# Space-Time Trade-offs in SW Evaluation of Boolean Functions

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

## Abstract

*Fast evaluation of multiple-output Boolean functions with the smallest memory footprint is often required in embedded systems. The paper describes a novel method of linked tables for representation and evaluation of Boolean functions and compares it with traditional methods; PLAs from the MCS-51 micro-controller are used for comparison. Traditional methods use masks to emulate PLA one way or another. The suggested method of linked tables is based on iterative disjunctive decomposition and leads only to a series of table look-ups. Linked tables are also shown to be equivalent to specific "in-line" decision diagrams. They proved to be most flexible in making trade-offs between performance and memory space. The method of linked tables may be quite useful for embedded microprocessor or microcontroller software as well as for digital system simulation.*

Keywords: Boolean function evaluation in software, PLA in software, decision diagrams, linked tables, iterative disjunctive decomposition.

## 1. Introduction

Efficient evaluation of Boolean functions is an important part of many embedded software systems. Simultaneous evaluation of several Boolean functions of many variables, that we are going to focus at, can be either compiled or interpretive. Compiled evaluation is targeted to a specific set of functions and if a new function set is to be used, re-writing a program and re-compilation are necessary. On the other hand, interpretive techniques use a general-purpose program that operates on application-specific data structures and on the input vector. Changing Boolean functions means that data structures representing them must be changed, not the program itself. We are interested only in the latter class of techniques. In embedded systems the size of the code/data structure and speed of evaluation are usually of concern.

If the interpretive program implements a virtual logic processor, then the description of Boolean functions in a form of Boolean expressions (or Binary Decision Diagrams, BDD) is appropriate. An evaluating algorithm uses either logic operations or (in case of BDD) branching. In both cases redundant reading/testing of input variables is used and multiple Boolean functions are evaluated sequentially one after another, which is not too efficient. This was all right for PLCs [1] or specialized event processing [2].

On the contrary, in embedded systems we do care for performance and memory space, as well as for power consumption, and alternative approaches are needed. For example, the techniques that utilize bit-wise operations on computer words (1 or more bytes in size) perform much better. However, when we have to trade-off speed and memory size, these techniques will not help. That is why the evaluation problem is studied anew in this paper.

The paper is structured as follows. In the following Section 2 we will take a look at traditional methods of evaluation and their complexity. Our approach of obtaining a description of Boolean functions in a form of M-ary decision diagrams or linked tables is explained in Section 3 and 4. In Section 5 we compare traditional and novel techniques with respect to size of required data structures and speed of evaluation on the sample set of Boolean functions (control PLAs from MCS-51 microcontroller family). Results are commented on in Conclusion.

## 2. Evaluation of Boolean functions with bit-wise logical instructions

To begin our discussion, we define the following terminology. A system of $m$ Boolean functions of n Boolean variables (also known as a multiple-output Boolean function),

$$f_n^{(i)} : Z_2^n \rightarrow Z_2,\ \ i = 1, 2, ..., m \qquad (1)$$

will be simply referred to as Boolean function $F_n$ with output values from $Z_R = \{0, 1, 2, \ldots, R-1\}$, $R = 2^m$,

$$F_n: Z_2^n \rightarrow Z_R\,. \qquad (2)$$

Function $F_n$ is incomplete if it is defined only on set $X \subset Z_2^n$ ; $Z_2^n \setminus X = D$ is then the don't care set.

Representation of Boolean functions by means of Boolean expressions can follow either eq. (1) or (2). In the first case, several non-disjunctive Boolean terms add up to generate a single binary output value, whereas in the second case they describe sub-domains mapped into one $R$-ary output value. Let us note, that the transition between both forms is not trivial.

Alternative representation of Boolean functions by means of binary decision diagram (BDD) can have similarly two forms, either m BDDs, one for each of m Boolean functions, or one BDD with $R$ terminal values. The latter form is more concise, but to obtain it from form 1 is not easy.

Hardware implementation of Boolean functions in Programmable Logic Array (PLA) can serve as an initial prototype for software implementation. PLA consists of AND-matrix and OR-matrix. Rows of the AND-matrix define terms and OR-matrix serves for accumulation of their contributions to the binary outputs.

The set of $p$ terms produced by AND-matrix (a term vector) can be generated in parallel, each term in one bit of the computer word. If the capacity $w$ bits in a single word are not enough, $\lceil p/w \rceil$ computer words can be used to accommodate all the terms. The terms are evaluated in $n$ steps – one input Boolean variable at a time. Two masks $m0(x)$ and $m1(x)$ are maintained for each variable $x$. The masking bit in a position of term $t$ is denoted $m0(x, t)$ or $m1(x, t)$. Given the value $v$ of input variable $x$, two masks are generated (only once, at the beginning) using the following rules:

if $x$ occurs in $t$, then $m0(x, t) = m1(x, t) = v$
if $!x$ occurs in $t$, then $m0(x, t) = m1(x, t) = !v$
if x does not occur in $t$, $m0(x, t) = m1(x, t) = 1$.

The term vector is initialized to all ones and then a sequence of masks is applied to it using logical AND operation. For variable $x$ either mask $m0(x)$ or $m1(x)$ is used according to the input value $x = v$. All the terms are thus updated in parallel and the (full) width of computer word is utilized.

As soon as all terms are ready, we have to emulate OR-matrix – apply OR operation selectively to certain bits. Another set of r masks will be used for r outputs. Unused terms in the term vector are masked out and if at least single 1 remains, we have the result TRUE. The memory size for storing all sets of masks is

$$\text{space} = (2n + r)\lceil p/w \rceil \text{ words} \tag{3}$$

and time complexity is

$$\text{time} = C_1 n + C_2 r, \tag{4}$$

where $C_1$ and $C_2$ are execution times in clock cycles related to mask applications.

If the number of terms $p$ is less than the number of variables $n$, a dual evaluation method may be more advantageous. The relevant terms are generated one after another from the input vector using again two sets of masks. As soon as the term vector is assembled, the outputs are generated similarly as before. Space and time complexities are now

$$\text{space} = 2p\lceil n/w \rceil + r\lceil p/w \rceil \text{ words} \tag{5}$$

$$\text{time} = C_3 p + C_4 r \text{ steps.} \tag{6}$$

## 3. Evaluation of Boolean functions on a walk through M-ary Decision Diagram (MDD)

Binary decision diagrams (BDD), ordered binary decision diagrams (OBDD) and reduced ordered binary decision diagrams (ROBDD) are well known representation of Boolean functions in a form of a directed acyclic graph [3]. Whereas ROBDD is canonical (unique) representation for any given function, in case of incomplete Boolean functions we may have apparently more choices.

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 ROBDDs belongs among NP-hard problems: the size of the ROBDD depends on variable ordering and there are $n!$ possible orderings of $n$ variables. A heuristic approach can be used in search for near optimal orderings [4]. Even though the upper bounds on the OBDD's size for general Boolean functions are not too encouraging, many practical functions do have a reasonable BDD size.

M-ary decision diagrams are straightforward generalization of BDDs. They have two types of nodes: decision and terminal nodes. Decision node $L$ is testing $M$-ary variable var($L$) and its outgoing edges are marked by its values 0, 1, …, $M$-1. The terminal node assigns a single value from $Z_M$ (generally $Z_R$ , $R \neq M$) to output $y = F_n(x_1, x_2, \ldots, x_n)$. Ordered MDDs are better suited to evaluation of Boolean functions as the walk through them can be much shorter than through OBDDs, depending on the value of $M$. If the $M$-ary variable is coded in $m$ bits, we have to visit at most $\lceil n/m \rceil$ nodes in the same number of steps, so that in the worst case

$$\text{time} = \lceil n/m \rceil \text{ steps.} \tag{7}$$

Each of $N$ nodes in OMDD is described by a table with $M \leq 2^m$ items. Each item has a format indicator (decision/ terminal node) and then either a pointer to a successor node $\lceil \log_2 N \rceil$ bits wide or the output value $r = \lceil \log_2 R \rceil$ bits. The size of the data structure is therefore

$$\text{space} = NM\,[1 + \max(\lceil \log_2 N \rceil + r)] \text{ bits.} \qquad (8)$$

## 4. Iterative disjunctive decomposition, BDDs, and linked tables

Evaluation of Boolean functions in software could rely on the full map stored in the memory. In embedded systems is this approach acceptable for about less than 10 variables. For several tens of variables we have to use more compact data structures and one way how to obtain them makes use a disjunctive decomposition of original functions. The basic idea is shown at Fig.1.

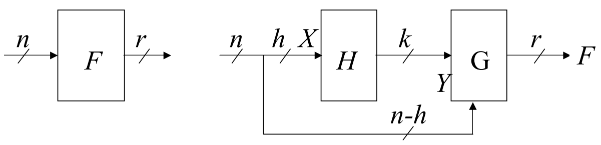

**Fig.1. Disjunctive decomposition of multiple output Boolean function *F* of *n* variables**

The original function $F$ is split into two functions $H$ and $G$, so that $F = G(Y, H(X))$, where multi-valued variables $X$, $Y$, $H$ and $G$ are binary coded using $h$, $n-h+k$, $k$ *and* $r$ bits, respectively:

$$X \in Z_2^{\,h},\ Y \in Z_2^{\,n-h+k},\ H \in Z_2^{\,k}, \text{ and } G \in Z_2^{\,r}, \qquad (9)$$

Fig.1. Of course, we are interested only in non-trivial decompositions for which $k < h$, such that descriptions of functions $H$ and $G$ are more thrifty in memory space than the description of original function $F$, i.e.

$$k\,2^h + r\,2^{n-h+k} \le r\,2^n. \qquad (10)$$

We prefer tables describing $H$ and $G$ to have the same size, to be stored in the same memory area (e.g. two table items in one word). This requirement translates to

$$h = k + n - h \qquad (11)$$

and eq. (10) is then rewritten into

$$2^k \le r2^h/(r + k). \qquad (12)$$

The lower a value of $k$ (with values $n$, $h$, $r$ fixed), the better. In a special case $k=r$, eq.(12) turns to $k \le h - 1$.

The value of $k$ cannot be selected at will, it is given by complexity of the function under consideration. The minimum value of $k$ is given by decomposition Theorem 1 [5], which under notation (9) and according to Fig.1 says:

*Theorem* 1.

Function $F$ is decomposable into

$$F = G(Y, H(X))$$

if and only if the value of $2^k$ is equal or greater than the number of distinct *sub-functions* of $n-h$ variables. (Note: A sub-function $f_{n-h}$ of $n-h$ variables is an instance of function $F_n$ with $h$ remaining variables fixed at certain values, [6]).

In the following sections we will use a technique based on bottom-up heuristic construction of BDDs, which uses a concept of sub-function, namely enumeration and counting distinct sub-functions in the set of all $2^h$ possible sub-functions. Then we will also need to count separately distinct *non-constant* sub-functions.

Decomposition shown at Fig.1 can be repeated iteratively with functions $H$ and $G$. The result will be a cascade of four function blocks (cells). If we go on, the cascade will be ultimately composed of n cells, each cell with one vertical input and cells interconnected horizontally to one another. The procedure of obtaining this cascade will be referred to as iterative disjunctive decomposition. We will illustrate it on a small example.

The map of 2 Boolean functions of 4 variables is specified by the leftmost map at Fig.2. Four function values can be considered as sub-functions of 0 variables. Counting sub-functions of a single variable gives the result

$$[x1] = 5/4,\ [x2] = 5/5,\ [x3] = 6/6,\ [x4] = 6/2,$$

where [x] is the number of all/non-constant distinct sub-functions of variable x. The first parameter will be also denoted later by an {integer}. According to previous consideration, we should select a variable x4 in the first step as there are only two (non-constant) sub-functions. (As we will see later, constant sub-functions do not count in software evaluation methods.) Sub-functions of variable x4 defined as

$$F\,|_{\,x3\ x2\ x1\ \in\ \{000,\ 001,\ \ldots,\ 111\}}$$

are 00, 11, 22, 33, 30, 12. E.g. sub-function "30"means

$$f(x4) = \begin{cases} 3 & \text{if } x4 = 0 \\ 0 & \text{if } x4 = 1. \end{cases} \qquad (13)$$

Enumeration of sub-functions is carried out next:

00 := 0, 11 := 1, 22 := 2, 33 := 3, 30 := 4, 12 := 5 (14)

and using the new sub-functions' IDs, the map of a residual function of 3 remaining variables is obtained. The similar decomposition step is carried out with this residual function, variable x1 is selected, [x1] = 4/3 sub-functions are identified, etc. Finally we end up with 1 sub-function of four variables – the original function:

$$[x4, x1, x3, x2] = 1/1.$$

The whole procedure and the resulting cascade are illustrated in Fig.2. The cascade is constructed from the end to the beginning and the process will be clarified further at the discussion of a related BDD.

Let us note, that incomplete functions can be decomposed the same way. However, the enumeration process must be done more carefully, because sub-functions can also be incomplete and can be combined with complete constant or non-constant sub-functions in different ways to reduce the total count as much as possible.

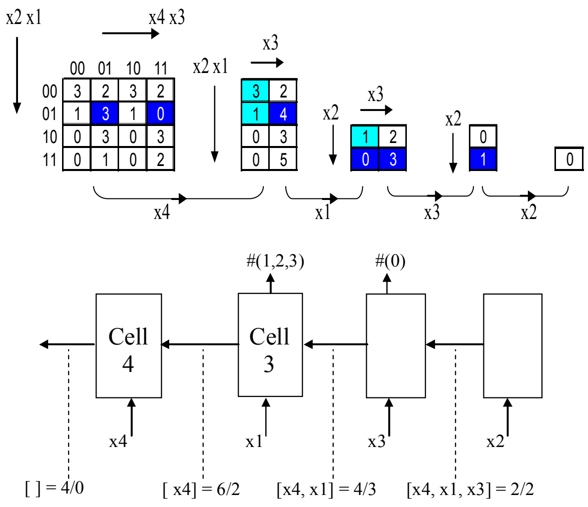

**Fig. 2. Iterative disjunctive decomposition and an associated non-redundant cascade**

If we realize that a non-constant sub-function of a single binary variable corresponds to a switch (see eq. 13), we can easily convert our decomposition procedure to construction of a related BDD. The BDD is constructed from the bottom up. The number of nodes is given by the number of single-variable sub-functions in all decomposition steps together, with exception of constant sub-functions. The same constant value for either value of the test variable does not require a decision, and the relevant edge can thus be terminated.

The difference between the BDD on left and the cascade of cells on the right side of Fig. 3 is apparent: sub-functions IDs are communicated between layers of BDD coded in "one-hot" edge style; on the other hand, compact code for IDs is used between neighbor cells of the cascade. One layer of the BDD is implemented as one cell of the cascade. (We assume OBDDs in which the same variable controls all the nodes in one layer).

The cells in the cascade are described by maps that can be obtained by reversing the assignments from the enumeration process. E.g. the cell 4 implementing 6 sub-functions (14)

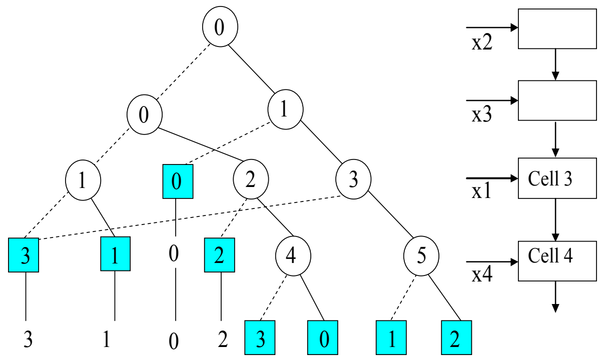

**Fig. 3. Construction of BDD by means of iterative disjunctive decomposition**

$$00 := 0,\ 11 := 1,\ 22 := 2,\ 33 := 3,\ 30 := 4,\ 12 := 5$$

is described by the left table 1. We can reduce the size of this table by omitting the constant sub-functions (the right table 1). The constant sub-functions will be identified in previous cell 3 and the evaluation can stop right there. The only overhead is a format indicator (FI) in every table item, specifying whether the item is a final value or an index to an item in the next cell table. The immediate function values evaluated earlier in the cascade are denoted by "#" in Fig.2.

**Table 1. Description of the cell 4 in Fig. 2**

| sub ID | x 4: 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 4 | 3 | 0 |
| 5 | 1 | 2 |

| sub ID | x 4: 0 | 1 |
|---|---|---|
| 4 | 3 | 0 |
| 5 | 1 | 2 |

The cells in the cascade in Fig.2 can be combined together to speed up the evaluation. The larger cells can be controlled by two variables simultaneously. Unfortunately the size of cell tables will differ: cell 4 and 3 will be described by table 16 x 2 bits, other pair of cells by table 4 x 2 bits, altogether 40 bits, what is more than the size of the original table (16 x 2 bits). Thus two "linked tables" obtained by unifying cells do no good in this special case. Generally functions used in practice are typically sparse – have many don't cares or attain a constant value in large sub-domains. These functions do have a space efficient decomposition, typically into a homogenous cascade with all the cells of the same size, [4]. The cell tables of the same size are desirable if the width of a computer word is large enough to accommodate table items indexed by the same ID in two or more cells.

## 5. 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 2.

**Table 2. 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 incomplete Boolean functions, which are after minimization described by Boolean expressions. The number of terms in AND arrays are p = 31 and 53. The size in bytes gives memory space $r2^n$ required for storing full function tables.

Iterative decomposition (done by an exhaustive search) used the selection of those two variables at a time that produced the minimum number of sub-functions. The PLA1 was implemented by the cascade of 6 cells, Fig. 4a 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.4b. 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.

As far as PLA2 is concerned, computer-generated cascades are shown in Fig. 5 and 6. The cascade at Fig. 5b 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.

We can also split output variables into two halves and then decompose them separately. The result for PLA2 is shown at Fig. 6. 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.

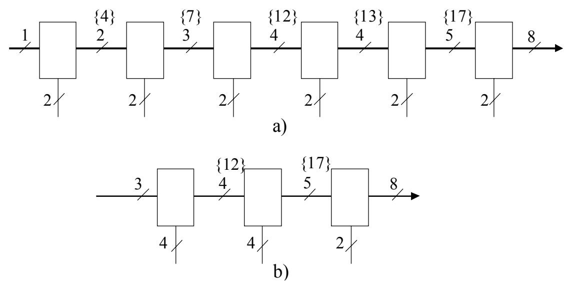

**Fig. 4. Two cellular cascade implementations of PLA1**

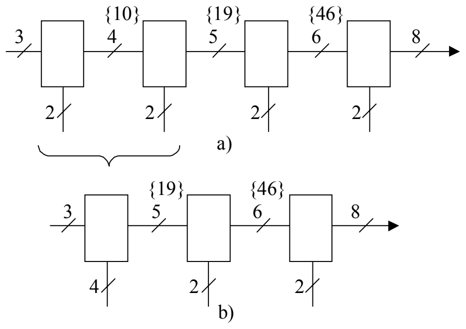

**Fig. 5. Cascade of 4 or 3 cells for PLA2**

## 6. 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.

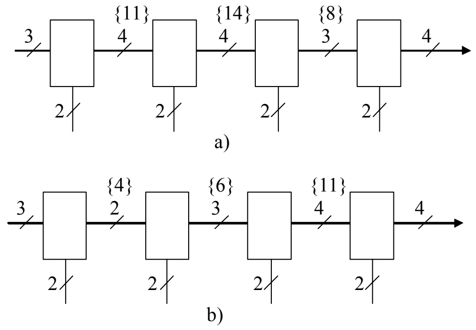

**Fig.6. Two parallel cascades implementing PLA2.**

The case study of PLA1 and PLA2 offered the size of data structures and speed of evaluation as given in

Table 3. 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).

Even though we cannot draw general conclusions from one case study, the method of linked tables seems to be well suitable for trade-offs between speed of evaluation and required memory space. It seems to have a potential for high performance and customization.

**Table 3. Software implementations of PLA1 and PLA2**

| | PLA emulation AND + 0R | | 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 |

The method of iterative disjunctive decomposition suggested for construction of linked tables can also be used for creation of OBDDs or OMDDs. Whereas OBDDs would require up to *n* table look-ups for evaluation a single- or multiple-output Boolean function, OMDDs could reach similar performance as linked tables. Comparison of space requirements is still to be done.

Future research will be oriented to the use of evolutionary techniques for iterative decomposition of complete as well as incomplete Boolean functions of many variables and to their hardware acceleration. Also an efficient procedure for finding sub-optimal OMDDs for a set of Boolean functions given by expressions would be very valuable.

## 7. References

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

[2] 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.

[3] H.R Andersen, An Introduction to Binary Decision Diagrams. Lecture notes for 49285 Advanced Algorithms E97, http://www.itu.dk/~hra/notes-index.html

[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] H. A Curtis, *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, 1962.

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

## Acknowledgement

This research has been carried out under the financial support of the research grants “Architectures of Embedded Systems Networks”, Grant Agency of Czech Republic GACR GA102/05/0467, 2005-2007 and “Security-Oriented Research in Information Technology”, MSM 0021630528.