DAGON: Technology Binding and Local Optimization by DAG Matching

Kurt Keutzer
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Technology binding is the process of mapping a technology independent description of a circuit into a particular technology. This paper outlines a formalism of this problem and offers a solution to the problem in terms of matching patterns, describing technology specific cells and optimizations, against a technology independent circuit represented as a directed acyclic graph. This solution is implemented in *DAGON*. *DAGON* rests on a firm algorithmic foundation, and is able to guarantee locally optimal matches against a set of over three thousand patterns. *DAGON* is an integral part of a synthesis system that has been found to provide industrial quality solutions to real circuit design problems.

1. Introduction

† We begin our treatment of the problem of technology binding with local optimizations by outlining a formalization of it. Using this formalization we will find that the problem is related to those that have been encountered in the field of programming language compilers. Indeed, we claim that compiler techniques are highly relevant to many problems in logic synthesis. This thesis was originally stated in [Jo82] and in [TJB86]. In particular here we claim that technology binding for logic synthesis is a very closely related problem to code generation for programming language compilers. More specifically, matching a graph-like description of a technology independent circuit against a library of patterns in a technology, such as a standard cell library, is similar to matching a graph-like intermediate representation of a computer program against the patterns of an instruction set of a given machine. Thus *twig* [Tj86], a tree manipulator used for constructing code generators for programming language compilers is used to build an optimizing technology binder. The result is a technology binder, *DAGON*, that is capable of optimizing for time, area or a function of both. Because *DAGON* uses data abstraction, and a modular technology pattern description format, *DAGON* is easy to "port" to new

technologies. *DAGON* rests on a firm algorithmic foundation, and is able to guarantee locally optimal matches against a set of over three thousand patterns.

2. The Formalism

Before going into how compiler techniques are used in *DAGON* we will first take a step back to view the problem of technology binding and local optimizations under a broad formalism. In this paper we assume that the reader is familiar with basic tree and graph terminology, as described, for instance, in [AHU76]. Any set of boolean functions with common sub-expressions can be viewed as a directed acyclic graph (DAG) in which the labels of the vertices in the graph are the boolean operators AND and OR and edges are labeled either 1, for the true value of the signal associated with the vertex, or 0, for the inverted value. We may think of this DAG as a circuit with edges directed from outputs to inputs. We present here the different stages of the technology binding process.

1. A technology independent graph $G_i = (V,E)$ such that each v_i in V, label$(v_i) \in \{AND,OR\}$ and each e_i in $E \in \{0,1\}$.
2. A technology independent graph $G_d = (V,E)$ such that each v_i in V, label$(v_i) \in \{AND,OR\}$, the outdegree (fanin) of each v_i is less than some technological limit, and each e_i in $E \in \{0,1\}$.
3. A canonically represented technology independent graph $G_c = (V,E)$ such that each v_i in $V \in \{NAND,NOT\}$ the outdegree of each v_i in V is less than some technological limit. Edges have no labels.
4. A technology bound graph $G_t = (V,E)$ such that for each v_i in V label$(v_i) \in \{$ technology pattern set (e.g. gates in a standard cell library) $\}$. The outdegree of each v_i in V is less than some technological limit. Edges have no labels.

It is assumed that common subexpressions have already been discovered by a global optimization technique such as [BrMc84] and are reflected in G_i. The translation from G_i to G_d is called decomposition. Decomposition begins the binding into technology by realizing fan-in limitations, but also considers non-local transformations within a single boolean function. We will not treat this problem here, but our work on this problem will be presented in [KLV87]. The translation from G_d to G_c is a direct process. In this paper we concern ourselves with the problem of the translation from G_c to G_t. We treat this problem as one of as DAG covering or a DAG rewriting. A precise formalization of the notion of DAG covering or DAG rewriting is beyond the scope of this paper, but the formalizations for trees presented in [HoOD82] and [AhGa85] may be extended to

DAGS. In taking a covering approach to the problem of technology binding, we have purposely moved all non-local optimizations up into global optimization and decomposition stages.

Having abstracted away the domain specific details of the problem, we may now observe that problem of covering a DAG is also encountered in a programming language compiler's code generation for language expressions with common subexpressions [AJU77]. Unfortunately, [BrSe76] and [AJU77] show that this problem is NP-complete and to date no efficient technique for attacking this problem directly has been realized. However, as the problem of code generation is an essential one, practical ways of approaching the problem have been found. An attractive approach is to treat the problem of matching a DAG as the problem of matching against a forest of trees which compose it [RSU86]. This is the approach followed in *DAGON*.

3. The DAGON Approach

DAGON takes a canonical technology-independent description of a combinational circuit (G_c) and a list of patterns describing both the cells in the technology and local transformations. *DAGON* creates a technology bound circuit (G_t) by partitioning the circuit into a forest of trees, then using a tree pattern matching automaton to match the individual trees. The tree matcher that is used in *DAGON* is based on *twig* [Tj86], a tree matching generator tool that is generally used for constructing code generators for programming language compilers. Figure 1 gives an overview of the *DAGON* approach. These aspects are described in more detail below.

3.1 Partitioning

The technique used here for partitioning is simply to make each node (gate) with indegree (fanout) greater than one, the root of a new tree. This process requires only time linear in the size of the DAG. The size of these resultant trees in the forest depends very much on the nature of the circuit. If the circuit has a great deal of fanout, then there will probably be many small trees after partitioning. On the other hand if the original circuit has more of a tree-like structure, then the circuit will be partitioned into a few large trees. Figure 2 shows the partitioning of a small circuit into trees. X's mark partition points.

3.2 Tree Matching

After partitioning, *DAGON* proceeds to find minimal cost matches or coverings of the partitioned trees against the technology patterns. *DAGON* guarantees an optimal match in the entire tree, how ever many levels of logic it may contain, and in this way avoids the pitfalls of a greedy approach or any other method that must limit its search to some fixed number of levels. To accomplish this the *twig* tree manipulation program [Tj86] is used. While for a user of *DAGON*, and even for a developer of a rule set for *DAGON*, *twig* may be treated as a "black box", we will briefly describe how *twig* works in this application. There are two key elements in finding a minimal cost match for a tree. The first is to identify the set of candidate matches over the tree. The second is to identify the minimal cost match from among the candidates. We shall describe each of these, beginning with how patterns are described in *DAGON*.
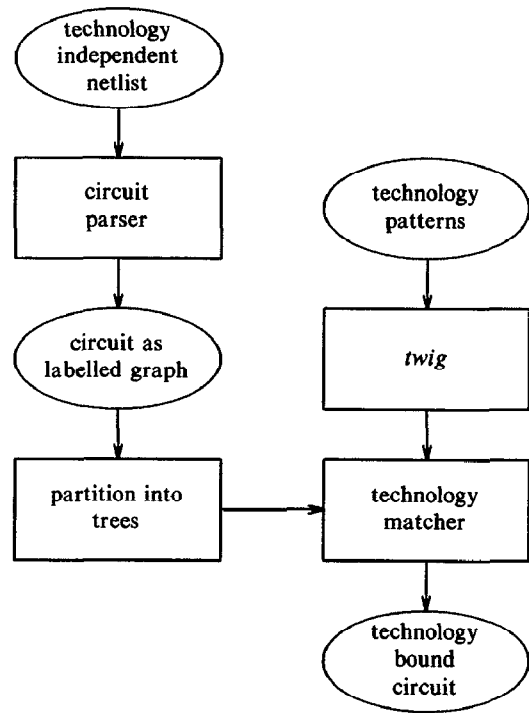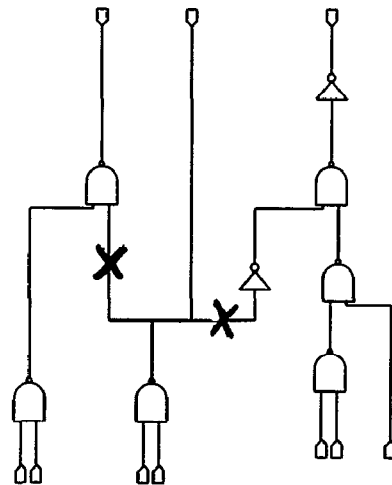

Figure 1. The DAGON Approach

Figure 2. The Partioning the Circuit

3.2.1 Description of Patterns Patterns in *DAGON* are small trees in the canonical NAND/NOT form. These trees correspond either to a direct mapping into a cell in the target technology or may describe a mapping that contains a local transformation. These patterns are fed directly to *twig* and consist of three parts. The first part describes the pattern in a grammatical form. An important feature of the grammatical description is that is able to describe not just a single fixed pattern but a whole family of patterns instances. In Figure 3 we see a group of patterns describing a family of AND-OR-INVERT (AOI) gates. The five patterns given in Figure 3 actually describe sixty-four unique pattern instances from an AOI444 to an AOI211. Many of these patterns are symmetric, thus an AOI114 is equivalent to, and would be output as, an AOI411. A graphical representation of the family of trees described by these patterns is given in Figure 4.

The second part of a pattern is the cost evaluation part. This is part of the *twig-DAGON* interface. A person developing a pattern set for *DAGON* using *twig* gives a cost for each pattern in the reserved variable "cost". The type of "cost" (in *DAGON* it is an array) and the functions for evaluating it (routines for adding costs and comparing costs) must also be defined. The pattern-set developer inserts code in this portion that computes the cost of a subtree that has this pattern at its root and puts the result in the variable "cost". Given a tree to be matched *twig* uses these costs, and the routines which manipulate them, to evaluate the cost for candidate matches.

In Figure 3 the cost is computed at the root of the AOI tree once the lower level patterns have been bound to their specific values. This cost consists of the costs of all the subtrees below this pattern plus the cost of the AOI gate itself. The cost of all the subtrees below this pattern is also the default cost and is represented by the DEFAULT_COST. To maintain modularity time (T) and area (A) costs are computed by functions that are loaded based on the target technology.

The third part of a pattern is of two kinds: either rewrite or action. In Figure 3 if there is a match of the entire AOI gate, the action is to print the appropriate gate.

This format is very convenient and makes adding and modifying patterns, costs, and actions quite easy.

3.2.2 Size of Pattern Set DAGON uses 52 parameterizable patterns. These parameterizable patterns expand into over seven hundred unique pattern instances representing various permutations of AT&T's standard cell library of approximately 165 gates, as well as a few thousand unique local optimization patterns.

3.2.3 Tree Matching Given the technology patterns as described above, *twig* builds a tree pattern-matching automaton that will indicate for each node *n* in the subject tree all the technology patterns which have a match in the subject tree rooted at *n*. A straightforward approach to this problem is to simply traverse the tree trying each pattern at each node. Even this naive approach takes only time $O(TREE\_SIZE \times PATTERN\_SET\_SIZE)$. *Twig* uses a more sophisticated approach based on the Aho-Corasick [AhCo75] algorithm. The Aho-Corasick algorithm is a string matching

```
/**** AOIxxx canonically expressed as
                      not
                       |
                     nand_3
          |            |          |
        nand         nand       nand          */
eqn:   not(nand_3(inand,inand,inand))
/*$$ refers to the root of the pattern*/
{
DEFAULT_COST;  /*sum up cost of children*/
cost.cost_a[AREA]+=d_get_aoi_area_cost($$);
cost.cost_a[TIME]+=d_aoi_time_cost($$)
cost.cost_a[AT_K]
  += my_pow(cost.cost_a[TIME],cost_power)
     *cost.cost_a[AREA];
}
={
 d_print_aoi(stdout,$$);
};
inand:  nand_2(eqn,eqn)
{} /*default cost is the sum of costs of the
children-it doesn't have to be spelled out*/
={};/*action taken at root of pattern tree*/
inand:  nand_3(eqn,eqn,eqn)
{}/*default cost is the sum of children*/
={};/*action taken at root of pattern tree*/
inand: nand_4(eqn,eqn,eqn,eqn)
{}/*default cost is the sum of children*/
={};/*action taken at root of pattern tree*/
inand: not(eqn)
{}/*default cost is the sum of children*/
={};/*action taken at root of pattern tree*/
```

Figure 3. An AOIxxx Pattern

algorithm, but can be applied to trees because trees can also be described as strings. This approach can reduce the matching time to $O(TREE\_SIZE)$. For further details see [AhCo75] and [AhGa85].

At the end of this process a search space which contains all possible tree matches has been created. Figure 5 shows a subcircuit of Figure 2 with nodes labeled with all candidated matches for a small pattern set. What remains is to search this space and find the optimal match.

3.2.4 Finding a Minimal Cost Match Given the set of all possible matches, as determined by the procedure described in the previous section, it now remains to find a match of least cost. Dynamic programming, based on the approach of [AhJo76], is used. A dynamic programming, or divide-an-conquer, approach is to find a least cost match of a tree T with root r and subtrees $T_1,...,T_n$ by first finding least cost matches for each of the subtrees. This procedure is applied in a recursive depth-first manner. As the leaf of a tree can only be a NOT or a NAND, it is easy to determine the least cost match. At the root r of an arbitrary subtree T, the process consists of simply trying each pattern p_i that has matched at r. The leaves of the pattern p_i will extend into the subtrees of T. The least cost match for each of these subtrees has already been computed by the bottom up procedure, so the cost of matching p_i at r can be computed in terms of the cost of p_i together with the cost of each of the subtrees with roots at the leaves of p_i. The number of patterns matching at a node is bounded by the total number of patterns, so the time bound for finding an optimal match is
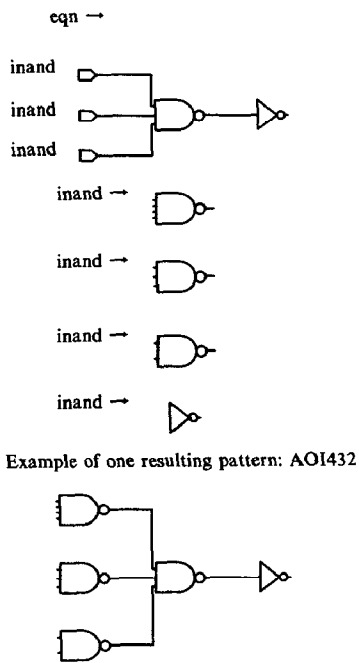
eqn →

inand
inand
inand

inand →

inand →

inand →

inand →

Example of one resulting pattern: AOI432

Figure 4. Geometric Interpretation of Figure 3
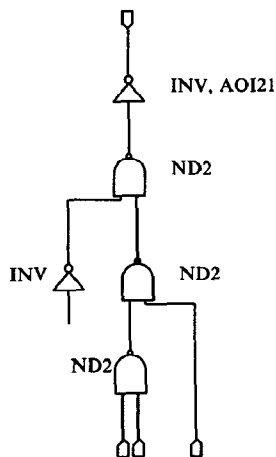
INV, AOI21

ND2

INV   ND2

ND2

Figure 5. Tree After Matching

$O(TREE\_SIZE \times PATTERN\_SET\_SIZE)$.

Figure 6 shows the discovery of a least cost match among candidated matches. There are no choices for a least cost match of the tree until we reach the inverter at the root (node A). Costs

are in terms of grids, a measure of the width of a standard cell. At A we have two choices, an inverter or a complex AOI21 gate. The cost of match of an inverter would be the cost of the inverter (2 grids) plus the cost of the subtree rooted at B (11 grids) . The match of the AOI21 is the cost of the AOI21 (4 grids) plus the cost of the subtrees at the leaves of it's patterns: two inputs (of cost 0 grids) and the ND2 at E (3 grids). The
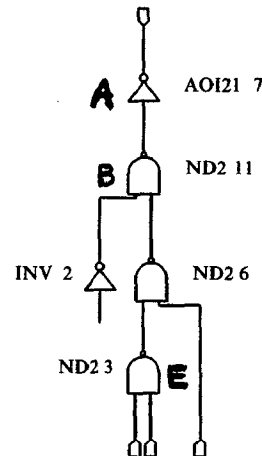
A   AOI21 7

B   ND2 11

INV 2   ND2 6

ND2 3   E

Figure 6. Finding a Least Cost Match

AOI21 is an easy winner.

3.3 Quality of Tree Matching

Matching against a forest of trees rather than against the original DAG loses the guarantee of a globally optimal solution. But as mentioned above, finding a globally optimal solution to DAG matching or covering is an NP-complete problem. However, the simple observation that optimal tree matching is realizable in linear time and optimal DAG matching is NP-Complete shows that there is a considerable range between these two classes of problems. There is a very general optimization question involved as to whether it is better to optimally solve a subproblem or to suboptimally solve the whole problem. Greedy approaches to the problem of covering DAG's are considered in [AJU77] and are shown to be very suboptimal. A bottom-up greedy approach is used to match against high-level modules in the silicon compiler described in [Ka86]. In the following section we discuss our own experience with the limitations of tree matching as well as our answers to them.

3.4 Enhancements

Our experience shows that a tree matching approach that uses only information local to a tree will not give satisfactory results. To improve the quality of result a few extensions to tree matching were required. For each node in the tree to be matched, attributes are attached which are describe properties that are not local to the tree. The relevant attributes are *fanout*, *inverse*, and *signal*. *Inverse* tells if the inverted sense of a signal is available somewhere in the circuit. This is very useful because

drastic simplifications can often be made if the inverted sense of a signal can be used rather than the signal itself. *Signal* gives the name of a signal. This can be used together with *inverse* to represent a non-tree pattern such as an Exclusive-Or. *Fanout* is used to determine timing costs.

Another important advance in quality came when *DAGON* began to be applied to sequential circuits. To handle this *DAGON* had to be able to match against general directed graphs as well as DAG's. To accomplish this the partitioning procedure had to be enhanced to make each sequential element the root of a new tree. One advantage of matching against sequential circuits is that optimization patterns which use the both the inverted and non-inverted (Q and Q_bar) outputs of a flip-flop can be described. This class of optimization has been very important in producing a quality binding of some circuits.

A final source of enhancement is the incorporation of a "peephole optimizer" that looks for redundant gates that may arise due to two local optimizations colliding across tree boundaries. With these enhancements *DAGON* has been able to produce a high quality binding of the circuit. When circuits bound by *DAGON* are examined by human designers, suggestions for improvements have always been related to the need for new optimization rules rather than problems related to any other of the limitations of an approach based on tree matching.

4. Other Features of DAGON

DAGON runs in time $O(DAG\_SIZE \times PATTERN\_SET\_SIZE)$. This speed allows it to be applied to industrial sized problems and can even be used interactively. It also allows DAGON to support a very large cell library, such as the one currently in use at AT&T, together with a few thousand optimizations.

In the present system the user may choose to bind the circuit in such a way that minimizes time (T), area (A) or some function $(A \times T^k)$ of both. This gives the user flexibility in choosing the nature of the target implementation.

Just as code generator-generators allow for ease of porting a compiler to a new machine, the use of a code generator-generator in *DAGON* allows for ease of porting to a new technology. Porting to a new technology requires introducing a new set of patterns, including the costing evaluators and the action routines. The patterns, as shown in Figure 3, are easy to revise. The costing evaluators and action routines are modular and access the circuit representation only through an abstract data access mechanism, so rewriting these portions is also straightforward.

5. Comparison With Other Approaches

A fair comparison with other approaches to technology is made difficult by the fact that technology binding plays a different role in each system. The model of synthesis that we described on the section 2 has not been widely accepted at this point. In addition, the comparisons in this section are based on this author's understanding based on the limited details published and

personal conversations, as acknowledged later.

5.1 Logic Synthesis System (LSS)

Among the first work on local optimization was an approach based on a sequence of local transformation rules applied heuristically [DJBT81]. Most of the optimizations described in [DJBT81], and in later papers on LSS, involve common subexpression elimination that in our present environment would be handled as part of the decomposition process. The class of non-local optimizations presented in [TJB86] would be handled by the global optimizer. While such a discussion is beyond the scope of this paper, it is the opinion of this author that common subexpression elimination for combinational logic is one problem that is best *not* handled by programming language compiler methods because restricted domain found in combinational circuits allows for the use of a method [BrMc84] that is able to find many more candidate sub-expressions than are efficiently detectable in the vastly more complex domain of programming language compilers.

As for local transformations techniques in general, we saw above that identifying the class of local transformation patterns that can be applied to a circuit is a straightforward problem and one that is efficiently solved by the Aho-Corasick algorithm. The important underlying problem is to identify which transformations to apply.

5.2 Functional Design System (FDS)

Only one portion of this synthesis system [DLT84] is devoted to random combinational logic synthesis and technology binding plays a rather small role in this system as a whole. The approach of the combinational logic synthesis portion is to first globally optimize and decompose the circuit. The resulting circuit is partitioned into trees and bound to a technology. Although the binding is limited to trees, the greedy algorithm used offers no guarantee of optimality. Local transformations also do not seem to play a significant role in this system.

5.3 SOCRATES

A rule-based approach to the problem of technology binding with local optimizations is used in SOCRATES described in [GeCo85]. The separation of the rules from the search strategy, as employed in this approach, greatly aids the ease of adding and changing rules, as well as changing to a new technology. This feature is also present in *DAGON*. The addition of time and area trade-offs is another important advancement introduced in SOCRATES. *DAGON* is also able to trade time for space by allowing the user to optimize some function $(A \times T^k)$ of time and area. *DAGON* does not have the advantage of allowing the user to give an upper bound on time, then minimizing with respect to area (or vice-versa).

While tree patterns and rules in a knowledge base are similar in many ways, SOCRATES and *DAGON* differ significantly in how these rules or patterns are applied. To avoid the expensive running time of a backtracking approach, *DAGON* applies its patterns to a circuit that has been partitioned into a forest of trees. This allows it to use a dynamic programming approach it is also able to look among all candidate coverings of a tree using

all rules are patterns and applying arbitrary look-ahead, in order to choose the candidate covering of the tree that is minimal in cost. Comparing run-times with those given in [GeCo85], it appears that *DAGON* is at least two orders of magnitude faster for similarly sized circuits.

In fairness to a backtracking approach, it is attempting to perform global optimizations that would not be addressed by *DAGON*. As SOCRATES approach to binding it is more general, it can potentially reach a global minimum, and has the advantage that the more time it is given, the better the results will be.

6. Current Synthesis Environment

DAGON is presently part of a highly automated synthesis environment that is producing designs that often exceed hand-crafted quality even on relatively small examples. Synthesis begins with CONES [SMP86] which takes a functional, simulatable [Fr84], C description of a circuit, and synthesizes it into a combinational and a sequential portion. The combinational portion is fed into the *mis* [BDKM86] multi-level optimizer.

Coordination between the global optimizer and the technology binder, which is the role of the decomposer, is critical. At present a technology driven decomposition process [KLV87] has been substituted into *mis*. *Mis* then outputs the circuit in a NAND gate realization. This does not mean that *mis* is prematurely binding the technology, only that the output is expressed in a canonical form. The representation of the circuit in a canonical form allows for a drastic reduction in the number of patterns required to cover a technology. This NAND form is then combined with the sequential portion of the circuit and fed to *DAGON* for binding to the target technology.

DAGON presently supports three sets of technology patterns: a small standard cell library of sixteen cells; the AT&T standard cell library that is coordinated with the LTX2 [Ch87] placement and routing package; and a larger library for Sc2 [Hi85], a gate matrix style automated layout program. This synthesis environment is presently in production use on an experimental basis.

7. Implementation

By relying on *twig*, *DAGON* has remained small and the entire *DAGON* system consists of approximately 4000 lines of C code, and another 700 lines of pattern descriptions. It presently runs on a VAX 8650 under UNIX 4.3BSD.

8. Results

The results of running *DAGON* on the de Geus benchmarks [Ge86], as well as larger examples are shown in Table 1. The target technology in these tables was the AT&T standard cell library. In this table statistics for *DAGON* are shown side-by-side with an NAND/NOT of the same logic. This is only intended to give a feeling for *DAGON's* individual contribution. The measurements are in terms of standard cell grids. To compare the grid measure with two-input NAND gate equivalents,

use the fact that a two-input NAND gate is three grids. The results on these benchmarks are an improvement over others published at last year's Design Automation Conference. Running times are on a VAX 8650 (about 6 times as fast as a VAX 11/780).

These results also support the claim that *DAGON* runs in time linear in the input size. This has ensured that *DAGON* can be used on large examples in a production environment.

These results show that a system such as *mis* together with *DAGON* gives the user an exciting ability to interactively explore a wide space of design trade-offs from the logic level all

Example	NAND/NOT		DAGON		runtime
	gates	grids	gates	grids	(sec.)
rd53	36	109	19	79	.1
9sym	66	202	44	163	.2
vg2	90	253	65	208	.2
rd73	90	275	47	196	.3
sao1	111	331	64	245	.4
sao2	111	370	61	273	.4
bw	171	479	118	380	.6
duke2	338	1013	251	851	1.2
gpio	400	1073	281	835	1.3
lmtc	908	2559	660	2078	3.1
pla4	1478	4338	1022	3465	5.6

Table 1. Results of Running DAGON

the way to layout.

9. Future Directions

While *DAGON* has been able to meet timing requirements without manual intervention on circuits to which it has been applied, *DAGON's* present ability to optimize for timing is highly limited by it's pattern set. A class of patterns which would allow for timing optimizations such as moving signals off of a critical path, add buffers, and so forth, would improve timing directed performance a great deal.

10. Summary

In this paper we have outlined three contributions to the area of technology binding and local optimization. The first is a formalism of the problem in terms of graph transformations. The second is an approach to the problem of graph transformations by DAG or graph matching and in term an approach to DAG matching by reducing it to the problem of (attributed) tree matching. These first two contributions have already reshaped the way that this problem is being approached. The third is the description of a working implementation of these ideas using the *twig* tree matcher that has proved to be of high enough quality to provide superior industrial designs as well as efficient enough in implementation to be used in a production environment.

11. Acknowledgements

12. References

[AHU76] A. Aho, J. Hopcroft, J. Ullman, Addison-Wesley Publishing Company, Third Edition pp.186-194.

[ASU86] A. Aho, R. Sethi, J. Ullman, Addison-Wesley Publishing Company, pp.557-584.

[AhGa85] A. Aho, M. Ganapathi, "Efficient tree pattern matching: an aid to code generation", "Conf. Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages", January 1985, 334-340.

[AhJo76] A. Aho, S. C. Johnson, "Optimal Code Generation for Expression Trees", 23, 3, 488-501 (1976).

[AJU77] A. Aho, S. C. Johnson, J. D. Ullman, "Code Generation for Expressions with Common Subexpressions", 24, 1, 146-160 (1977).

[BrMc84] R. Brayton, C. McMullen, "Synthesis and Optimization of Multi-Stage Logic", "Proc. of the ICCD", October 84, 23-28,

[BDKM86] R. Brayton, E. Detjens, S. Krishna, T. Ma, et. al., "Multiple-Level Logic Optimization System", "Proc. of the ICCAD", November 1986.

[BrSe76] J. Bruno, R. Sethi, "Code Generation for a One Register Machine", 23, 3, 502-510 (1976).

[Ch87] M. Chi, "An Automatic Rectilinear Partioning Procedure for Standard Cells", *to appear* "Proc. of the Design Automation Conference", June 87.

[DJBT81] J. Darringer, W. Joyner, C. L. Berman, L. Trevillyan, "Logic Synthesis Through Local Transformations", 25, 4, 272-280 (1981).

[DBGJ84] J. Darringer, D. Brand, J. Gerbi, W. Joyner, L. Trevillyan, "LSS: a system for production logic synthesis", 28, 5, 537-545 (1984). 1975.

[DLT84] J. Dussault, C-C. Liaw, M. Tong, "A High Level Synthesis Tool for MOS Chip Design", "Proc. of the 21st Design Automation Conference", June 1986, 308-314.

[Fr84] E. Frey, "ESIM: A functional level simulation tool", "Proc. of the ICCAD", November 84, 48-53.

[Ge86] A. J. de Geus, "Logic Synthesis and Optimization Benchmarks for the 1986 Design Automation Conference", "Proc. of the 23rd Design Automation Conference", June 86, 78.

[GeCo85] A. J. de Geus, W. Cohen, "A Rule Based System for Optimizing Combinational Logic" 2, 4, 22-32 (1985).

[GBGH86] D. Gregory, K. Bartlett, A. De Geus, G. Hachtel, "Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic", "Proc. of the Design Automation Conference", June 86, 79-85.

[Hi85] D. D. Hill, "Sc2: A Hybrid Automatic Layout System", "Proc. of the ICCAD", November 85, 172-174.

[HoOD82] C. Hoffman, M. O'Donnell, "Pattern Matching in Trees" 29, 1, 68-95 (1982).

[Jo83] S. C Johnson, "Code Generation for Silicon", "Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages", January 1983, 14-19.

[JTB86] W. Joyner, et.al., "Technology Adaptation in Logic synthesis", "Proc. of the 23rd Design Automation Conference", June 1986, 94-100.

[Ka86] M. Kahrs, "Matching a parts library in a silicon compiler", "Proc. of the ICCAD", November 1986.

[KL87] K. Keutzer, M. Lega, M. Vancura, "A Multi-Level Synthesis Methodology", *to be presented* "International Workshop on Logic Synthesis", North Carolina, May 87.

[SMP86] C. Stroud, R. Munoz, D. Pierce, "CONES: A System of Automated Synthesis of VLSI and Programmable Logic from Behavioral Models", "Proc. of the ICCAD", November 1986.

[Tj86] S. Tjiang, "Twig Reference Manual", January 1986.

[TJB86] L. Trevillyan, W. Joyner, L. Berman, "Global Flow Analysis in Automatic Logic Design", C35, 1, 77-81 (1986).