1992-12

A minimization algorithm for non-concurrent PLA's

Dueck, Gerhard W.

International Journal of Electronics, Vol. 73, No. 6, Dec. 1992, pp. 1113-1119
http://hdl.handle.net/10945/35754

INT. J. ELECTRONICS, 1992, VOL. 73, NO. 6, 1113–1119

Minimization algorithm for non-concurrent PLAs

GERHARD W. DUECK† and JON T. BUTLER‡

In the design of certain self-checking programmable logic arrays (PLAs), at most one line is activated in the AND plane, such PLAs are termed non-concurrent. A heuristic algorithm for the minimization of non-concurrent PLAs is presented. It operates on two adjacent cubes, replacing them by one, two, and sometimes more than two cubes. The algorithm produces the best solutions known so far.

1. Introduction

In an AND-OR PLA, a 1 at the output occurs because one or more AND lines are 1. In a non concurrent PLA, exactly one AND line is 1 for every input pattern. Such circuits are used in totally self-checking circuits (Wang and Avizienis 1979, Khakbaz and McCluskey 1982, Tao and Lala 1986, Marcynuk 1990). The minimization of concurrent PLAs by Karnaugh map consists of covering all ones with as few valid circles as possible. In a non-concurrent PLA, the procedure is the same except that no two circles overlap. The sum-of-product expression of a non-concurrent PLA is said to be disjoint, since every minterm is covered exactly once.

Disjoint sum-of-product expressions are also needed in the minimization of mixed polarity Reed–Muller representations. For example, the Reed–Muller expression minimization program Exorcism (Helliwell and Perkowski 1988) requires a disjoint sum-of-product expression as its starting point. The algorithm described by Saul (1990) has a similar requirement.

As in a multi-output concurrent PLA, in a multi-output non-concurrent PLA, two or more outputs may share a product term. Indeed, for the second case, if two outputs are 1 for some assignment of values to the variables, non-concurrency requires those 1's to come from one product term. Thus, given two output patterns, for example 10 and 11, there can be no single product term that produces the 1's in both. It follows, therefore, that a multiple-output function should be partitioned into separate output patterns (e.g. 10 and 11) where each is considered separately; i.e. each is minimized in the same way as a single-output function is minimized. We therefore, adopt the Marcynuk (1990) approach of dividing the function by output pattern and minimizing each separately.

Wang and Avizieniz (1979) were the first to describe an algorithm for non-concurrent minimization. Marcynuk (1990) has shown that the Wang and Avizieniz algorithm produces poor results for many functions. Marcynuk (1990) has presented an algorithm which is based on a list of minterms. He proposes seven criteria according to which cubes should be selected. An adjacency count is computed for each minterm. All prime cubes that include the minterm with the maximum adjacency count are generated, and one is chosen to be part of the solution. These

Received 7 May 1992; accepted 19 May 1992.

†Department of Mathematics and Computer Science, St. Francis Xavier University, Antigonish, Nova Scotia, B2G 1CO, Canada.

‡Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA 93943-5004, U.S.A.

steps are iterated until all minterms are covered. The main drawback of Marcynuk's algorithm is that it enumerates all minterms. Our algorithm manipulates cubes, which are usually much less numerous.

Falkowski and Perkowski (1991) have presented an algorithm for the generation of a disjoint cubes. Their algorithm deals with single output functions only. In general, the minimal cover will not be obtained.

Finding the minimal sum-of-products expression for a non-concurrent PLA is intractable for all but the smallest functions. Indeed, the search for an exact sum-of-products expression for a non-concurrent PLA cannot be restricted to just the set of prime cubes, as can be done in concurrent PLAs. This motivates our interest in heuristic algorithms for non-concurrent PLAs.

2. Background and notation

We use the cube notation of Brayton *et al.* (1984). For example, over the variables x_1, x_2, x_3, and x_4, the product terms $A = x_2 x_3$ and $B = \bar{x}_1 x_2 x_4$ are $A = -11-$ and $B = 01-1$. See Fig. 1(*a*). In addition we define the following. The cost of a function is the number of cubes in its sum-of-products expression. The size of a cube is the number of $-$ in its cube notation. The distance between two cubes is the number of variables which are 0 in one cube and 1 in the other. Cubes at distance 1 are said to be adjacent. For example, $A = -11-$ and $C = 0-01$, shown in Fig. 1(*a*), are adjacent. The consensus $C(P_1, P_2)$ of two adjacent cubes $P_1 = P_1^i x_i$ and $P_2 = P_2^i \bar{x}_i$, where P_1^i and P_2^i are cubes that do not involve x_i, is $P_1^i P_2^i$. Applying this to $A = -11-$ and $C = 0-01$ yields $C(A, C) = 01-1 = B$.

The sharp operation (#) of two cubes P_1 and P_2 produces a minimal sum-of-products expression for $P_1 \bar{P}_2$, i.e. $P_1 \# P_2 = P_1 \bar{P}_2$. If the distance between P_1 and P_2 is greater than 0, then $P_1 \# P_2 = P_1$. Otherwise, for each 0 and 1 in P_1, all cubes in $P_1 \# P_2$ have a 0 and 1, respectively. For each $-$ in P_1 where P_2 is 0 or 1, there is a product term in $P_1 \# P_2$ that has a $-$ in all variables except one, where the value is 1 or 0, respectively. For example, with $A = -11-$ and $B = 01-1$, $A \# B = 111- + -110$. It should be noted that $P_1 \# P_2$ is unique.

Hong *et al.* (1974) define a disjoint sharp operation $\textcircled{\#}$, which is the same as the # operation except that the product terms are disjoint. For example, for $A = -11-$ and $B = 01-1$, $A \textcircled{\#} B = 111- + 0110 = -110 + 1111$. The second expression corresponds to the two rightmost cubes in Fig. 1(*b*). Another disjoint sum-of-products expression, $1111 + 1110 + 0110$ exists for $A \bar{B}$, but it does not represent $A \textcircled{\#} B$

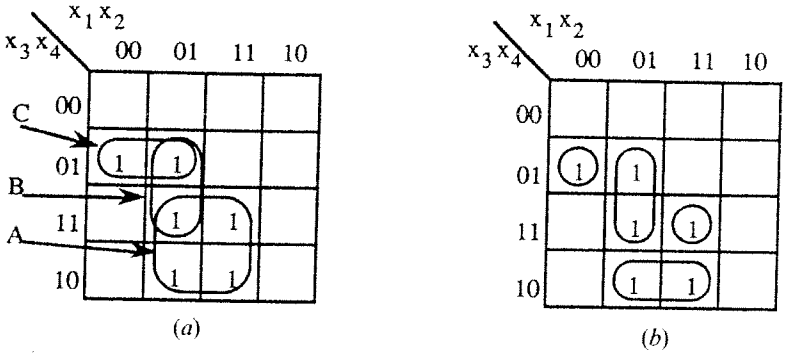

Figure 1. Covers of a function.

because it is not minimal. Unlike #, there is generally more than one representation of $\textcircled{\#}$ and we interpret $P_1 \textcircled{\#} P_2$ to be any of these. Indeed, there are $q!$ such expressions, where q is the number of — entries in P_1 that are matched by or 0 or 1 in P_2.

Let P_1 and P_2 be two adjacent cubes. The reshape of P_1 and P_2 is $R(P_1, P_2) = C(P_1, P_2) + [P_1 \textcircled{\#} C(P_1, P_2)] + [P_2 \textcircled{\#} C(P_1, P_2)]$. For example, $R(A, C) = 01—1 + 0001 + —110 + 1111$. This is shown in Fig. 1(*b*). Note that all cubes are disjoint. In this example, the reshape of two cubes produces four cubes, increasing the cost by two.

3. Simulated annealing

Kirkpatrick *et al.* (1983) introduced simulated annealing as a means to solve optimization problems. The basic computation in simulated annealing is a move. A move is a transition from one solution to another solution in the solution space. In the problem at hand, the solution space is the set of all disjoint sum-of-product expressions for some given function. Associated with each solution is a cost (i.e. number of cubes). A move can either increase, decrease, or leave the cost unchanged. Intuitively, one favours moves that decrease the cost, since this drives the system to a 'better' solution. However, in a solution space with local minima, the exclusive application of cost-decreasing moves can produce non-optimal solutions. Cost-increasing or hill-climbing moves are also needed if the system is to recover from a local minima.

In simulated annealing, prospective moves are chosen at random. If a move decreases the cost, it is always accepted. If it increases the cost, it is accepted with the probability $P(\Delta E) = e^{-\Delta E/T}$, where T is the temperature and ΔE is the increase in cost as the result of making the move. Initially, a high temperature is chosen and $P(\Delta E)$ is high, in which case, almost all moves are accepted. When the system reaches a steady-state at a high temperature, it is said to be melted.

Next, the temperature is reduced and the process is repeated. This continues, as the temperature gradually decreases the probability of accepting cost-increasing moves. There is slow progress toward an optimal state, and the system reaches a point where there is no further improvement, in which case, it is considered to be frozen.

The sequence of decreasing temperatures is called the annealing schedule. The annealing schedule we use is described by $T_n = \alpha T_{n-1}$, where α is between 0·80 and 0·99. When the temperature is rapidly reduced, a process called quenching, the result is often far from optimal. With $T_n = \alpha T_{n-1}$, this corresponds to a small α. Therefore, a slow decline is preferred, even though this requires more computation time.

4. Algorithm

We implemented the Marcynuk (1990) algorithm to make a multiple output function non-concurrent. This algorithm is concerned with both the input parts and the output parts of all cubes. However, it does not attempt to find a minimal cover. Marcynuk's algorithm is outlined in Fig. 2. We found that for some functions this algorithm generated a large number of cubes and required excessive cpu time. For example, for the function *vg*2 (from the Espresso test suite, Brayton *et al.* 1984) 73825 cubes were generated requiring 4081 s of cpu time on a SUN SPARCstation 2. A significant improvement was achieved by processing the cubes in descending order of size. That is, in the algorithm shown in Fig. 2, replace the line

```
algorithm MakeNoneConcurrent(CubeSet, NCubeSet)
    /*  Given a set of cubes CubeSet find an equivalent cover NCubeSet
        which is nonconcurrent. NCubeSet is initially empty.
    */
    for all c∈CubeSet
        add c to NCubesSet such that NCubesSet remains nonconcurrent
        /*for details see algorithm 4.4 in (Marcynuk 1990)*/
    end
end MakeNonConcurrent
```

Figure 2. Algorithm to make a function nonconcurrent.

for all c∈CubeSet

with the following 3 lines

while CubeSet$\neq\emptyset$ **do**
let c be a cube of maximum size in CubeSet
CubSet←CubeSet-{c}

Let NCubeSet be the set of non-concurrent cubes. With this change, NCubeSet for the function *vg2* contained 5648 cubes and require 12 s of cpu time on the same machine. Similar improvements were obtained with other functions.

Once we have a non-concurrent cover for a given function, the minimization problem is partitioned into N subproblems, where N is the number of distinct output patterns. Functions for each distinct output pattern are minimized using Espresso (Brayton *et al.* 1984). This yields an approximate lower bound for the number of disjoint cubes, since a non-concurrent cover cannot contain less cubes than a concurrent cover. It is approximate, since Espresso only produces near-minimal results.

Given a near-minimal concurrent cover of a single output function, with p product terms, we perform the following steps to find a near-minimal non-concurrent cover.

Step 1. *Make non-concurrent.* This step is similar to the algorithm shown in Fig. 2 (with the changes described above). However, the output parts of cubes may be ignored, since each cube is guaranteed to have the same output.

Step 2. *Simple Merge.* Sort the cubes by size in ascending order. For each distinct size, enumerate all pairs of adjacent cubes. Merge any cubes that can be combined. The sorting of cubes helps in two ways. Firstly, it reduces the enumeration of adjacent cubes, and secondly, cubes that result from a merge may later be considered for another merge. If there are p or fewer product terms, stop.

Step 3. *Cost preserving reshape.* Enumerate all pairs of adjacent cubes. Replace each pair of cubes P_1 and P_2 by $R(P_1, P_2)$ if $R(P_1, P_2)$ results in two cubes. After each reshape, merge the resulting cubes if possible. If there are p or fewer product terms, stop. This step is iterated twice.

Step 4. *Simulated annealing.* We use simulated annealing as described in the previous section. A move starts by choosing two adjacent cubes at random. They are combined if possible. If they cannot be combined, they are reshaped. The

```
algorithm SimulatedAnnealing(SetOfCubes,LowerBound);
    /*  Minimize a nonconcurrent set of cubes, SetOfCubes, given the lower bound.
        LowerBound, by simulated annealing with the following global constraints. All
        cubes produce the same output pattern. LowerBound can be obtained by using
        tranditional (concurrent) minimization. Default values are shown on parentheses.
        All values can be overriden by the user.
            InitialTemp - T0, temperature at which simulated annealing begins (0.7).
            LowestTemp - The lowest temperature below which simulated annealing is no
                longer applied (0.01).
            MaxMoves - Maximum allowed number of moves completed at each tempera-
                ture (4 x number of cubes in SetOfCubes).
            MaxAttemptedMoves - Largest number of attempted moves made before the
                current temperature is abandoned (50 x MaxMoves).
            MaxFrozen - Longest sequence of temperatures at which the number of
                attempted moves is MaxAttemptedMoves (4). At this temperature, many
                attempted moves are needed to achieve completed moves, and the solution
                is considered to be frozen. The simulated annealing is stopped, since little
                progress is achieved with continued computational effort.
            CoolRate - α, the factor used to determine the next temperature, i.e. Tn
                = αTn-1 (0.9).
    */
    BestCubeSet←CubeSet;
    CurrTemp←InitialTemp;
    Frozen←0;
    while (Frozen < MaxFrozen) and (CurrTemp ≥ LowestTemp)
            and (number of cubes in BestCubeSet < LowerBound) do;
        AttemptedMoves←0;
        Moves←0;
        while (Moves < MaxMoves) and (AttemptedMoves < MaxAttemptedMoves)
                and (number of cubes in BestCubeSet < LowerBound) do;
            choose two adjacent cubes C1 and C2 from CubeSet;
            AttemptaMove(C1 C2,CurrTemp,C1plusC2,MoveMade);
            AttemptedMoves←AttemptedMoves + 1;
            if (MoveMade) then
                Moves←Moves + 1;
                CubeSet←CubeSet - C1 - C2 ∪ C1plusC2;
                if (|CubeSet| < |BestCubeSet|) then
                    BestCubeSet←CubeSet;
                end;
            end;
        end;
        if (AttemptedMoves = MaxAttemptedMoves) or
          (|CubeSet|remained constant for all Moves) then Frozen←Frozen + 1;
        else Frozen←0;
        CurrTemp←CoolRate x CurrTemp;
    end;
    CubeSet←BestCubeSet;
end SimulatedAnnealing;
```

Figure 3. Simulated annealing algorithm.

acceptance of the move is based on the cost increase due to the reshape according to the annealing schedule. A formal description of the algorithm is given in Fig. 3.

Note, the procedure may stop at any time during Steps 2, 3, or 4 when there are no more than p product terms, since p is an approximate lower bound for the

PLA	n	m	N	p	1*con*	Step 2	Step 3	Step 4	Marcynuk
9sym	9	1	1	85	85	152	143	136	
alul	12	8	80	19	950	1123	1056	1032	1032
misex2	25	18	34	28	471	471			472
misex3	14	14	1040	690	1934	2198	2112	2100	
sao2	10	4	9	58	58	67	63	62	64
vg2	25	8	23	110	1584	1989	1951	1681	1921
apla	10	12	37	25	71	71			71
dist	8	5	21	120	164	164			164
dk17	10	11	24	18	49	49			49
dk27	9	9	38	10	41	41			41
in7	26	10	112	54	592	638	618	599	
x9dn	27	7	21	120	1280	2219	1854	1626	

n number of input variables
m number of outputs
N number of output pattern
p number of product terms after minimization (Espresso)
1*con* product terms minimizing each output pattern independently (Espresso)
Step 2 product terms after Step 2
Step 3 product terms after Step 3
Step 4 product terms after Step 4
Marcynuk product terms using Marcynuk's algorithm

Experimental results.

solution. In practice, the procedure frequently terminates before reaching Step 4. This is advantageous, since the simulated annealing step is the one which is computationally expensive.

5. Results

The algorithm has been implemented using the programming language C. Some results are shown in the Table. All benchmark functions are from the Espresso test suite (Brayton *et al.* 1984) and the FMS benchmarks from the Microelectronic Center of North Carolina (1987). The results are better than any published results known to us. Execution times for the minimization of *alul* are as follows: 0·12 s for Step 1 and 2, 0·23 s for Step 3, and 37·32 s for Step 4. The program was run on a SUN SPARCstation 2.

Two columns, p and *1con*, represent the best solutions for a *concurrent* cover produced by Espresso. p represents the best for a multi-output function, where sharing of cubes is used to find a near-minimal solution. *1con* is the result of minimizing each output pattern independently, where no sharing occurs. The columns labelled Step 2, Step 3, and Step 4 show the results obtained from the algorithm presented here. For functions, where there is no entry for Step 3 or Step 4, these steps were not applied. Such function achieved the number of cubes obtained by Espresso for the concurrent cover.

The columns labelled *1con*, Step 1, Step 2, and Step 3 summarize the minimization of N single output functions. Steps 2 and 3 may not have been applied to all functions. For example, consider the function *alul*. Step 1 achieved the bounds established by Espresso for 51 out of the 80 single output functions. Three

additional bounds were achieved in Step 3. Simulated annealing (Step 4) was applied to 26 functions.

6. Conclusions

We have presented a new algorithm for the minimization of nonconcurrent PLAs. By using standard minimization techniques, such as Espresso (Brayton *et al.* 1984), we can establish an approximate lower bound for any single output function. With the use of proposed heuristics, namely Simple Merge and Cost Preserving Reshape, these bounds are often achieved. If the lower bound is not achieved with these heuristics we apply simulated annealing. Simulated annealing is only used as a last resort, since it can be computationally expensive. The algorithm produces the best solutions known so far.

Acknowledgments

A preliminary version of this paper was presented at the IEEE Pacific Test Workshop held at Whistler, B. C. in May 1992. The research was supported in part by the Naval Research Laboratory, Washington DC through direct funds at the Naval Postgraduate School, Monterey, CA, by a National Research Council Research Associateship, and by an operating grant from NSERC Canada.

References

Brayton, R. K., Hachtel, G. D., McMullen, C. T., and Sangiovanni-Vincentelli, A. L., 1984, *Logic Minimization Algorithms for VLSI Synthesis* (Boston, Mass: Kluwer Academic).

Falkowski, B. J., and Perkowski, M. A., 1991, Algorithm for the genration of disjoint cubes for completely and incompletely specified boolean functions. *International Journal of Electronics,* **70,** 533–538.

Helliwell, M., and Perkowski, M., 1988, A fast algorithm to minimize multiple-output mixed polarity generalized Reed–Muller forms. *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 427–432.

Hong, S. J., Cain, R. G., and Ostapko, D. L., 1974, MINI: A heuristic approach for logic minimization. *IBM Journal of Research and Development,* **18,** 443–458.

Khakbaz, J., and McCluskey, E. J., 1982, Concurrent error detection and testing for large PLAs. *I.E.E.E. Transactions on Electron Devices*, 756–764.

Kirkpatrick, S., Gelatt, C. D. Jr., and Vecchi, M. P., 1983, Optimization by simulated annealing. *Science*, **220,** 671–680.

Marcynuk, D. M., 1990, *On-line checking of programmable logic arrays.* PhD dissertation, University of Manitoba.

Microelectronics Center of North Carolina, 1987, FMS benchmarks.

Saul, J. M., 1990, An improved algorithm for the minimization of mixed polarity Reed–Muller representations. *International Conference on Computer Design*, pp. 372–375.

Tao, D. L., and Lala, P. K., 1986, A concurrent testing strategy for PLSs. *International Test Conference Proceedings*, 705–709.

Wang, S. L., and Avizienis, A., 1979, The design of totally self-checking circuits using programmable logic arrays. *Digest of Papers 9th International Symposium on Fault-Tolerant Computing*, pp. 173–180.

```
    checkforcontradic:
      if OUTPUTFUNC(MIDTERMOUT) = INPUTFUNC
      (MIDTERMIN) then
      begin
        nextmidterm:
          MIDTERMIN ← MIDTERMIN + 1
          if MIDTERMIN ⩾ (2↑N1) + 1 then
            begin
              E ← 1
              CONVERT2(OUTPUTFUNC, FNCTN OUT)
            end
          else go to computefoutmid
      end
      else E ← 0
end procedure EXTRACT
```

Acknowledgments

I would like to thank Moshe Liron of Bell Telephone Laboratories, Naperville, Illinois for the program which generated the data of Tables I, II and II. Special thanks are due Patrick White, also of Bell Telephone Laboratories, Naperville, Illinois for conversations which inspired topics in this paper. Thanks are due the referee and Edward Bender of the University of California-San Diego for comments which led to an improved version of this paper.

Received: January 27, 1978; revised September 14, 1979

References

Amoroso, S., and Cooper, G. (1970), The Garden-of-Eden configuration for finite configurations, *Proc. Amer. Math. Soc.* **26**, 156–164.

Amoroso, S., and Epstein, I. J. (1976), Indecomposable parallel maps in tessellation structures, *J. Comput. System Sci.* **13**, 136–142.

Smith, A. R. III (1971), Cellular automata complexity tradeoffs, *Inform. Contr.* **18**, 466–482.

Winograd, T. (1970), "A Simple Algorithm for Self-Reproduction," MIT Project MAC Artificial Intelligence Memo, No. 197.

Printed by the St. Catherine Press Ltd., Tempelhof 37, Bruges, Belgium