital Systems*. Allyn and Bacon, Boston, MA, 1978.

[13] E. M. Sentovich et. al. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, UC Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[14] J. P. Fishburn. LATTIS: An iterative speedup heuristic for mapped logic. In *Proc. 29st ACM/IEEE Design Automation Conference*, pages 488–491, June 1992.

[15] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: A system for automatically synthesizing and optimizing combinational logic. In *Proceeding of the 23rd Design Automation Conference*, pages 79–85, 1986.

[16] G. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On properties of algebraic transformations and the synthesis of multifault-irredundant circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11(3):313–321, March 1992.

[17] K. Keutzer. DAGON: technology binding and local optimization by DAG matching. In *Proc. 24rd Design Automation Conf.*, pages 341–347, June 1987. Reprinted in 25 Years of Electronic Design Automation.

[18] K. Keutzer, A. R. Newton, and N. Shenoy. The future of logic synthesis and physical design in deep-submicron process geometries. In *ISPD'97*, pages 218–224, 1997.

[19] P. Kurup and T. Abbasi. *Logic Synthesis Using Synopsys*. Kluwer Academic Publishers, 1997.

[20] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. In *Proc. Int. Conf. Computer Design*, pages 263–271, 1995.

[21] F. Mailhot and G. D. Micheli. Technology mapping using boolean matching. In *European Design Automation Conference*, pages 180–185, March 1990.

[22] P. McGeer, R. Brayton, and A. Sangiovanni-Vincentelli. Performance enhancement through the generalized bypass transform. In *IEEE Int. Conf. on CAD (ICCAD)*, pages 184–187, November 1991.

[23] M. Pedram and N. Bhat. Layout driven technology mapping. In *Proc. 28th Design Automat. Conf.*, pages 99–105, June 1991.

[24] G. Saucier, J. Fron, and P. Abouzied. Lexicographical expressions of boolean functions with applications to multilevel synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 12:1642–1654, November 1993.

[25] H. Savoj, M. J. Silva, R. K. Brayton, and A. Sangiovanni-Vincentelli. Boolean matching in logic synthesis. In *IEEE International Conference on Computer Aided Design*, pages 168–174, 1992.

[26] P. R. Schneider and D. L. Dietmeyer. An algorithm for synthesis of multiple-output combinational logic. *IEEE Trans-*

actions on Computers, C-17(2):117–128, February 1968.

[27] K. Scott and K. Keutzer. Improving cell libraries for synthesis. In *The Proceedings of the Custom Integrated Circuits Conference*, pages 721–724, 1994.

[28] K. J. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *Proc. 27st ACM/IEEE Design Automation Conference*, June 1990. 357-360.

[29] H. J. Touati, H. Savoj, and R. K. Brayton. Delay optimization of combinational logic circuits and partial collapsing. In *Proc. 28th Design Automat. Conf.*, pages 188–191, June 1991.

[30] H. Vaishnav and M. Pedram. Minimizing the routing cost during logic extraction. In *32nd ACM/IEEE Design Automation Conference*, 1995.

[31] S. Yang. *Logic synthesis and optimization benchmarks user guide – version 3.0*. Microelectronics Center of North Carolina, Research Triangle Park, NC, January 1991.

[32] K. Yoshikawa, H. Ichiryu, H. Tanishita, S. Suzuki, N. Nomizu, and A. Kondoh. A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between. In *Proc. 28st ACM/IEEE Design Automation Conference*, pages 112–117, June 1991.