Time- and Space-Efficient Evaluation of Sparse Boolean Functions in Embedded Software

Václav Dvořák
Brno University of Technology
dvorak@fit.vutbr.cz

Abstract

The paper addresses software implementation of large sparse systems of Boolean functions. Fast evaluation of such functions with the smallest memory consumption is often required in embedded systems. A new heuristic method of obtaining compact representation of sparse Boolean functions in a form of linked tables is described that can be used for BDD minimization as well. Evaluation of Boolean functions reduces to multiple indirect memory accesses. The method is compared to other techniques like a walk through a BDD or a list search and is illustrated on examples. The presented method is flexible in making trade-offs between performance and memory consumption and may be thus useful for embedded microprocessor or microcontroller software.

1. Introduction

Efficient evaluation of Boolean functions is an important part of many embedded software systems. Some applications include: search and optimization, modeling, simulation and verification of digital circuits in CAD, extracting routing information from packets, etc. We will restrict ourselves to sparse systems of Boolean functions, as these are most frequent in practice. Also we will address Boolean functions of large numbers (tens) of variables because small size systems can be implemented directly in hardware, e.g. in PLA, ROM or TCAM (Ternary Content Addressable Memory).

Software implementation of Boolean functions will be assumed in a form of a data structure describing the function and of a compiled program that reads the input vector and evaluates the function with the use of this data structure. The size of the code and of the data structure is one figure of merit, the other is the evaluation time from reading the input to generating the output. Sometimes the evaluation time per one input may get reduced if many inputs follow one another.

Hereafter we will use three compact representations: a TCAM-like table, linked tables and binary decision diagrams (BDDs). The BDDs are well known, especially the reduced ordered BDDs (ROBDDs), [1]. On the base of ROBDDs we will develop a more practical representation - linked tables.

Software implementation of Boolean functions has been up to now studied especially in connection with PLCs ("ladder diagrams") [2], digital system simulation, formal verification and testing [1], or specialized event processing [3], where either a speed (PLC) or a required memory were not that important. On the contrary, in embedded systems we do care for performance and memory space as well as for power consumption. We will demonstrate that presently used algorithms are generally too slow (TCAM emulation, BDDs); the use of linked tables enables faster evaluation with similar (selectable) size data structures which can be minimized by utilizing don't cares directly.

The paper is structured as follows. In the following Section 2 we define sparse Boolean functions and show their description by data structures, a relation among various representations and their complexity. Our novel heuristic approach for minimizing the relevant data structures is explained in Section 3. In Section 4 we exemplify creation of data structures on a sample Boolean function and give ways how to trade speed of evaluation against required memory space in Section 5 and 6. Results obtained with selected functions of 8 to 64 variables are commented on in Conclusions.

2. Representation of sparse systems of Boolean functions

To begin our discussion, we define the following terminology. Multiple-output Boolean functions of n Boolean variables will be simply referred to as Boolean functions F_n with output values from

$$Z_R = \{0, 1, 2, \ldots, R-1\},$$

$$F_n : Z_2^n \to Z_R . \quad (1)$$

Under a *sparse Boolean function* we will understand function F_n: $Z_2^n \to Z_R$ with the domain split into two parts X and D, $Z_2^n = X \cup D$, $|X| << 2^n$, and specified in one of two ways:

1) $Z_2^n \setminus X = D$ is the don't care set (F_n is an incomplete function in Z_2^n, F_n: $X \to Z_R$)

2) Mapping of the don't care set $Z_2^n \setminus X = D$ into Z_R is artificially defined to make implementation as easy as possible, F_n: $X \cup D \to Z_R$.

A binary decision diagram (BDD) is a directed acyclic graph in which each decision node is labeled by a control variable tested in this node. Two edges coming out from the decision node, leading to the nodes in the subsequent levels, correspond to the values of control variable. Beside decision nodes there are terminal nodes (leaves) labeled by the value of the function that is being evaluated by the diagram. A BDD is ordered, if the order of control variables tested along every path in the BDD is the same. All the nodes of the ordered BDD (OBDD) labeled by the same variable make up a level of this diagram. An ordered BDD is reduced (ROBDD) if [1]

1. no two distinct nodes have the same control variable name and the same 0- and 1-successor node.
2. no node has the identical successor for both values of a control variable.

ROBDD is canonical (unique) representation for any given function [1]. Any pair of functions will have different ROBDDs unless the functions themselves are equivalent. In the rest of the paper, we will consider OBDDs or ROBDDs , even if the term BDD is used.

An important parameter is a size of BDD, i.e. the total number of decision nodes, as it determines the size of data structure needed to store a BDD. The construction of minimum-size BDDs belongs among NP-hard problems [9]. Upper bounds on the OBDD's size for general Boolean functions are not too encouraging, but many practical functions do have a reasonable BDD size. The upper bound on the size of the related BDD belonging to a function with L literals in its DNF can be obtained as [4]

$$\min_k \left\{ L - \left\lceil k \tfrac{L}{n} \right\rceil + 2^k - 1 \right\}, \quad k = 1,2,...,L \quad (2)$$

As the first approximation to (2) we can use L only.

Traditional description of a Boolean function by the Boolean expression (expressions in case of multiple outputs) may sometimes be the best for evaluation purposes, especially if the function of many variables is defined only in a few points. Another possibility is a list of ternary input vectors (composed of 0, 1 and don't care elements) that can specify a relevant output value for each input vector in the list. Ternary vectors can be stored as two binary vectors, the vector of values and a mask indicating don´t cares.

Efficiency of evaluation techniques strongly depends on the used description of the Boolean function. Let us illustrate above possibilities on the N queens problem. We are to compare possible representations of a Boolean function of $N \times N$ variables that returns 1 for every configuration of 1's (queens) on the $N \times N$ chessboard, such that no queen attacks another one. For example if $N = 4$, there are 2 solutions that can be generated by Boolean function F_{16} (used notation is required by the applet [5], but variables in the diagram are enumerated from 0 to 15):

$$F_{16} = !a_{11}*!a_{12}*a_{13}*!a_{14}*a_{21}*!a_{22}*!a_{23}*!a_{24}*!a_{31}*!a_{32}*!a_{33}*a_{34}*!a_{41}*a_{42}*!a_{43}*!a_{44} + !a_{11}*a_{12}*!a_{13}*!a_{14}*!a_{21}*!a_{22}*!a_{23}*a_{24}*a_{31}*!a_{32}*!a_{33}*!a_{34}*!a_{41}*!a_{42}*a_{43}*!a_{44} \quad (3)$$

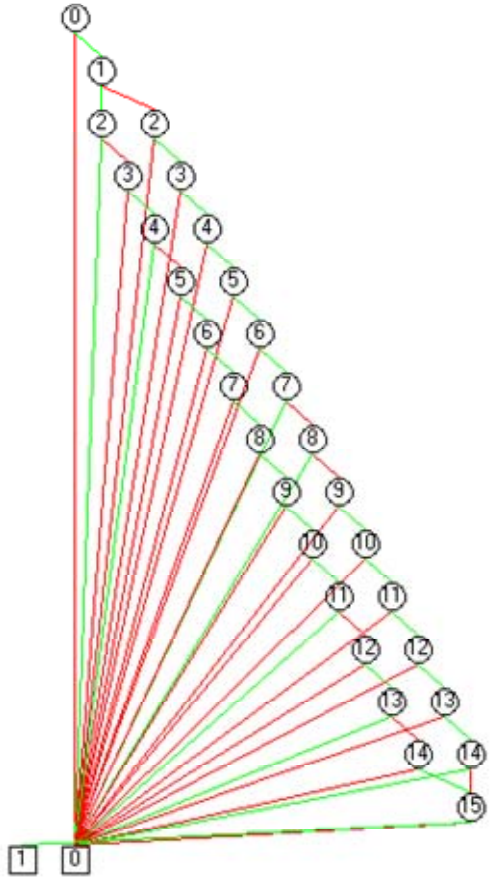

Fig.1. ROBDD for 4 queens problem

In our case L = 32, n = 16 and the upper bound according to eq. (2) is 31 nodes. The real ROBDD generated by the applet [5] has 29 nodes and is shown in Fig.1. The most efficient representation and evaluation of F_{16} is thus clear: two memory words are

sufficient to store two min-terms in (3) and two bitwise comparisons will do for the quickest evaluation. A BDD representation is no good in this case.

Now if we move to 8 queens problem, there will be 92 solutions described by F_{64}. Solutions can be found by the known algorithm [6], [1]. To store 92 binary vectors of length 64 is still acceptable, but instead of linear search we can order solutions and do better with the logarithmic search in $\lceil \log_2 92 \rceil = 7$ steps at most.

The BDD size is upper-bounded by 5535 nodes according to eq. (2), so that storing of such BDD would not be space efficient at all. A pass through this BDD would need 64 steps in the worst case, what is bad as well.

3. Heuristic construction of BDDs and linked tables

We did not see too much use for BDDs in the former example. However, in this section we will give a method of construction of linked tables which are much more useful for the purpose of function evaluation. The method is based on bottom-up heuristic construction of BDDs, which uses a concept of sub-function [7].

Informally, the sub-function f of F_n is a function of s variables obtained from F_n by setting $n-s$ variables to fixed constant values. The number of distinct sub-functions of s variables, s = 1, 2, …, n-1, characterizes the Boolean function and its complexity. Sub-functions themselves may also be incomplete (don't care values for some binary s-tuples). A compatibility relation can be defined on the co-domain of such sub-functions: don't care (denoted by "x") is compatible with any value from Z_R.

Using the concept of sub-functions, we will now decompose iteratively the given sparse function of 8 variables, see Fig.2. The map of the function at the top is sparsely populated by 16 function values (0 to F). For clarity don't care cells are left empty in tables, but otherwise are denoted by symbol "x" in the text. Single-variable sub-functions can be created with respect to any variable. E.g. two vertically adjacent cells correspond to a sub-function of the first variable that attains alternate values 0 and 1 at even and odd rows (see e.g. [F,8] in Fig.2). Using compatibility relation we can combine pairs [α,x] and [x,β] into a single sub-function [α,β]. Altogether nine sub-functions of the first variable are detected in the topmost table. The first decomposition step is described below the table; each sub-function is given a new symbol ([1,0]→0, [2,7]→1,[F,8]→3, etc.), thereby

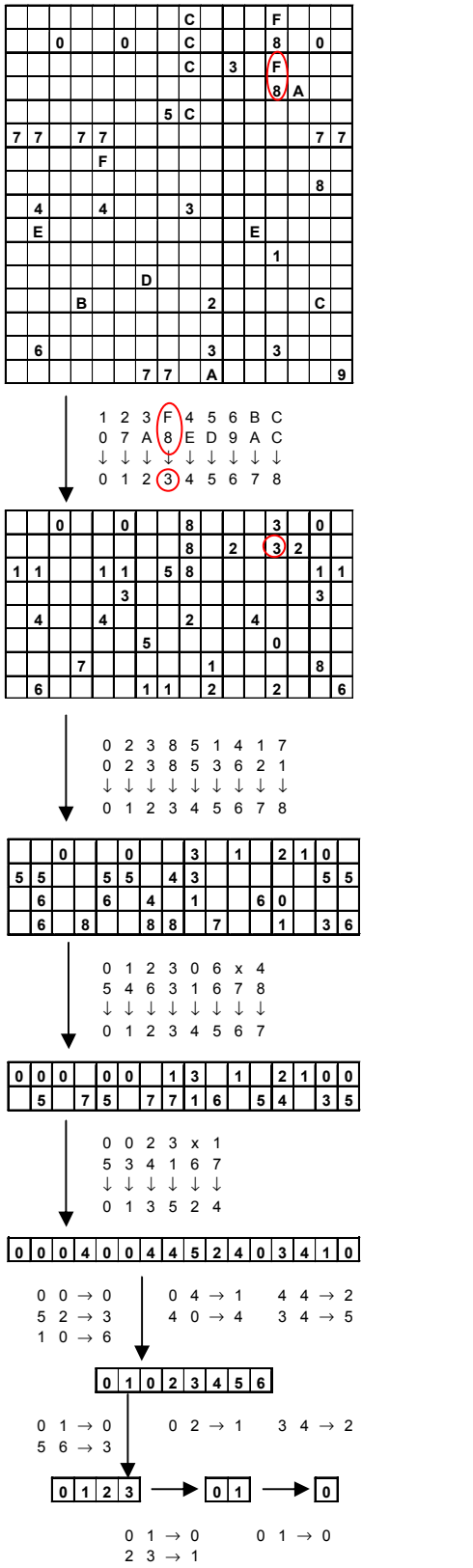

Fig. 2. Iterative decomposition (8 variables)

removing the first variable from the function. A map of the new intermediate Boolean function of 7 variables is now created replacing sub-functions by new values (symbols, IDs). This process repeats 8 times.

The OBDD can now be created starting from root 0. Every assignment [a,b]→ c, when reversed, specifies one decision node with input c and two outputs a and b controlled by the relevant variable. Assignments of the type [a,a]→b, [a,x]→c, [x,a]→d do not represent decision nodes because the outputs are the same (or compatible); such a decision node degenerates to a wire. Going up from the root (a map of 0 variables) to the original map of 8 variables, the OBDD in Fig.3 is created. Usually BDDs have a root at the top, but we displayed the BDD upside down in order to keep the BDD structure in correlation with the sequence of map transformations in Fig.2. Nodes are labeled by intermediate function values. Out of 46 assignments 34 correspond to decision nodes and 12 to wires only.

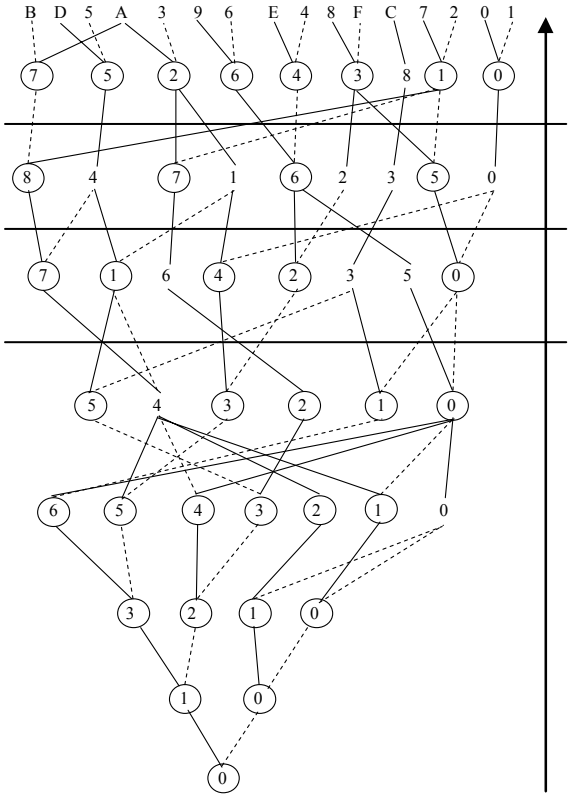

Fig.3. OBDD of the sample function of 8 variables (0 = - - - - - , 1 = ———)

In our example we did not care about variable ordering; the ordering was chosen more or less randomly. However, it is known, that the size of the BDD is determined both by the function being represented and the chosen ordering of the variables [8]. For some functions, the size of a BDD may vary between a linear to an exponential range depending upon the ordering of the variables. The problem of optimal variable ordering is unfortunately NP hard [9].

If we want to minimize the size of a BDD, the following heuristics can be used: do sub-function counting for all variables in each decomposition step and use for this step the variable with the minimum sub-function count. By intuition, the minimum count of symbols may hopefully produce a minimum count of their pairs.

Note also that the above small example with maps of the original and intermediate functions was done only for illustration. When we have sparse functions with several tens of variables represented by a list of defined points, all the processing is done on these lists. The case with don´t cares already defined for the purpose of minimization is given in Section 6.

4. Linked tables and OBDDs

In this Section we first introduce the technique of linked tables, programs for interpreting OBDDs as well as linked tables and then compare both techniques on examples.

Linked tables and OBDD are equivalent descriptions of a Boolean function; one layer of the OBDD or more layers combined can be described by a table. For example linked table 4 is constructed (Fig. 4) from the top layer of the OBDD in Fig. 3. Transformation of 9 symbols to 16 symbols is described by reversed assignments under the topmost map in Fig.2.

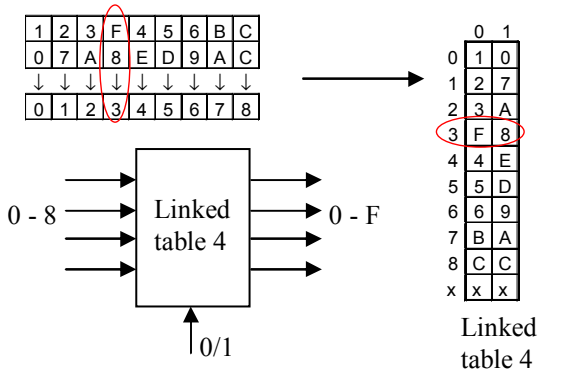

Fig.4. Construction of one of linked tables

The whole BDD (Fig. 3) is then described by 4 tables as shown in Fig. 5. The chain of tables is homogeneous, but generally the tables may have different size. However, sparse functions are typically implementable by homogeneous cascades, since the

number of sub-functions (and therefore decision nodes) follows a pattern: rising – constant – dropping, [4].

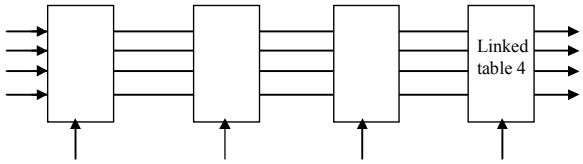

Fig.5. A chain of linked tables for the Boolean function at Fig. 2.

As can be seen, the difference between OBDD and linked tables is in communication among the layers or tables: in OBDD each symbol requires an individual edge (”wire”), whereas the symbols being sent between tables are binary coded. Another way to look at linked tables is to consider each table as an M-ary decision node and the chain of tables as a special “*in-line*” decision diagram.

This difference of two representations reflects itself in the way how the program interprets a certain application-specific OBDD or a table chain. In case of the OBDD we may use for each node a record with 3 fields. A format indicator is one-bit field specifying the leaf node. Two other fields of the leaf node are then used for output. If the node is not a leaf, two fields (adjacent words) contain pointers to the base addresses of other nodes. The base address is modified by the value of a current control variable(s) and is used to extract the correct field with the pointer to the next node. The program walks through a certain path in the ROBDD from the root to a leaf in at most *n* steps.

Linked tables are interpreted similarly, only the pointer to the next table is obtained from the current output by concatenating it with the control variable value and adding to the next table base address. As seen from Fig.5, only few steps will do. If suitable, some linked tables can be combined to provide even faster access. E.g. 4 tables in Fig.5 can be reduced to two with 6 inputs each.

5. Linked tables versus other methods

On the example of 4 queens and 16 queens we have already seen that Boolean expressions may support very fast evaluation and take up minimum memory space.

Let us now analyze 4 functions of 8 variables in Fig.2. Had we used an ordered list of defined points with function values, there would be 39 items, 8 (input) + 4 bits (output) per item, 468 bits in total, i.e. half of the full function table with size 256×4 bits. To look up the item in the table we would need $\lceil \log 39 \rceil = 6$ steps in the worst case.

On the other hand, if we use a chain of linked tables according to Fig.5, the capacity of all tables will be $4\times(32\times4) = 512$ bits and only 4 steps (composed of read, append a value of a selected variable, add to the base address of the table to create a pointer) will do. This seems to be the best in speed and memory efficiency. Four tables may be implemented in memory as one table 32×16 bit with the correct output extracted from 16-bit word as needed. Additional flexibility is obtained with linked tables as they are combined together. For example with 2 tables 64×4 bits, the response will be 2-times faster. The size of 2 and 4 linked tables remains the same, but 2 tables combined need 64 words in memory, 8 bits per word.

As the last example we shall consider the following sparse Boolean function of 32 variables: it attains the value 1 if the given 6-bit string is detected anywhere within an input string of 32 Boolean values; otherwise the function has the value 0.

As the string of 6 consecutive values of variables may be located in 27 positions (we do not assume that the pattern wraps around), we can specify the function by 27 words of 32 ternary digits (0, 1, x). The logarithmic search is now not possible and we have to step through these words sequentially. In the worst case it may take 27 steps.

We can do much faster with linked tables, though. First the ROBDD of this function may be obtained using the applet [4], since the Boolean expression with 27 min-terms, each with 6 literals, is easy to write. The ROBDD is too large (162 nodes in total) to display, but its shape can be described like this (from the root): the number of decision nodes per level linearly increases from 1 to 6, then stays at 6 for 22 levels and finally drops from 6 to 1. From this shape of the ROBDD an optimal size and count of linked tables can be determined, Fig.6. We can keep 6 tables in 256 words of memory, 3+3+3+3+2+1= 15 bits per word. The table items can also have 1-bit format indicator “continue/end” (6 additional bits in total) and the length of processing may vary between 1 to 6 steps.

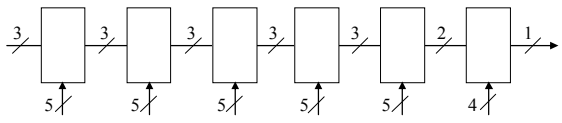

Fig. 6. Linked tables detecting 6-bit string in 32-bits

On the other hand, we could use the ROBDD implementation directly. Since there are 162 nodes, 8-bit address is needed. With format indicator (1 bit) we

will not be able to map one decision node to a single 16-bit word. Anyway, we can use 2 × 162 = 324 words, 16 bit each or 162 words, 32 bit each. The pass through the ROBDD may take from 1 to 32 steps. Apparently, this solution is worse than the linked tables.

Returning to the first example in Fig.1, we can also use linked tables here. Two sub-function symbols plus two constants 0 and 1 are transferred between BDD layers, so that 2-bit code will do. Possible configurations of linked tables are in Fig.7.

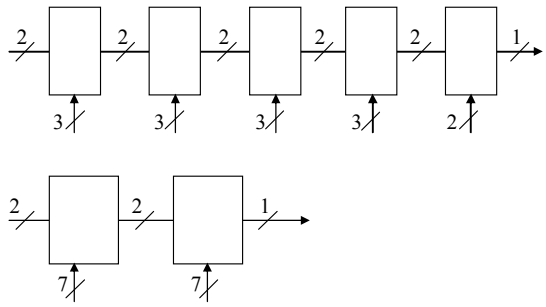

Fig. 7. Linked tables for 4 queens problem

Two table look-ups are sufficient in the shorter version, the same speed as with two comparisons suggested earlier. However, memory consumption is worse, 512 × (2+1) bits is incomparable to 2 words, 32 bits each.

6. A case study - MCS-51 microcontroller family: PLA1 and PLA2 in software

Space and time efficiency of various configurations of linked tables obtained by computer-aided iterative decomposition have been tested on two PLAs used in the core of MCS-51 family of microcontrollers,

PLA: $X \rightarrow R, X \subset Z_2^n, R \subset Z_2^r$,

with parameters in the following Table 1.

Table 1. Parameters of PLA1 and PLA2

	n	r	p	|X|	size [B]
PLA1	13	8	31	175	8192
PLA2	11	8	53	632	2048

Both PLAs implement sparse (incomplete) Boolean functions, which are after minimization described by Boolean expression in Appendix. The number of terms in AND arrays are p = 31 and 53. The size in bytes The size in bytes gives memory space $r2^n$ required for storing full function tables.

Iterative decomposition used the selection of those two variables at a time that produced the minimum number of sub-functions. Not too large size of the problem allowed still an exhaustive search – on the Pentium-based PC it took tens of seconds. The PLA1 was implemented by the cascade of 6 cells, Fig. 8a, with the total size of cell tables (ROMs) only 1792 bits. That is reduction by factor of 36. The size of tables is not uniform and evaluation would take 6 table look-ups. We can make it faster and more uniform by combining 6 cells into 3 as shown in Fig.8b. All sub-functions are counted (results given in {integer}), coded and communicated between cells, so that function values are outputs from the last cell only. The total size of linked (cell) tables is then 2816 bits; if the size of computer word *w* is known, further optimization can be done to minimize the total memory space in bytes occupied by all 3 (or possibly 4) tables.

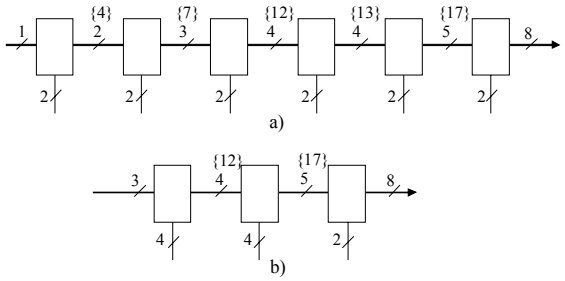

Fig. 8. Two cellular cascade implementations of PLA1

As far as PLA2 is concerned, computer-generated cascades are shown in Fig. 9. The cascade at Fig. 9b is obtained from the cascade a) by merging first two cells. The capacity of linked tables is 3264 and 3456 bits, respectively. The evaluation speed is given by 4 or 3 table look-ups.

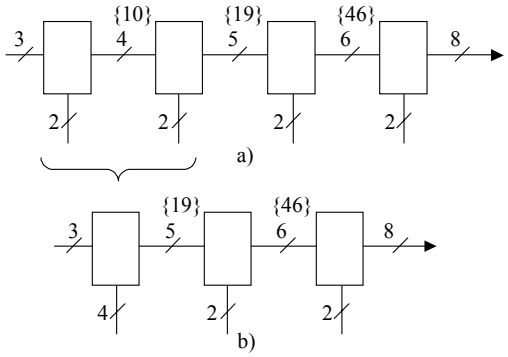

Fig. 9. Cascade of 4 or 3 cells for PLA2

We can also split output variables into two halves and then decompose them separately. The result for PLA2 is shown at Fig. 10. The size of linked tables is reduced to 1200 bits only, but the speed is reduced also. Eight table look-ups are needed and can be done

on one CPU core in 8 steps sequentially or on a 2-core processor concurrently in 4 steps.

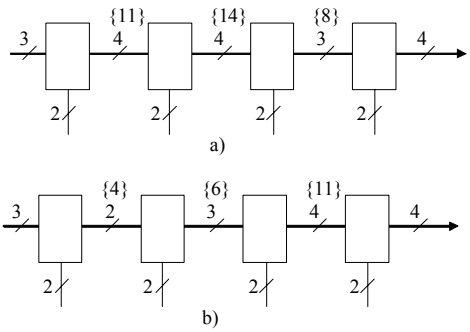

Fig. 10. Two parallel cascades implementing PLA2.

The case study of PLA1 and PLA2 offered the size of data structures and speed of evaluation as given in Table 2. The data in the table are valid under the assumptions:

- size is in bits, the length of a computer word is not considered;
- steps may have different duration in the left and the right part of the table (mask load + bitwise logical operation vs table look-ups).

Table 2. Software implementations of PLA1 and PLA2

	PLA emulation AND + OR matrix		linked tables	
	size bits	steps	size bits	steps
PLA1	1054	13 + 8	1792	6
PLA1	1054	31 + 8	2816	3
PLA2	1590	11 + 8	3456	3
PLA2	1590	53 + 8	1200	8

7. Conclusions

There is no single software evaluation method optimal for all Boolean functions. Complexity of functions that can appear in embedded systems varies a great deal and so do their space and time requirements in various evaluation techniques.

Even though the very narrow analysis done above cannot be taken as convincing, certain conclusions for engineering practice can be drawn from it, if the fast and memory efficient evaluation of sparse Boolean functions $F_n : X \rightarrow Z_R$ of several tens of variables is the main concern.

1. If the set $X \subset Z_2^n$ contains only a small number of elements, e.g. when the function is specified by DNF with few tens of minterms, the search in the ordered list of minterms can be very effective solution.

2. If $X \subset \{0,1,x\}^n$, sequential TCAM emulation may be too slow as it takes | X | steps.

3. OBDDs or ROBDDs may be useful for checking equivalence between two implementations or for formal verification [1], but they are less useful for evaluation purposes in both speed as well as memory consumption.

4. Linked tables obtained from ROBDDs seem to be a very good and effective data structure and should always be considered for evaluation of Boolean functions. They are flexible in making trade-offs between response time and memory consumption. If implemented as special hardware (a cascade of ROMs), they can support pipeline processing with one evaluation in each ROM cycle. Otherwise, in case of software implementation, several linked tables can be compacted into one table and stored in memory. The evaluation then reduces to a short chain of indirect memory accesses. Generally speaking, every sparse function can be implemented as a chain of linked tables or equivalently as a special "in-line" multi-valued decision diagram [4].

Future research will be oriented to study of evolutionary techniques for iterative decomposition of sparse Boolean functions of many variables where the exhaustive search is out of question. Large systems specified by expressions (such as those in Appendix) will be tackled either by parallel processing or by hardware acceleration.

8. References

[1] H.R.Andersen, Lecture notes for 49285 Advanced Algorithms E97, www.itu.dk/~hra/notes-index.html

[2] F. D. Petruzella: *Programmable Logic Controllers,* McGraw Hill Science/Engineering/Math, 2004.

[3] R. Sosic, J. Gu, and R. Johnson. "The Unison algorithm: Fast evaluation of Boolean expressions". *ACM Transactions on Design Automation of Electronic Systems*, 1(4): pp. 456--477, Oct. 1996.

[4] V. Dvořák, "Bounds on Size of Decision Diagrams", JUCS - *The Journal of Universal Computer Science*, Berlin, Heidelberg, CZ, Springer, 1997, s. 2-22.

[5] University of Hamburg, Department of Informatics, http://tams-www.informatik.uni-hamburg.de/applets

[6] W. Stallings, *Computer Organization and Architecture*, Sixth Edition, Prentice Hall, 2005.

[7] I. Wegener, *The Complexity of Boolean Functions*. John Wiley & Sons, New York, 1987.

[8] R. Drechsler, B. Becker, *Binary Decision Diagrams - Theory and Implementation.* Springer 1998.

[9] B. Bollig, I. Wegener, "Improving the Variable Ordering of OBDDs Is NP-Complete". *IEEE Transactions on Computers*, 45(9):993—1002, September 1996.

Acknowledgement

This research has been carried out under the financial support of the research grants "**Design and hardware implementation of a patent-invention machine**", GACR 102/07/0850, Grant Agency of Czech Republic, 2007-2009 and "Security-Oriented Research in Information Technology", MSM 0021630528..

Appendix

Programmable logic arrays PLA1 and PLA2 in MCS-51 microcontroller family

Legend: ! = logical negation, * = logical AND,
+ = logical OR

PLA1
Inputs: A, B, C, D, E, F, G, H, I, J, K, L, M
Outputs: SO, CS, BL, NL, V1, V3, V4, V5

SO = !A*!G*!I*J*M+A*!B*!I*J*M+A*F*!I*M

CS = !A*!B*D*!E*!F*!G*!H*!I*!K*!L*M + A*B*!E*!F*!G*!H*!I*!J!K*!L*!M + !A*!E*!I*M + !E*!I*J*M+!D*!I*M

BL = !B*E*!F*!G*!H*!I*!J*!K*!L + !B*C*!D*!H*!I*!J*M + !B*D*E*!H*!I*!J*M + !D*!I*!J*K*M + !A*!G*!I*J*M + E*H*!I*!L*M +C*!D*G*!I*M + !A*F*!I*M + G*!I*K*M + E*G*!I*M

NL = !B*E*!F*!G*!H*!I*!J*!K*!L + C*!D*!H*!I*L*M + !D*!I*!J*K*M + !A*!G*!I*J*M + D*E*!N*!I*M + !A*F*!I*M + E*!I*!L*M + G*!I*K*M

V1 = !A*!G*!I*J*M + C*!D*F*!I*M + A*!B*!I*J*M + !A*F*!I*M + F*!I*K*M + E*F*!I*M

V3 = !B*!C*!D*E*!F*!G*!H*!I*!J*!K*!L + !B*!G*!I*J*K*M + !D*!I*!J*K*M + B*C*!I*K*M

V4 = !B*C*!D*E*!F*!G*!H*!I*!J*!K*!L + !B*D*E*!F*!G*!H*I*!J*!K*!L*M + !A*!G*!I*J*L*M + C*!D*!H*!I*L*M + !A*F*!I*L*M + C*!D*H*!I*M +D*E*!I*L*M

V5 = !B*D*E*!F*!G*!H*I*!J*!K*!L*M + !B*E*!F*!G*!H*!I*!J*!K*!L + C*!D*!H*!I*L*M +!D*!I*!J*K*M + !A*!G*!I*J*M + C*!D*H*!I*M + A*!B*!I*J*M + D*E*!I*L*M + !A*F*!I*M + E*!I*!L*M

PLA2
Inputs: A, B, C, D, E, F, G, H, I, J, K
Outputs: Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8

Q1 = !A*!B*!C*!D*E*!G*I*!J + A*D*!E*F*!G*I*!J + A*!B*!E*F*I*!J + C*D*E*F*I*!J + A*C*G*!H + A*!C*!F + A*!C*!D + A*!B*!D +A*E*G + A*!I + A*J

Q2 = A*D*!E*F*!G*I*!J + !A*D*E*!G*I*!J + C*D*E*F*I*!J A*B*!D*F + F*G*!H*I + D*!F*H*I + !A*B*F*J + F*G*!H*J + D*!F*H*J + B*!I*!J + B*!E*!H + B*!C*H + B*C*E + B*E*G + B*D

Q3 = A*D*!E*F*!G*I*!J + !B*C*!D*F + A*C*G*!H + F*G*!H*I + F*G*!H*J + C*!I*!J + !B*C*!E + B*C*E C*F*J + !A*C

Q4 = !A*!B*!C*!D*E*!G*I*!J + !A*B*C*!G*I*!J + B*C*!E*H*I + F*G*!H*I + D

Q5 = A*D*!E*F*!G*I*!J + !A*B*C*!G*I*!J + !A*C*D*!G*I*!J + B*D*F*!G*I*!J + A*B*C*D*F*J + F*G*!H*I + F*G*!H*J + E*F*!I + E*F*H + E*F*J + E*!F

Q6 = !B*!C*D*G*!I*!J*!K + !A*B*D*E*!I*!J*!K + A*B*D*E*G*!I*!J + !E*G*!H*!I*!J*!K + !A*C*D*!G*I*!J + B*D*F*!G*I*!J + !D*!H*!I*!J*!K + C*D*E*I*!J + !B*!C*F*!J + C*F*!I*!J + !B*C*!D*F + A*B*!D*F + !A*B*F*J + !B*D*F*J + E*F*!I + C*F*G + E*F*H + E*F*J

Q7 = A*!B*!E*F*I*!J + !A*B*E*I*!J + C*F*G + G*!J + H

Q8 = H