# A New Data Structure for Shannon Decomposition

Prof. Dr. Jörg Keller, FernUniversität-GHS Hagen, FB Informatik, D-58084 Hagen, Germany
Dipl.-Inf. Milan Manasijevic, FernUniversität-GHS Hagen, FB Informatik, D-58084 Hagen, Germany

## Abstract

When modeling a dependable computer system of components that can fail, one often expresses the structure in the form of a boolean function $f$, where each variable represents one component. Function $f$ is normally given as a polynomial. To compute reliability parameters from the components' failure probabilities, one transforms boolean operators in $f$ into arithmetic operators. This transformation can lead to an exponential growth of the formula if monomials are not disjunct. This leads to huge evaluation times. To keep formula size acceptably small, one can apply an orthogonalization algorithm first. These algorithms are time– and space–consuming because of the huge number of intermediate boolean functions that must be represented and processed. We will concentrate on Shannon decomposition and present a data structure to represent intermediate functions in a compact way that allows fast execution of a decomposition step.

We sketch an implementation and give results of our experiments. The experiments show that even for a polynomial with 272 monomials in 26 variables, the time for decomposition is at most 3 seconds on a conventional personal computer. Hence, a system for reliability modeling, e.g. SyRePa, can now be used (almost) interactively, which raises productivity, allows fast and iterative changes of the systems according to the findings of the previous analysis, and thus hopefully improves the outcome of the modeling effort.

## 1 Introduction

We consider computer systems of several components where each component has a probability of failure. To compute the unavailability of such a system, i.e. the probability that that the system fails, one derives from the system's structure the boolean function of the fault-tree, a boolean polynomial, and transforms it into an arithmetic expression of the failure probabilities. To obtain an expression of acceptable length, one applies an orthogonalization algorithm such as a Shannon decomposition to the polynomial before transforming it. In the course of the decomposition, many intermediate polynomials are generated.

We investigate representations of boolean functions with regard to their space efficiency and their suitability to perform decomposition steps efficiently. We propose a new data structure which is compact and allows for the exploitation of word-level parallelism during the decomposition, thus accelerating it. We present an example implementation within a system for reliability modeling. We give experimental results which indicate that for medium-sized systems (up to 25 components, fault-tree function with up to 272 monomials), the decomposition can be performed in a few seconds. This allows for the interactive design and tuning of a dependable computer system which is to yield a certain reliability. The data structure can easily be extended to handle arbitrary boolean polynomials.

The remainder of the paper is organized as follows. In Section 2, we detail the role of Shannon decomposition in reliability modeling and sketch the data structure used so far. In Section 3, we present our data structure and show how to efficiently perform a decomposition step with it. In Section 4, we give performance results for some benchmarks. Section 5 concludes the paper.

## 2 Preliminaries

We consider a computer system $S$ consisting of $k$ components, where each component $i$ has a probability $p_i$ of failure. We assume that the failure probabilities of the components are independent of each other. We assign a variable $x_i$ to each component $i$ with $x_i = 1$ if component $i$ fails. Then the structure of the system defines a boolean function $\varphi$ with $\varphi(x_1, \ldots, x_k) = 1$ if the system fails. The function $\varphi$ is called the structure function of the system $S$. It can be generated from a fault tree of $S$ and is normally given as a polynomial where no variable appears

| boolean | = | arithmetic |
|---|---|---|
| $\bar{x}$ | = | $1-x$ |
| $x \wedge x'$ | = | $x \cdot x'$ |
| $x' \vee x'$ | = | $x + x' - x \cdot x'$ |

**Table 1** Replacement of boolean operators by arithmetic operators

inverted. In order to compute from the $p_i = \text{Prob}(x_i = 1)$ reliability parameters of $S$ such as the unvailability, i.e. $E(\varphi) = \text{Prob}(\varphi = 1)$, one normally proceeds as follows (see e.g. [2]). First, the boolean operators in $\varphi$ are replaced by arithmetic operators according to the identities in **Table 1**. As a sidenote, the arithmetic operators were originally used by Boole in his book "Laws of Thought". Then, the variables $x_i$ are replaced by the probabilities $p_i$. Evaluation of the resulting formula computes $E(\varphi) = \text{Prob}(\varphi = 1)$.
Because of the third identity, the replacement can lead to an exponential growth in formula size if the monomials are not disjunct. As formula size directly influences evaluation time and numeric stability of the result, a short formula is needed. As an example, we consider the simple polynomial

$$\varphi(x_1, \ldots, x_4) = x_1 \wedge x_2 \vee x_2 \wedge x_3 \vee x_3 \wedge x_4 \ .$$

We first transform the disjunction of the first two monomials:

$$x_1 \wedge x_2 \vee x_2 \wedge x_3 = x_1 \cdot x_2 + x_2 \cdot x_3 - x_1 \cdot x_2 \cdot x_3 \ .$$

The complete transformation proceeds as follows:

$$\begin{aligned} & \varphi(x_1, \ldots, x_4) \\ = \ & (x_1 \cdot x_2 + x_2 \cdot x_3 - x_1 \cdot x_2 \cdot x_3) + x_3 \cdot x_4 - (x_1 \cdot x_2 + x_2 \cdot x_3 - x_1 \cdot x_2 \cdot x_3) \cdot x_3 \cdot x_4 \\ = \ & x_1 \cdot x_2 + x_2 \cdot x_3 - x_1 \cdot x_2 \cdot x_3 + x_3 \cdot x_4 - x_1 \cdot x_2 \cdot x_3 \cdot x_4 - x_2 \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot x_4 \\ = \ & x_1 \cdot x_2 + x_2 \cdot x_3 + x_3 \cdot x_4 - x_1 \cdot x_2 \cdot x_3 - x_2 \cdot x_3 \cdot x_4 \ . \end{aligned}$$

There were only 5 boolean operators, but there are 11 arithmetic operators.
One avoids the blow-up in size if one applies an orthogonalization algorithm to $\varphi$ first. Application of such an algorithm also increases formula size but at a much smaller scale. Two commonly used algorithms are Abraham's algorithm [1] and Shannon decomposition (see e.g. [2] for an overview). Both have their advantages, and both have been cited to be "faster" than the other due to some (out of plenty) improvements. We therefore refrain from a comparison but concentrate on the latter.
A Shannon decomposition on $\varphi$ consists of several steps according to the rule

$$\varphi(x_1, \ldots, x_n) = x_i \wedge \varphi(x_i = 1) \vee \bar{x_i} \wedge \varphi(x_i = 0) \ .$$

In $\varphi(x_i = 1)$, one checks whether some of the monomials can be eliminated because they are absorbed by others. In $\varphi(x_i = 0)$, one deletes all monomials that contain $x_i$ and checks whether some additional variables can be factored out because they are part of all remaining monomials.
For our example, we apply the rule with $i = 1$:

$$\begin{aligned} \varphi(x_1, \ldots, x_4) &= x_1 \wedge x_2 \vee x_2 \wedge x_3 \vee x_3 \wedge x_4 \\ &= x_1 \wedge (x_2 \vee x_2 \wedge x_3 \vee x_3 \wedge x_4) \vee \bar{x_1} \wedge (x_2 \wedge x_3 \vee x_3 \wedge x_4) \qquad (1) \\ &= x_1 \wedge (x_2 \vee x_3 \wedge x_4) \vee \bar{x_1} \wedge x_3 \wedge (x_2 \vee x_4) \qquad (2) \end{aligned}$$

In $\varphi(x_1 = 1)$, the term $x_2$ absorbed $x_2 \wedge x_3$. In $\varphi(x_1 = 0)$, the variable $x_3$ could be factored out. The resulting formula has 8 boolean operators and can be transformed in a formula with an equal number of arithmetic operators.
Like any orthogonalization algorithm, the Shannon decomposition is a resource-consuming task itself, which could hinder its applicability in practice. Hence, it must be performed efficiently. To achieve this goal, a data structure is needed to represent the intermediate functions in the decomposition process. This representation must be very

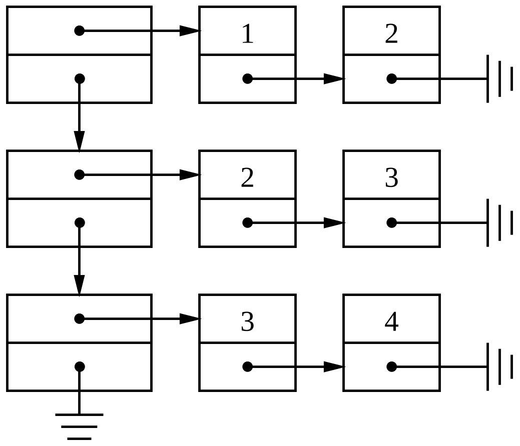

**Figure 1** Representation of example polynomial by linked lists

compact because of the large number of intermediate functions, and must allow each decomposition step to be performed fast.

A usual data structure to represent boolean polynomials is a linked list of all monomials, where each monomial is represented by a linked list of its variables. The representation of our example function $\varphi$ is depicted in **Figure 1**. Intermediate functions are polynomials with a number of variables factored out. The variables factored out can also be represented by a linked list.

## 3 New Data Structure

We will now present a more compact data structure that allows exploitation of word-level parallelism during the Shannon decomposition. Let $B = \{0,1\}$. Given is a set of $l$ monomials $m_i$, $1 \leq i \leq l$, in $k$ boolean variables $x_1, \ldots, x_k$, where no variable appears negated in any monomial. We represent each monomial $m_i$ by a unique bit vector $s_i \in B^k$, denoted by $s_i := \mathrm{vec}(m_i)$, such that $s_{i,j} = 1$ if and only if variable $x_j$ appears in monomial $m_i$. The inverse function to vec is mon, i.e. $m_i = \mathrm{mon}(s_i) := x_1^{s_{i,1}} \cdots x_k^{s_{i,k}}$, where we define $x^1 := x$ and $x^0 := 1$. Then $\mathrm{mon}(\mathbf{0}) = 1$, where $\mathbf{0}$ represents a vector of zeroes (of appropriate length). For completeness we also define a function $\overline{\mathrm{mon}}$ for negated variables, i.e. $\overline{\mathrm{mon}}(s_i) := \overline{x_1}^{s_{i,1}} \cdots \overline{x_k}^{s_{i,k}}$. We represent the set of monomials by a $(l,k)$-bitmatrix $S = (s_1, \ldots, s_l)^T$, i.e. row $i$ of $S$ represents monomial $m_i$. For example, the monomials of our example function $\varphi = x_1 \wedge x_2 \vee x_2 \wedge x_3 \vee x_3 \wedge x_4$ are represented by

$$S = \begin{pmatrix} 1100 \\ 0110 \\ 0011 \end{pmatrix} .$$

Boolean operators applied to bit vectors are to operate bitwise. Then, for two vectors $a, b \in B^k$, let $\mathrm{red}(a,b) := a \wedge \overline{b}$. The function red sets in $a$ those bits to zero that are 1 in $b$.

### 3.1 Representation of certain functions

The tuple $t = (v_+, v_-, r) \in B^k \times B^k \times B^l$ represents the function

$$\begin{aligned} F(t) \quad &:= \quad \mathrm{mon}(v_+)\overline{\mathrm{mon}}(v_-) \left( \bigvee_{1 \leq i \leq l, r_i = 0} \mathrm{mon}(\mathrm{red}(s_i, v_+ \vee v_-)) \right) \\ &= \quad x_1^{v_{+,1}} \cdots x_k^{v_{+,k}} \cdot \overline{x_1}^{v_{-,1}} \cdots \overline{x_k}^{v_{-,k}} \cdot \left( \bigvee_{1 \leq i \leq l, r_i = 0} \ \bigwedge_{1 \leq j \leq k, v_{+,j} = v_{-,j} = 0} x_j^{s_{i,j}} \right) . \end{aligned}$$

This means, that $v_+$ and $v_-$ denote the non-negated and negated variables that have been factored out of the inner polynomial. The inner polynomial is formed out of monomials from $S$, where monomials $m_i$ marked by $r_i = 1$ in the *row vector* $r$ are ignored. In each monomial taken, the variables factored out are ignored.
In our example, the function

$$\overline{x_1} \wedge \varphi(x_1 = 0) = \overline{x_1} \wedge x_3 \wedge (x_2 \vee x_4)$$

as obtained in (2) is represented by

$$\begin{array}{rcll} v''_+ & = & 0010 & \text{because } x_3 \text{ is factored out,} \\ v''_- & = & 1000 & \text{because } \overline{x_1} \text{ is factored out,} \\ r'' & = & 100 & \text{because the first monomial } (x_1 \wedge x_2) \text{ has value zero in } \varphi(x_1 = 0). \end{array}$$

The following properties hold:

1. The polynomial formed by disjunction of all monomials is represented by $F(\mathbf{0}, \mathbf{0}, \mathbf{0})$.
2. If $(v_+ \wedge v_-) \neq \mathbf{0}$, then $F(t) = 0$, i.e. if a factored out variable appears both in negated and non-negated form.
3. If $r = \mathbf{1}$, then $F(t) = \text{mon}(v_+)\overline{\text{mon}}(v_-)$. This is a deviation from standard conventions, where $\vee_{i \in I} a_i = 0$ if $I = \emptyset$ and $\wedge_{i \in I} a_i = 1$ if $I = \emptyset$. The reason for this deviation is that $r = \mathbf{1}$ will — if we only apply operations defined below — only occur if we are able to factor out all variables of the inner polynomial.
4. If $r_i = 0$ and $\text{red}(s_i, v_+ \vee v_-) = \mathbf{0}$, then the inner polynomial is 1 and $F(t) = \text{mon}(v_+)\overline{\text{mon}}(v_-)$.

## 3.2 Decomposition Step

To perform one step of a Shannon decomposition for a function $F(t)$, one chooses a variable $x_j$ and replaces $F(t)$ by $x_j F(t)(x_j = 1) \vee \overline{x_j} F(t)(x_j = 0)$. We derive representations $t' = (v'_+, v'_-, r')$ and $t'' = (v''_+, v''_-, r'')$ such that $F(t') = x_j F(t)(x_j = 1)$ and $F(t'') = \overline{x_j} F(t)(x_j = 0)$.
Then obviously $v'_+ = v_+ \vee \text{one}(j)$, where $\text{one}(j)$ is a $k$-bit vector consisting of $j-1$ zeroes followed by a one and $k - j$ zeroes. Further, $v'_- = v_-$ and $v''_- = v_- \vee \text{one}(j)$.
We obtain $r'$ by applying a check for absorption:

$$t' = (v'_+, v'_-, r') = \texttt{absorb}((v'_+, v'_-, r)) \ .$$

Checking for possibilites to factor a variable out is not necessary. However, this check can be applied to detect a monomial $\text{mon}(\mathbf{0})$ and to transform the function into a form as given in remark 3.
In our example, $F(\mathbf{0}, \mathbf{0}, \mathbf{0})$ represents $\varphi$ in accordance to remark 1. To obtain $x_1 \wedge \varphi(x_1 = 1)$, we set $v'_+ = v_+ \vee \text{one}(1) = 1000$ and $v'_- = v_- = \mathbf{0}$, thus

$$F(v'_+, v'_-, r) = x_1 \wedge (x_2 \vee x_2 \wedge x_3 \vee x_3 \wedge x_4)$$

as in (1). The check for absorption finds that the first monomial $x_2$ absorbs the second, $x_2 \wedge x_3$, i.e. $r' = 010$. Hence,

$$x_1 \wedge \varphi(x_1 = 1) = F(v'_+, v'_-, r') = x_1 \wedge (x_2 \vee x_3 \wedge x_4) \ .$$

To obtain $r''$, we first remove all monomials which included $x_j$ in $t$, i.e.

$$r''_i = \begin{cases} 1 \text{ if } r_i = 0 \text{ and } \text{red}(s_i, v_+ \vee v_-)_j = 1 \\ r_i \text{ otherwise.} \end{cases}$$

Then, we check whether additional variables can be factored out. Hence,

$$t'' = (v''_+, v''_-, r'') = \texttt{factor}(v_+, v''_-, r'') \ .$$

In our example, $v_+ \vee v_- = \mathbf{0}$, therefore $\text{red}(s_i, v_+ \vee v_-)_j = s_{i,j}$. Consequently, for $j = 1$, the first monomial must be removed, and $r'' = 100$. We already know that $v''_- = 1000$. As both remaining monomials contain $x_3$, operation `factor` will deliver $v''_+ = 0010$. Hence,

$$\overline{x_1} F(t)(x_1 = 0) = F(v''_+, v''_-, r'') = \overline{x_1} \wedge x_3 \wedge (x_2 \vee x_4) \ .$$

Note that if $v_{+,j} = 1$ then $F(t'') = 0$ as expected, and if $v_{-,j} = 1$ then $F(t') = 0$. Note also that if $v_{+,j} = v_{-,j} = 0$ and $s_{i,j} = 0$ for all $i$ with $r_i = 0$, i.e. if $F(t)$ is independent of $x_j$, then $r' = r'' = r$, i.e. both new functions have identical inner polynomials. In all these cases, the result of the step should be $t$ and $t'$ and $t''$ deleted.
We will now describe how to perform the operations `factor` and `absorb` on our data structure.

## 3.3 Factoring out

Given a function $F(t)$, we can factor a variable $x_j$ out of the inner polynomial if it appears in all monomials. This is the case if and only if

$$f_j := \bigwedge_{1\le i\le l, r_i=0} \mathrm{red}(s_i, v_+ \vee v_-)_j = 1\ .$$

Hence, with

$$\begin{aligned} \mathrm{fac}(t) := (f_1,\ldots,f_k) &= \bigwedge_{1\le i\le l, r_i=0} \mathrm{red}(s_i, v_+ \vee v_-) \\ &= \mathrm{red}\left(\bigwedge_{1\le i\le l, r_i=0} s_i, v_+ \vee v_-\right)\ , \end{aligned} \tag{3}$$

we can form a representation $t' = (v'_+, v'_-, r')$, denoted by `factor`$(t)$, with $F(t') = F(t)$ and all possible variables factored out:

$$\begin{aligned} v'_+ &= v_+ \vee \mathrm{fac}(t) \\ v'_- &= v_- \\ r' &= \begin{cases} \mathbf{1} & \text{if } \exists i : r_i = 0 \text{ and } \mathrm{red}(s_i, v'_+ \vee v'_-) = \mathbf{0} \\ r & \text{otherwise.} \end{cases} \end{aligned}$$

In our example, we execute `factor`$(0, 1000, 100)$. We compute

$$\begin{aligned} \mathrm{fac}(t) &= \mathrm{red}\left(\bigwedge_{2\le i\le 3} s_i, v_+ \vee v_-\right) \\ &= (0010 \wedge 0111) \wedge (\overline{0000 \vee 1000}) \\ &= 0010 \wedge 0111 \\ &= 0010 \end{aligned}$$

and thus see that $x_3$ can be factored out.
Note that the definition of $r'$ ensures that if the inner polynomial in $F(t')$ contains a monomial 1, then the representation can be simplified (see also remarks 3 and 4 above).

## 3.4 Checking for absorption

A monomial $m_{i'}$ is absorbed by a monomial $m_i$, where $i \neq i'$, when each variable occuring in $m_i$ also occurs in $m_{i'}$. Then $m_{i'}$ need not be considered further. This is the case if and only if

$$\mathrm{red}(s_i, s_{i'}) = s_i \wedge \overline{s_{i'}} = \mathbf{0}\ .$$

Then, a monomial $i'$ in the inner polynomial of $F(t)$ is absorbed if and only if there exists an $i \neq i'$, $1 \le i \le l$, with $r_i = 0$ such that

$$\mathrm{red}(\mathrm{red}(s_i, v_+ \vee v_-), \mathrm{red}(s_{i'}, v_+ \vee v_-)) = \mathbf{0}\ .$$

Hence, we can form a representation $t' = (v'_+, v'_-, r')$, denoted by `absorb`$(t)$, with $F(t') = F(t)$ and all possible monomials absorbed:

$$\begin{aligned} v'_+ &= v_+ \\ v'_- &= v_- \\ r'_{i'} &= \begin{cases} 1 & \text{if } r_{i'} = 0 \text{ and } \exists i : r_i = 0 \text{ and } \mathrm{red}(\mathrm{red}(s_i, v_+ \vee v_-), \mathrm{red}(s_{i'}, v_+ \vee v_-)) = \mathbf{0} \\ r_{i'} & \text{otherwise.} \end{cases} \end{aligned}$$

In our example, we compute `absorb`$(1000, 0000, 000)$. We first calculate

$$\tilde{s_i} = \mathrm{red}(s_i, v_+ \vee v_-) \text{ for } i = 1,2,3$$

and obtain

$$\begin{aligned}\tilde{s_1} &= 0100 = s_1 \wedge \overline{1000}\,, \\ \tilde{s_2} &= 0110 = s_2\,, \\ \tilde{s_3} &= 0011 = s_3\,.\end{aligned}$$

We find

$$\text{red}(\tilde{s}_2, \tilde{s}_1) = 0110 \wedge \overline{0100} = 0010 \text{ and } \text{red}(\tilde{s}_3, \tilde{s}_1) = 0011 \wedge \overline{0100} = 0011\,,$$

thus, monomial 1 is not absorbed. We find

$$\text{red}(\tilde{s}_1, \tilde{s}_2) = 0100 \wedge \overline{0110} = 0000\,,$$

thus, monomial 2 is absorbed by monomial 1. The test whether monomial 3 absorbs monomial 2 is not necessary. We finally find

$$\text{red}(\tilde{s}_1, \tilde{s}_3) = 0100 \wedge \overline{0011} = 0100 \text{ and } \text{red}(\tilde{s}_2, \tilde{s}_3) = 0110 \wedge \overline{0011} = 0100\,,$$

thus, monomial 3 is not absorbed.
Note that if $\text{red}(s_i, v_+ \vee v_-) = \text{red}(s_{i'}, v_+ \vee v_-)$, i.e. if two monomials in the inner polynomial are identical, then both are removed. However, this cannot happen in our algorithm as this would need the prerequisite that $t$ was the result of a decomposition step of a representation where absorption had not been applied.
Note also that factoring and absorption are independent operations, because any variable in $m_i$ also was present in $m_{i'}$. Thus, by absorbing $m_{i'}$, the decision whether this variable can be factored out is not influenced.

## 3.5 Complete Shannon Decomposition

As our application is the analysis of fault-trees, we start with a DNF given by $t = (\mathbf{0}, \mathbf{0}, \mathbf{0})$, and first apply our absorb and factor operations. Then we perform a decomposition step of $F(t)$ with respect to a variable $x_j$ not marked in $v_+$ or $v_-$, i.e. which is (possibly) used in the inner polynomial. We continue with the resulting representations $t'$ and $t''$. Thus, we get a pool of representations upon which we perform one decomposition step each. We also get a second pool of representations $t_1, t_2, \ldots$ where a decomposition step cannot be successfully applied anymore. We have finished as soon as our first pool is empty. Our resulting function is

$$F(t_1) \vee F(t_2) \vee \ldots\,.$$

Note that the choice of the variable $x_j$ in any step is not depending on the particular data structure we use and thus any heuristic can be used. Some heuristics might even be simplified by our data structure. E.g. a popular heuristic to find the next variable used to decompose is to search for monomials containing few variables, because they have the potential to absorb many others. Among the variables in the shortest monomials, we choose one by an arbitration rule.
As our monomials are represented by bit vectors, the above heuristic requires to count the number of ones in a bit vector. Some processors, such as the Alpha, provide an assembler instruction that gives the number of ones in a data word. However, as not all processors have this, a portable solution relies on a software formulation. A straightforward approach is to have a function `cntones` that shifts the vector bit after bit and counts the ones in position zero.
A much faster solution relies on tables that contain the number of ones for bit vectors of a fixed length. Let `onecnt` be an array such that for $x$ being an integer, `onecnt`$[x]$ contains the number of ones of the binary representation of $x$. The array can be initialized using the function above. Note that for array sizes up to $2^{255}$, the binary representation of $x$ will contain at most 255 ones, and thus each element of `onecnt` can be an `unsigned char`. If the size of the array is $2^{16}$, and we use at most 32 variables, then function `cntones` can be rewritten as

```
unsigned int cntonesfast(w)
unsigned long int w;  /* the word of which the ones are counted */
{ return (int)onecnt[w & 0xffff] + (int)onecnt[(w >> 16) & 0xffff]; }
```

This function only needs one shift, two bitwise AND operations, two array indexings, and one addition. In contrast, the function `cntones` would have needed 32 bitwise AND operations, shifts and additions, plus the overhead for the loop.

### 3.6 Efficiency

Our data structure $t$ consists of three bit vectors. Two of them have length $k$, the number of variables, on has length $l$, the number of monomials in the polynomial to be decomposed. Modern microprocessors operate on 32-bit words. Hence, if $k \leq 32$, then vectors $v_+$ and $v_-$ fit into one word. The same holds for each $s_i$. For larger numbers of variables, we use a small but fixed number of words per vector. The row vector is stored in an appropriate number of words. The matrix $S$ needs $l$ words, but is stored only once, while the data structure $t$ occurs for each intermediate function.
Hence, our data structure is much more compact than a list based representation which needs at least one word to store a literal plus one word to store a pointer to the next literal. Also, linking the monomials requires one word per monomial. The next section will give more details about the memory consumption in our implementation.
Monomials that are stored in a single word allow the exploitation of word-level parallelism. This is obvious when considering the operation red. It is most striking in equation (3) where one can see that by a single loop over all monomials, where each iteration only needs one bit-oriented operation, all variables to be factored out have been determined. The check for absorption needs a nested loop, however the inner loops have an exit condition: as soon as an absorbing monomial is found, the inner loop can be quit. The check whether one monomial absorbs another only needs a few bit-oriented operations.
We have to take care with the loops. Their number of iterations is oriented towards the length of the row vector, which is the number $l$ of monomials in the polynomial $\varphi$ to be decomposed. In the intermediate functions fewer and fewer monomials will be used. Thus, many of the iterations will simply check that $r_i = 1$, but they are still lost time. Hence we extend the row vector by a so-called *row index vector*:
We consider $r$ as $z = \lceil l/b \rceil$ blocks of $b$ bits each, i.e. $r = b_1, \ldots, b_z$ with $b_i \in B^b$ for $1 \leq i \leq z$. If $l$ is not a multiple of $b$, we artificially pad $r$ by appending an appropriate number of ones. The *row index vector* $ri \in B^z$ of $r$ is defined by

$$ri_i = \begin{cases} 1 & \text{if } b_i = \mathbf{1} \\ 0 & \text{otherwise.} \end{cases}$$

Then, instead of looping over the row vector, one can use the row index vector to skip $b$ iterations if the corresponding index vector entry is 1, i.e. if all these $b$ iterations would only have tested that the corresponding rows are not used anymore. A satisfactory choice for $b$ is 32, i.e. one word of the row vector is skipped.
Our method can be easily extended to decompose polynomials in which negated variables appear. Each monomial then $m_i$ is represented by two bit vectors $s_i^+$ and $s_i^-$. Bit vector $s_i^+$ is defined similar to $s_i$, bit vector $s_i^-$ is defined such that $s_{i,j}^- = 1$ if and only if $m_i$ contains $\overline{x_j}$. Then $m_i = \text{mon}(s_i^+)\overline{\text{mon}}(s_i^-)$. The definition of $F(t)$ is extended accordingly. Hence, the memory consumption of $S$ is doubled, but the representations $t$ remain as they are. The checks for absorption and extraction now work on $s_i^+$ and $s_i^-$, hence their runtime doubles. In a decomposition step, we now have to check for absorption *and* extraction when generating $t'$ and $t''$. Hence, the total runtime is increased by a factor of four.

## 4 Experiments

We implemented a tool for Shannon decomposition based on our proposed data structure. It was programmed using MS Visual C++. The tool is embedded in a system for reliability modeling, SyRePa [3]. The system runs on a PC with a Pentium II MMX / 300 MHz microprocessor, 512 kB Cache Memory and 64 MB main memory. We compare our implementation with the previous implementation of Shannon decomposition in SyRePa with respect to runtime and memory consumption. As the resulting function is identical for both implementations (see remark in subsection 3.5), we do not compare formula sizes. As benchmarks, we used 7 formulae which are described in **Table 2**. The complete formulae can be found at
`http://ti2server.fernuni-hagen.de/~jkeller/formula.html`.
For the 5 small benchmarks, runtime was well below one second. For the last two benchmarks, runtime was 3 and 1.5 seconds, respectively. Hence, even for larger systems, interactive modeling and refining should be possible.
For benchmark 6, which the largest and hence the most significant, we also ran the old version which needed 9 seconds. Interestingly, the runtime of the new implementation without use of the row index vector had a runtime of 15 seconds. The reason is that the average number of monomials in intermediate functions was 4.90, i.e. the

| benchmark | # variables | # literals | # monomials |
|---|---|---|---|
| 1 | 3 | 6 | 3 |
| 2 | 5 | 10 | 4 |
| 3 | 7 | 32 | 10 |
| 4 | 7 | 105 | 35 |
| 5 | 12 | 131 | 24 |
| 6 | 26 | 2475 | 272 |
| 7 | 26 | 927 | 130 |

**Table 2** Characteristics of Benchmark formulae

program spent most of its time looping over row vectors while only a few rows were still used. Hence, use of the row index vector is indeed necessary.

The comparison of the old and the new data structure in terms of memory requirements showed that it was comparable for the small benchmarks 1 to 5, with differences of up to 20% in either direction, depending on the benchmark. For the larger benchmarks 6 and 7 however, the amount of memory needed to store intermediate functions was halved by the new data structure. This enabled the functions' representations to be kept in the cache, which contributed to the runtime improvements.

We compared our implementation with the CAOS algorithm [4]. There, the largest benchmarks contain 74 monomials in 16 variables and 81 monomials in 14 variables, respectively. For both, the reported runtime on a PC 486 DX2-66 is 3.1 seconds. As our machine is about 4.5 times faster, this should translate to about 0.7 seconds. We need about twice that time for benchmark 7 with almost twice as many monomials, thus we conclude that our algorithm is at least as efficient as the CAOS algorithm. Unfortunately, memory consumption was not reported so that a comparison is not possible in that respect.

## 5 Conclusion

We have shown how to efficiently represent boolean polynomials that occur in the course of a Shannon decomposition. Our representation allows the decomposition to be performed fast enough to approach interactive design and redundancy minimization of a dependable computing system which is to guarantee certain reliability parameters.

## References

[1] Abraham, J. A.: An improved algorithm for network reliability. IEEE Transactions on Reliability, Vol. R–28 No. 1, 1979, pp. 58–61

[2] Schneeweiss, W. G.: Boolean Functions with Engineering Applications and Computer Programs. Berlin: Springer, 1989

[3] Schneeweiss, W. G.: SyRePa '89 – a package of programs for systems reliability evaluations. Informatik–Bericht 91, FernUniversität-GH Hagen, 1990

[4] Vahl, A.: Interaktive Zuverlässigkeitsanalyse von Flugzeug–Systemarchitekturen. Dissertation, Technische Universität Hamburg–Harburg, 1998