Algorithms and Data Structures for Manipulating Boolean Functions

Hiroyuki OCHI

December 1993

Algorithms and Data Structures for Manipulating Boolean Functions

Hiroyuki OCHI

December 1993

Abstract

Recent advances in very large scale integration (VLSI) technology have made it possible to realize larger and more sophisticated logic circuits. Today, it is hard to design logic circuits efficiently and correctly without using computer-aided design (CAD) systems. However, with the growth of the scale of VLSI, CAD systems have revealed their problem of increasing time and space for computation.

In this thesis, three topics concerning Boolean function manipulation are discussed in order to solve very large problems in CAD of digital systems. One is on high-speed generation of prime implicants of a given Boolean function. It has been studied by many researchers as the first step of two level logic minimization, which is one of the most classical and yet very important problem in CAD. The other two topics are on Boolean function manipulation based on ordered binary-decision diagrams (OBDDs), or simply Binary-Decision Diagram (BDD). BDD is a graph representation of Boolean functions proposed by Akers and developed by Bryant. BDDs have excellent properties which are useful to solve CAD problems symbolically, including that (1)BDD is a canonical representation of Boolean function, (2)Boolean operation is performed in time proportional to the size of BDD, using two hash tables, (3)size of BDDs is not large for many Boolean functions found in digital designs, etc. Boolean function manipulators based on Shared BDDs (SBDDs), or multirooted BDDs, implemented on workstations are appreciated their

usefulness in CAD systems.

In chapter 2, high-speed algorithms for generating prime implicants of a given Boolean function are discussed, and the use of a vector supercomputer is proposed. The proposed algorithms are based on the consensus expansion presented by Tison. The proposed algorithms are implemented efficiently on vector supercomputers by performing consensus expansion in breadth-first manner, and employing truth table representation of Boolean functions and map representation of a set of prime implicants. Table look-up technique is also employed to reduce the consensus expansion stages. The proposed algorithms are implemented on the vector supercomputer FACOM VP-400E at the Kyoto University Data Processing Center and compared with several other algorithms. For example, by the consensus expansion method with table look-up, all prime implicants of randomly generated 18-variable Boolean functions are generated in about 1.4 seconds on the average. As an application of the proposed algorithm, we will show the results related to the number of prime implicants of Boolean functions. We will show that Igarashi's conjecture on the maximum number of n-variable Boolean functions is true for $n = 5$ and 6, i. e., the maximum number of prime implicants of 5- and 6-variable Boolean functions are 32 and 92, respectively.

In chapter 3 and chapter 4, algorithms for manipulating SBDDs are discussed in order to manipulate very large SBDDs which cannot be manipulated on conventional workstations, and the use of breadth-first algorithm is proposed. The breadth-first algorithm consists of two parts; an expansion phase and a reduction phase. In the expansion phase, new nodes sufficient to represent the resultant Boolean function are generated in a breadth-first manner from the root-node toward leaf-nodes. In the reduction phase, the nodes generated in the expansion phase are checked in a breadth-first manner from nodes nearby leaf-nodes toward the root-

node.

In chapter 3, a high-speed algorithm for manipulating SBDDs which is suitable for vector supercomputers is proposed. Breadth-first algorithm is employed to vectorize manipulation, and actually almost all steps are vectorized, including hash table access which is efficiently vectorized using high-speed vector indirect store instruction of a vector supercomputer HITAC S-820/80. A Boolean function manipulator based on the proposed algorithm is implemented on the HITAC S-820/80 at the University of Tokyo, and experiments of constructing the SBDDs representing the Boolean functions of all the primary outputs and nets from a circuit description chosen from ISCAS'85 are performed. From these experiments, the vector acceleration ratio on the S-820/80 is 5.3 to 27.8. Compared with the results on the workstation Sun3/60 by Minato et al., our results are up to 130 times faster in the best case. In addition, as an example of applications of developed SBDD manipulator, a design verification system based on computation tree logic (CTL) model checker is implemented and the experimental results are shown.

In chapter 4, the use of secondary memory is discussed in order to manipulate SBDDs which are too large to be stored within main memory. In order to avoid random accesses to the secondary memory, level-by-level manipulation of Shared Quasi-reduced BDDs (SQBDDs) upon a breadth-first algorithm is employed. The use of garbage collection with sliding type compaction is also introduced to reduce page faults in succeeding manipulation. A Boolean function manipulator based on the proposed algorithm is implemented and evaluated on the workstation Sun SPARC Station 10 with 64 megabyte main memory and a one gigabyte hard disk drive connected via SCSI-2 standard interface. More than 50 million nodes can be allocated within one gigabyte virtual memory space, and as a result, an SQBDD with more than 12 million nodes representing all

the primary outputs of a 15-bit multiplier is constructed from a circuit description in about 5.6 hours. If the conventional SBDD manipulator is used instead, it is estimated that it would take about 1,900 hours. So we can say that our manipulator achieved about 330 times improvement in elapsed time. Furthermore, we made experiments using semiconductor extended storage instead of hard disk, and showed that the required time for the 15-bit multiplier is reduced to about 2.2 hours.

Contents

Chapter 1

Introduction

1.1 Background

Recent advances in very large scale integration (VLSI) technology have made it possible to realize larger and more sophisticated logic circuits. Today, it is hard to design logic circuits efficiently and correctly without using computer-aided design (CAD) systems. However, with the growth of the scale of VLSI, CAD systems have revealed their problem of increasing time and space for computation.

Among many steps of designing hardware, logic minimization is one of the most classical and yet very important problem. Two level logic minimization is the most basic problem in logic minimization. It is useful to optimize combinational circuit, and today it is very important to realize programmable logic array (PLA). Quine showed that the minimum two level formula can be derived from a set of prime implicants of a given Boolean function[Qui55]. McCluskey proposed a method that consists of two steps; (1)Generate all prime implicants of a given Boolean function, and (2)derive a minimum cover of the given Boolean function by the prime implicants[McC56]. Since the so-called Quine-McCluskey method was presented, various algorithms for generating all prime implicants of

a given Boolean function which are suited to computer processing have been proposed.

Nelson showed that when a product-of-sums representation of a Boolean function is expanded to a sum-of-products representation by means of the distributive law ($A(B+C) = AB + AC$) and some other primitive laws, all prime implicants of a given Boolean function are generated with, possibly, some non-prime implicants[Nel54]. Slagle et al. proposed a method for generating prime implicants from a product-of-sums representation by means of tree search[SCL70]. Kambayashi et al. proposed the clause selection method by combining Nelson's theory and Slagle's method, in which the searched tree is smaller than that of the Slagle's method[KOY79].

Tison showed that all prime implicants of a given Boolean function can be generated by consensus expansion[Tis67]. The consensus expansion is based on the following equation which holds for any Boolean function f :

$$f = \overline{x_i} f(x_i = 0) + x_i f(x_i = 1) + f(x_i = 0) f(x_i = 1)$$

where x_i is a Boolean variable in f. Using the equation repeatedly for every variable, f is expanded in a ternary tree fashion, and consequently, its all implicants are generated. In order to use the consensus expansion for generating all prime implicants, the removal of the generated non-prime implicants or the prevention of the generation of non-prime implicants is necessary. Morreale proposed an algorithm in which the generation of non-prime implicants is prevented by means of tagging functions[Mor70].

Thus, various methods for generating all prime implicants have been proposed. However, generating all prime implicants is intrinsically very time and space consuming, and it was difficult to generate all prime implicants of a Boolean function with more than a dozen or so variables. The computation time and the required memory space increase exponentially to the number of variables. There are n-variable Boolean functions which have $O(3^n/n)$ prime implicants[DF59].

Recentry, Kagatani et al. proposed the use of vector supercomputers for generating prime implicants, and presented two high-speed vector algorithms, called the variable-oriented expansion method which is based on the clause selection method and the ternary tree expansion method which is based on the consensus expansion method[Kag87]. A vector supercomputer is a highly pipelined supercomputer which is primarily used for large scale scientific and engineering computation. It yields more than a giga floating operations per second (GFLOPS) of computation power by executing uniform operations on array structured data. In order to support large-scale computation, it has a large main memory unit (usually a hundred mega bytes or more) and powerful load/store pipelines. The use of vector supercomputers for non-numerical applications had been proposed, including logic simulation by Ishiura et al.[IYY87]

Another important step of designing hardware is design verification. There are two methods for design verification; logic simulation and formal verification. Logic simulation is a method to detect design errors by simulating the behavior of a designed circuit for an input sequence. Logic simulation is now widely used for design verification, however, it has a problem that there may still be undetected errors even if simulation has finished successfully, because it is difficult to simulate a logic circuit for its all possible input sequences. To overcome this problem, formal design verification have been studied in recent years.

Formal design verification is to show the correctness of a designed logic circuit with respect to its specification of the circuit based on a formal system. Among several approaches for formal design verification, symbolic simulation and symbolic model checking has been proved their usefulness by many researchers in recent years. The performance of both symbolic simulation and symbolic model checking owes to Boolean function manipulator based on ordered binary-decision diagrams (OBDD), or

simply Binary-Decision Diagram (BDD), that is one of representations of Boolean functions.

Various representations of Boolean functions have been proposed, including truth table, Boolean formula, cube representation, etc. For example, truth table is the most simple representation and suitable for vector processing, but it takes $O(2^n)$ space and time to construct the representation for an n-variable Boolean function. On the other hand, Boolean formula has advantages such as easy operation and relatively small space to store, but tautology check or equivalence check is very difficult.

BDD is a graph representation of Boolean functions proposed by Akers [Ake78] and developed by Bryant[Bry86]. A BDD is a directed acyclic graph with two leaf (terminal) nodes labeled by 0 and 1. Every non-terminal node is labeled by a Boolean variable and has two outgoing edges labeled by 0 and 1. No Boolean variable appears more than once in every path of a BDD, and the variables appear in a fixed order in all the paths of a BDD. A BDD is defined as the graph obtained from binary decision tree by removing all redundant nodes and non-unique nodes (but one). BDDs have excellent properties which are useful to solve CAD problems symbolically, including (1)BDD is a canonical representation of Boolean function, (2)Boolean operation is performed in time proportional to the size of BDD, using two hash tables, (3)size of BDDs is not large for many Boolean functions found in digital designs, etc.

At present, subroutine packages, called Boolean function manipulators, based on Shared BDD (SBDD), or multirooted BDD, are implemented on workstations which support primary operations of Boolean function manipulation. Several techniques for implementation of Boolean function manipulators are proposed in order to reduce the time and the storage for manipulation, such as various attributed edges, automatic garbage collection, and so on. Variable ordering of BDD has also been studied

by many researchers to reduce the size of BDD. SBDD based Boolean function manipulators are now widely utilized in various applications of CAD systems, not only formal design verification, but also test generation, logic synthesis and so on, and even the use for other combinatorial problems has been studied.

Thus, Boolean function manipulators based on SBDDs implemented on workstations are appreciated their usefulness in CAD systems. However, according to the recent progress of the VLSI technology, it is required to manipulate larger and larger scale Boolean functions, which will exceed the computational power of workstations. In order to fulfill this requirement, the use of parallel machines or connection machines is studied.

1.2 Outline of the Thesis

In this thesis, three topics concerning Boolean function manipulation are discussed in order to solve very large problems in CAD of digital systems.

In chapter 2, high-speed algorithms for generating prime implicants of a given Boolean function are discussed, and the use of vector supercomputer is proposed. The proposed algorithms are based on the consensus expansion. The proposed algorithms are implemented efficiently on vector supercomputers by performing consensus expansion in breadth-first manner, and employing truth table representation of Boolean functions and map representation of a set of prime implicants. Table look-up technique is also employed to reduce the consensus expansion stages. The proposed algorithms are implemented on the vector supercomputer FACOM VP-400E at the Kyoto University Data Processing Center and compared with several other algorithms. For example, by the consensus expansion method with table look-up, all prime implicants of randomly generated

18-variable Boolean functions are generated in about 1.4 seconds on the average. As an application of the proposed algorithm, we will show the results related to the number of prime implicants of Boolean functions. We will show that the Igarashi's conjecture on the maximum number of n-variable Boolean functions is true for $n = 5$ and 6.

In chapter 3 and chapter 4, algorithms for manipulating SBDDs are discussed in order to manipulate very large SBDDs which cannot be manipulated by conventional workstations, and the use of breadth-first algorithm is proposed. The breadth-first algorithm consists of two parts; an expansion phase and a reduction phase. In the expansion phase, new nodes sufficient to represent the resultant Boolean function are generated in a breadth-first manner from the root-node toward leaf-nodes. In the reduction phase, the nodes generated in the expansion phase are checked in a breadth-first manner from nodes nearby leaf-nodes toward the root-node.

In chapter 3, a high-speed algorithm for manipulating SBDDs which is suitable for vector supercomputers is proposed. Breadth-first algorithm is employed to vectorize manipulation, and actually almost all steps are vectorized, including hash table access which is efficiently vectorized using high-speed vector indirect store instruction of a vector supercomputer HITAC S-820/80. A Boolean function manipulator based on the proposed algorithm is implemented on the HITAC S-820/80 at the University of Tokyo, and experiments of constructing the SBDDs representing the Boolean functions of all the primary outputs and nets from a circuit description chosen from ISCAS'85 [BF85] are performed. From these experiments, the vector acceleration ratio on the S-820/80 is 5.3 to 27.8. Compared with the results on the workstation Sun3/60 by Minato et al.[MIY90], our results are up to 130 times faster in the best case. In addition, as an example of applications of developed SBDD manipulator, a

design verification system based on computation tree logic (CTL) model checker is implemented and the experimental results are shown.

In chapter 4, the use of secondary memory is discussed in order to manipulate SBDDs which are too large to be stored within main memory. In order to avoid random accesses to the secondary memory, level-by-level manipulation of Shared Quasi-reduced BDDs (SQBDDs) upon a breadth-first algorithm is employed. The use of garbage collection with sliding type compaction is also introduced to reduce page faults in succeeding manipulation. A Boolean function manipulator based on the proposed algorithm is implemented and evaluated on the workstation Sun SPARC Station 10 with 64 megabyte main memory and a one gigabyte hard disk drive connected via SCSI-2 standard interface. More than 50 million nodes can be allocated within one gigabyte virtual memory space, and as a result, an SQBDD with more than 12 million nodes representing all the primary outputs of a 15-bit multiplier is constructed from a circuit description in about 5.6 hours. If the conventional SBDD manipulator is used instead, it is estimated that it would take about 1,900 hours. So we can say that our manipulator achieved about 330 times improvement in elapsed time. Furthermore, we made experiments using semiconductor extended storage instead of hard disk, and showed that the required time for the 15-bit multiplier is reduced to about 2.2 hours.

In chapter 5, the conclusion of this thesis and future problems are stated.

Chapter 2

Vector Algorithms for Generating Prime Implicants

2.1 Introduction

Generation of all prime implicants of a given Boolean function is a fundamental task in two-level logic minimization[Qui55, McC56, Pet60] which is an important process in logic design and logic synthesis. Various studies have been made on this subject, and many algorithms suitable for computer processing have been proposed[Nel54, SCL70, Mor70, Tis67, KOY79]. However, since the subject is intrinsically very time and space consuming, it is difficult to generate all prime implicants of a Boolean function with more than a dozen or so variables by means of these methods on conventional scalar processors. The computation time and the required memory space increase exponentially to the number of variables. There are n-variable Boolean functions which have $O(3^n/n)$ prime implicants[DF59]. For example, there are 16-variable Boolean functions which have more than two million prime implicants[Iga79].

In this chapter, the use of vector supercomputers for generating all prime implicants of a given Boolean function is considered. Several vector supercomputers have been developed for large-scale computation. They

are used mainly for numerical computations, but can be also used efficiently for non-numerical computations such as logic simulation[IYY87]. The function pipelines which support bit-wise logical operations on a word and the load/store pipelines which support various access modes are useful for non-numerical computations.

In this chapter, two high-speed vector algorithms, called the consensus expansion method with table look-up and the Morreale method with table look-up, are proposed. These algorithms are based on the consensus expansion[Tis67], and effective data structure and a table look-up technique are introduced in the implementation. In addition to these high-speed algorithms, another algorithm, called the extended consensus expansion method with table look-up, is also proposed in order to generate all prime implicants of a Boolean function of more variables within the limited main memory space. They are implemented on the vector supercomputer FACOM VP-400E at the Kyoto University Data Processing Center and compared with several other algorithms including two formerly proposed vector algorithms[Kag87]. We show that by means of the new algorithms, the generation of all prime implicants of a given Boolean function can be performed in much higher speed with a high acceleration ratio. For example, by the consensus expansion method with table look-up, all prime implicants of randomly generated 18-variable Boolean functions are generated in about 1.4 seconds on the average.

In the next section, several terms relating to Boolean functions will be defined and an overview of algorithms for generating prime implicants will be made. The new vector algorithms will be proposed in section 3, 4 and 5. In section 6, evaluation and discussions will be made. In section 7, it will be shown that the maximum number of prime implicants of 5- and 6-variable Boolean functions are 32 and 92, respectively. These results are obtained by the experiments using the developed program based on the

consensus expansion method with table look-up. Section 8 will appear as a conclusion.

2.2 Preliminaries

2.2.1 Boolean Function and Prime Implicant

A *Boolean variable*, denoted by x_i, is a variable which assumes a binary value 0 or 1. A *literal* l_i for a variable x_i is either x_i or $\overline{x_i}$ (the complement of x_i). An *n-variable Boolean function*, denoted by $f(x_1, \ldots, x_n)$ or simply f, is a mapping from $\{0,1\}^n$ to $\{0,1\}$. In the following, a Boolean variable and a Boolean function are sometimes simply referred to as a variable and a function, respectively.

A logical product of literals where literals appear at most once for each variable is called a *product term*. Similarly, a *sum term* is defined. A product term p is *independent of* a variable x_i, if p contains no literal of a variable x_i. A *minterm* is a product term which consists of literals of all Boolean variables.

When a sum of product terms represents a Boolean function, it is called a *sum-of-products representation* of the function. Similarly, a *product-of-sums representation* of a function is defined.

We say that a Boolean function f *implies* another Boolean function g, when every combination of values of the variables which satisfies $f = 1$ also satisfies $g = 1$. The implication relation is similarly defined for product terms.

An *implicant* of a Boolean function f is a product term which implies f. A minterm, which is an implicant of a Boolean function f, is simply called a *minterm of* f. A *prime implicant* of a Boolean function f is defined as an implicant of f that implies no other implicant of f. An implicant of a function f is called a *non-prime implicant* of f, when it is

not a prime implicant.

$f(x_i = 0)$ is an $(n-1)$-variable function $f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ obtained from an n-variable function f by fixing x_i to 0, where x_i is a variable in f. Similarly, $f(x_i = 1)$ is defined. A Boolean function f is *independent of* a variable x_i, if $f(x_i = 0) = f(x_i = 1)$ for all combinations of values of all variables except x_i.

We denote the logical product of $f(x_i = 0)$ and $f(x_i = 1)$ by $f(x_i = *)$.

2.2.2 Conventional Algorithms for Generating Prime Implicants

Various algorithms for generating all prime implicants of a Boolean function which are suited to computer processing have been proposed. They can be classified in three types: the Quine-McCluskey method, algorithms based on the expansion of a product-of-sums representation and ones based on the consensus expansion.

In the *Quine-McCluskey method*[Qui55, McC56], adjacent implicants are combined exhaustively in a systematic manner using tables of implicants.

Nelson showed that when a product-of-sums representation of a Boolean function is expanded to a sum-of-products representation by means of the distributive law $(A(B + C) = AB + AC)$ and some other primitive laws, all prime implicants of a given Boolean function are generated with, possibly, some non-prime implicants[Nel54]. Slagle et al. proposed a method for generating prime implicants from a product-of-sums representation by means of tree search[SCL70]. Kambayashi et al. proposed the *clause selection method* by combining Nelson's theory and Slagle's method, in which the searched tree is smaller than that of the Slagle's method[KOY79]. Recently, the author's colleague developed a vector algorithm called the *variable-oriented expansion method* based on the same

idea, and showed that the computation time on the vector supercomputer is improved[Kag87].

Tison showed that all prime implicants of a given Boolean function can be generated by consensus expansion[Tis67]. The consensus expansion is based on the following equation which holds for any Boolean function f :

$$f = \overline{x_i} f(x_i = 0) + x_i f(x_i = 1) + f(x_i = *)$$

where x_i is a Boolean variable in f. Using the equation repeatedly for every variable, f is expanded in a ternary tree fashion, and consequently, its all implicants are generated. In order to use the consensus expansion for generating all prime implicants, the removal of the generated non-prime implicants or the prevention of the generation of non-prime implicants is necessary. Morreale proposed an algorithm (we call it the *Morreale method*) in which the generation of non-prime implicants is prevented by means of tagging functions[Mor70]. Recently, the author's colleague developed the ternary tree expansion method (we call it the *consensus expansion method with pointers* in this thesis), in which pointers are introduced for the removal of non-prime implicants[Kag87].

The three new vector algorithms proposed in this chapter are based on the consensus expansion.

2.2.3 Vector Supercomputer

A vector supercomputer is a highly pipelined supercomputer which is primarily used for large scale scientific and engineering computation. It has, in addition to a conventional processing unit (a *scalar unit*), several function pipelines and vector registers (a *vector unit*). It yields more than GFLOPS (Giga FLoating Operations Per Second) of computation power by executing uniform operations on array structured data. In order to

support large-scale computation, it has a large main memory unit (usually a hundred mega bytes or more) and powerful load/store pipelines.

In addition, vector supercomputers have many advanced features in order to make it versatile enough to be used in a wide range of applications. For example, vector supercomputers such as the FACOM VP-400E at the Kyoto University and HITAC S-820/80 at the University of Tokyo provides the following vector operations.

(1) Element-wise Vector Operations
Vector supercomputers can handle integer and logical data as well as floating-point data by function pipelines. For example, integer arithmetic operations, bit-wise (32 bits per word) logical operations and logical shift operations can be vectorized.

(2) Conditional Vector Operations
The above operations can be masked by conditions, i. e., operations work only on elements which satisfy a specified condition. For example, the following program is vectorized by this function.

```
      DO 10 I=1,N
        IF (IM(I).EQ.0) IA(I)=IB(I)+IC(I)
   10 CONTINUE
```

(3) Constant Stridden Vector Access and List Vector Access
Vector supercomputers provide constant stridden vector access and indirect memory access (referred to as *list vector access*) as well as contiguous vector access. For example, the following program is vectorized by constant stridden vector access;

```
      DO 20 I=1,N,K
        IA(I)=IB(I)+IC(I)
   20 CONTINUE
```

and the following program is vectorized by list vector access;

```
      DO 30 I=1,N
        IA(I)=IB(IL(I))
   30 CONTINUE
```

(4) Compress operations
Compress operation, which constructs new vector IA from vector IB by collecting elements which satisfy a specified condition, can be vectorized. An example program for compress operation is as follows.

```
      K=0
      DO 40 I=1,N
        IF (IM(I).EQ.0) THEN
          K=K+1
          IA(K)=IB(I)
        ENDIF
   40 CONTINUE
```

Discussions in the following sections are common to all vector supercomputers which have above four features.

In order to utilize vector supercomputers efficiently, we must tune up the coding schemes and/or modify the basic algorithms so that our programs are suitable for vector processing. The features of the programs required for efficient vector processing are

(1) *high vectorization ratio*, i. e., almost all operations in the program should be processed by a vector unit.

(2) *long vector length*, i. e., sufficiently many elements should be processed simultaneously.

2.3 Consensus Expansion Method with Table Look-Up

2.3.1 Algorithm

In the consensus expansion method with table look-up, a systematic procedure to remove non-prime implicants is introduced. Assuming that all prime implicants of every m-variable Boolean function are known for a certain m (≥ 0), following algorithm generates all prime implicants of a given n-variable Boolean function f_{given} ($n > m$). F_i and P_i ($m \leq i \leq n$) are a set of Boolean functions and a set of product terms, respectively.

[Algorithm 1]
Input: $f_{given}(x_1, \ldots, x_n)$: an n-variable Boolean function
Output: P_n : the set of all prime implicants of f_{given}

1. $F_n = \{f_{given}\}$
   (a set with only one element function)

2. for $k = n$ downto $m + 1$ do
   $F_{k-1} = \{\overline{x_k} f(x_k = 0), x_k f(x_k = 1), f(x_k = *) | f \in F_k\}$

3. P_m={all prime implicants of $f | f \in F_m$ }
   (Note that $f \in F_m$ is the logical product of a product term, say p, with at most $(n-m)$ literals and an m-variable function, say $\hat{f}$. A set of all prime implicants of f is easily obtained as the logical product of p and every prime implicants of $\hat{f}$.)

4. for $k = m + 1$ to n do
   Let P_k be the set of all product terms of P_{k-1} each of which either

   (a) independent of x_k, or

(b) dependent on x_k and there is not such product term in P_{k-1} that is obtained by removing the literal of x_k from the representation of it.

In step 2, the consensus expansion on f_{given} is performed for the $(n-m)$ variables. In every consensus expansion, the number of elements of the set is increased by at most three times, and the maximum number of elements of F_m is 3^{n-m}. In step 3, the set, P_m, whose each element is a prime implicant of a function in F_m, is obtained. P_m includes all prime implicants of f_{given} as well as, possibly, the greater part of the non-prime implicants. In step 4, all non-prime implicants in P_m are removed systematically, and the set P_n, which includes only all prime implicants of f_{given}, is obtained.

We will show the correctness of Algorithm 1.

[Lemma 1]
Let f be a k-variable Boolean function, and x_i a Boolean variable ($1 \leq i \leq k$). A product term p is a prime implicant of f, if and only if one of the following statements is true.

1. p is a prime implicant of $f(x_i = *)$.
2. p is a prime implicant of $\overline{x_i} f(x_i = 0)$, and does not imply $f(x_i = *)$.
3. p is a prime implicant of $x_i f(x_i = 1)$, and does not imply $f(x_i = *)$

A proof of Lemma 1 will appear in appendix.

[Lemma 2]
Let f be a Boolean function, p a prime implicant of f, and g a Boolean function which implies f. p implies g if and only if p is an implicant of g. If p is an implicant of g, p is a prime implicant of g.

Lemma 2 is obvious.

[Theorem 1]
P_n obtained by Algorithm 1 is a set of all prime implicants of f_{given}.
*(proof)*Assuming that P_{k-1}={all prime implicants of $f|f \in F_{k-1}$ } ($m < k \leq n$), it follows from Lemma 1 and Lemma 2 that P_k={all prime implicants of $f|f \in F_k$ }. □

2.3.2 Data Structure

In step 2, each function f in F_k is represented as

$$f = p\hat{f}(x_k, \ldots, x_1)$$

where p is a product term $L_n \cdots L_{k+1}$ and $L_i \in \{\overline{x_i}, x_i, 1\}$ $(k+1 \leq i \leq n)$. In order to implement efficiently on vector supercomputers, we represent every $\hat{f}$ by a truth table, i. e., a 2^k-bit sequence. (We assume that f_{given} is also represented by a truth table.) The truth table for $\hat{f}$ can be represented using 2^{k-5} words, because a word consists of 32 $(= 2^5)$ bits. Each word stores a truth table for the 5 variables, $x_5, \ldots, x_1$, which is a part of the truth table for $\hat{f}$ corresponding to a certain combination of values of the $k-5$ variables, $x_k, \ldots, x_6$. The adopted data structure for $\hat{f}$'s of F_k is shown in Figure 2.1. The product term $p = L_n \cdots L_{k+1}$ for every $f \in F_k$ is represented by an integer according to the following formula and stored in a word

$$\sum_{i=k+1}^{n} r_i 3^{i-1}$$

where r_i takes 0 or 1 or 2 accordingly as L_i is $\overline{x_i}$ or x_i or 1. Since there are at most 3^{n-k} functions in F_k, $(2^{k-5}+1)3^{n-k}$ words are sufficient to represent F_k. In addition, it is possible to implement step 2 so as to reuse the space for F_k to the space for F_{k-1}. Hence $(2^{m-5}+1)3^{n-m}$ words are sufficient to implement step 2.

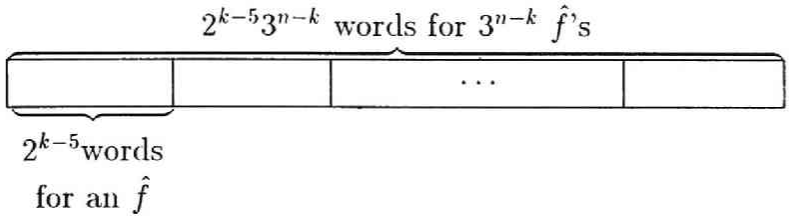


Figure 2.1: Data Structure of F_k

In step 3, a read-only table is introduced to give all prime implicants of arbitrary m-variable Boolean function. Table look-up is efficiently implemented on vector supercomputers using list vector access. Because the number of all m-variable Boolean function is 2^{2^m}, we let m be 4 considering the size of the table which have to be stored within main memory. It is true that a large space is required for this table, but the required space for step 2 is considerably saved choosing large m.

In step 3 and step 4, we use a 3^n-bit sequence to represent (candidates of) the prime implicants of an n-variable Boolean function. We call it a *map representation of prime implicants*. Every bit in the sequence corresponds to a product term of n variables, and is 1 when the corresponding product term is (a candidate of) a prime implicant of the function. To simplify the processing, we use 3^{n-3} words to represent each P_k where only $3^3 = 27$ bits are used in each word. The 27 product terms corresponding to a word are the same except the literals for x_3, x_2 and x_1. We arrange the words in the sequence so that each path in step 4 can be performed by a linear scan on P_k. The word corresponding to a product term $L_n \cdots L_4$ is in the $(\sum_{i=4}^{n} r_i 3^{i-4} + 1)$st location, where r_i takes 0 or 1 or 2 accordingly as L_i is $\overline{x_i}$ or x_i or 1. The data structure for P_k is shown in Figure 2.2. It is also possible to implement step 4 so as to reuse the space for P_{k-1} to the space for P, hence the required space for step 4 is

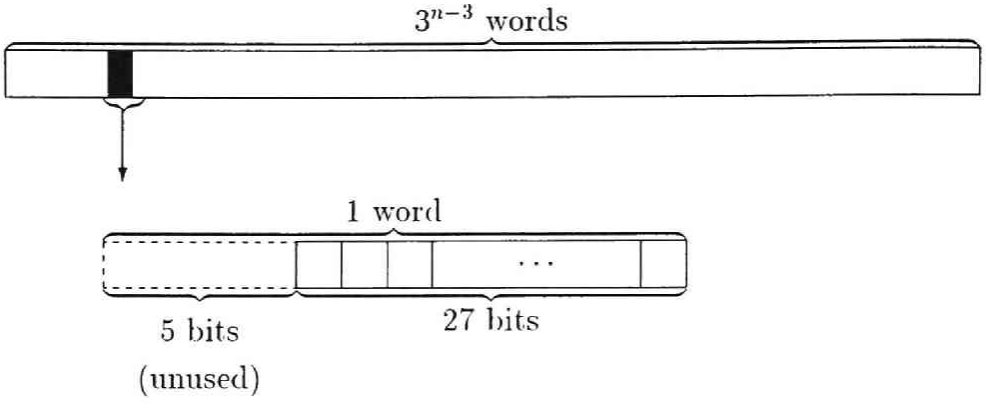


Figure 2.2: Data Structure of P_k

3^{n-3} words.

The constant stridden vector access is useful in step 2 and step 4, and the indirectly addressed access is useful in step 3. The function pipelines which support bit-wise logical operations on a word are useful through the whole processing. Therefore, Algorithm 1 is expected to be highly vectorized.

The required memory space is proportional to 3^n. It is reasonable because there are n-variable Boolean functions which have $O(3^n/n)$ prime implicants[DF59]. For example, generation of all prime implicants of 18-variable Boolean function is performed within 100 megabytes.

In order to estimate the computation time, let us consider the references of memory. In step 2, the number of references of memory is proportional to the size of F_m. In step 3, table references are at most $3n - m$ times. In step 4, 3^n-bit space is scanned $(n - m)$ times. Therefore, step 4 is the most time consuming, and the computation time of Algorithm 1 is $O((n - m)3^n)$. Compared to the case that the table is not used (i. e., choosing m=0), the table which gives all prime implicants of

all m-variable Boolean function enables us $n/(n-m)$ times speed up.

2.4 Morreale Method with Table Look-Up

2.4.1 Algorithm

In the Morreale method with table look-up, the Morreale method[Mor70] is modified so that the table look-up technique can be introduced. Following algorithm generates all prime implicants of a given Boolean function, assuming that, for a certain m (≥ 0), all prime implicants of every m-variable Boolean function are known. $[f, g_1, \ldots, g_j]$ is a tuple of Boolean functions f, g_1, ..., g_j. T_k ($m \leq k \leq n$) and P are a set of tuples and a set of product terms, respectively.

[Algorithm 2]
Input: $f_{given}(x_1, \ldots, x_n)$: an n-variable Boolean function
Output: P_n : the set of all prime implicants of f_{given}

1. $T_n = \{[f_{given}]\}$
   (a set with only one element which is a tuple of only one Boolean function)

2. for $k = n$ downto $m+1$ do
   $T_{k-1} = \{[f(x_k = *), g_1(x_k = *), \ldots, g_j(x_k = *)],$
   $[\overline{x_k} f(x_k = 0), \overline{x_k} g_1(x_k = 0), \ldots, \overline{x_k} g_j(x_k = 0), f(x_k = *)],$
   $[x_k f(x_k = 1), x_k g_1(x_k = 1), \ldots, x_k g_j(x_k = 1), f(x_k = *)]$
   $\mid [f, g_1, \ldots, g_j] \in T_k\}$

3. $P = \{$all prime implicants of f which are prime implicants of
   neither g_1, ..., nor $g_j \mid [f, g_1, \ldots, g_j] \in T_m]\}$
   (Note that f in $[f, g_1, \ldots, g_j] \in T_m$ is the logical product of a product term, say p, with at most $(n-m)$ literals and an m-variable function.

say $\hat{f}$. A set of all prime implicants of f is easily obtained as the logical product of p and every prime implicants of $\hat{f}$. In the same way, a set of all prime implicants of g_i's $(1 \leq i \leq j)$is easily obtained.)

The generation of non-prime implicants is prevented by using the *tagging functions* g_i's. In step 2, f_{given} is decomposed to m-variable functions by the consensus expansion and, at the same time, the tagging functions are generated. Consequently, the set T_m, of tuples of functions, is obtained. In step 3, the set, P, is obtained. Each element of P is a prime implicant of the function f but is a prime implicant of neither of the functions g_1, ... , nor g_j where $[f, g_1, ..., g_j]$ is in F_m. P includes all prime implicants of f_{given} but nothing else. The process for removing non-prime implicants is not necessary.

Algorithm 2 includes the Morreale method by choosing $m = 0$.

We will show the correctness of Algorithm 2.

[Lemma 3]

Let f be a k-variable Boolean function and g_1, ...and g_j $(j \geq 0)$ be k-variable Boolean functions which implies f, and x_i be a Boolean variable $(1 \leq i \leq k)$. p is a prime implicant of f which implies neither g_1, ...nor g_j, if and only if

1. p is a prime implicant of $f(x_i = *)$ which implies neither $g_1(x_i = *)$, ..., nor $g_j(x_i = *)$, or
2. p is a prime implicant of $\overline{x_i} f(x_i = 0)$ which implies neither $\overline{x_i} g_1(x_i = 0)$, ..., $\overline{x_i} g_j(x_i = 0)$ nor $f(x_i = *)$, or
3. p is a prime implicant of $x_i f(x_i = 1)$ which implies neither $x_i g_1(x_i = 1)$, ..., $x_i g_j(x_i = 1)$ nor $f(x_i = *)$.

A proof of Lemma 3 will appear in appendix. Note that Lemma 3 implies Lemma 1 by choosing $j = 0$.

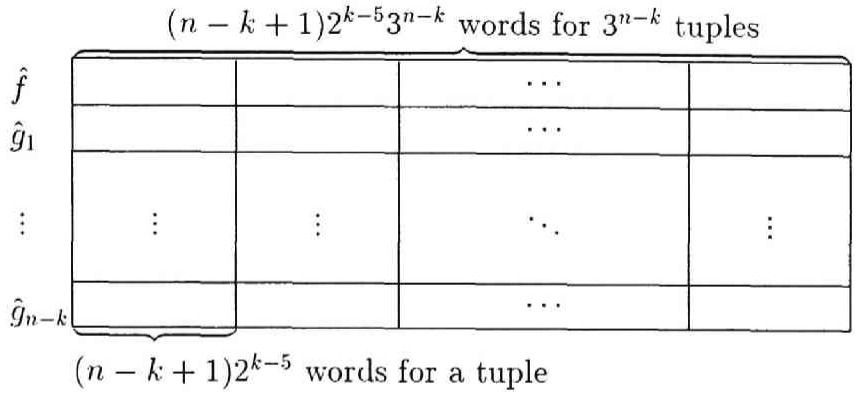


Figure 2.3: Data Structure of T_k

[Theorem 2]
P obtained by Algorithm 2 is a set of all prime implicants of f_{given}.

*(proof)*Apply Lemma 2 and Lemma 3 to every expansion of step 2 in Algorithm 2. □

2.4.2 Data Structure

Data structure for Algorithm 2 is basically the same as that for Algorithm 1. The data structure for T_k can be implemented as the $(n-k+1)$-ple of the data structure for F_k mentioned in previous section. We regard $[f(x_k = *), g_1(x_k = *), \ldots, g_j(x_k = *)]$ as $[f(x_k = *), g_1(x_k = *), \ldots, g_j(x_k = *), 0]$ in order to introduce uniform data structure. Therefore, the required space for T_k is $((n-k+1)2k-5+1)3^{n-k}$ words. The adopted data structure for T_k is shown in Figure 2.3. In addition, it is possible to implement step 2 so as to reuse the space for T_k to the space for T_{k-1}. Hence $((n-m+1)2^{m-5}+1)3^{n-m}$ words are sufficient to implement step 2.

P is obtained by calculating the logical product of the table content

referred to by $\hat{f}$ and the logical negations of those referred to by $\hat{g_i}$'s $(1 \leq i \leq n-4)$ for every tuple $[f, g_1, \ldots, g_{n-4}]$ in F_4. The data structuire adopted for P and the table is the same as in Algorithm 1. Therefore the size of P is 3^{n-3} words.

Various functions of a vector supercomputer can be effectively used, and Algorithm 2 is expected to be highly vectorized.

The required memory space to generate all prime implicants of arbitrary 18-variable Boolean functions is about 169 Mbytes.

In order to estimate the computation time, let us consider the references of memory. In step 2, the number of references of memory is proportional to the size of T_m. In step 3, table references are at most $(n-m+1)3n-m$ times. Therefore, the computation time of Algorithm 2 is $O((n-m)3^n)$. Compared to the case that the table is not used (i. e., choosing m=0), the table which gives all prime implicants of all m-variable Boolean function enables us $n/(n-m)$ times speed up.

2.5 Extended Consensus Expansion Method with Table Look-Up

Instead of using look-up table to obtain all prime implicants of an m-variable Boolean function in step 3 of Algorithm 1 or 2, it is possible to use a program which generates all prime implicants of an m-variable Boolean function. Let us consider the use of Algorithm 1 for the 'look-up table' for Algorithm 2. In the following, we denote m's in Algorithm 1 and 2 by m_1 and m_2, respectively, to distinguish them.

The consensus expansion method with table look-up is a high-speed algorithm suitable for vector supercomputer. However, its ability is limited by required memory space. During step 4, whole set of (candidates of) prime implicants should be held on the main memory. On the other hand,

the removal of non-prime implicants is not necessary in Algorithm 2, and every tuple in T_{m_2} can be processed sequentially. That is, Algorithm 2 can be implemented if only there is the space for T_{m_2} and the space to obtain (applying Algorithm 1) all prime implicants of m_2-variable Boolean functions in just one tuple in T_{m_2}. This approach enables us to enjoy the high-speed of Algorithm 1 for generating all prime implicants of a Boolean function with larger number of variables within limited memory space.

It is clear from Lemma 2 that the set of prime implicants of tagging functions used in step 3 may include non-prime implicants. In other word, step 4 of Algorithm 1 can be omitted for generating a set of prime implicants of tagging functions of Algorithm 2.

The required memory space for this method, called the extended consensus expansion method with table look-up, is as follows;

- For T_{m_2}, $((n - m_2 + 1)2^{m_2-5} + 1)3^{n-m_2}$ words.
- For Algorithm 1 called by Algorithm 2, $O(3^{m_2})$.

Therefore, when n is not much larger than m_2, the required memory space for this method is $O(3^{m_2})$. In other words, this method enables us to generate all prime implicants of an n-variable Boolean function within only the memory space required for generating all prime implicants of an m_2-variable Boolean function by Algorithm 1, where n is just a little larger than m_2.

In order to estimate the computation time, let us consider the references of memory. In step 2, the number of references of memory is proportional to the size of T_{m_2}. In step 3, Algorithm 1 is called to obtain prime implicants of 3^{n-m_2} $\hat{f}$'s and $(n - m_2)3^{n-m_2}$ tagging functions. The number of memory references of Algorithm 1 for an m_2-variable Boolean function is $O((m_2 - m_1)3^{m_2})$. For tagging functions, step 4 may be omit-

ted, and the number of memory references of Algorithm 1 is reduced to $O(3^{m_2-m_1})$. Therefore, the number of memory references in step 3 of Algorithm 2 is $O((m_2 - m_1)3^n + (n - m_2)3^{n-m_1})$.

2.6 Implementation and Evaluation

2.6.1 Implementation

The methods proposed in sections 2.3, 2.4 and 2.5 are coded in Fortran77 and implemented on FACOM VP-400E at the Kyoto University Data Processing Center. We call the program *CE/T*, *M/T* and *ECE/T*, respectively. Following techniques are adopted for implementation of CE/T.

1. In step 2, *inconsistency* in F_k, i. e., the f whose corresponding $\hat{f}$ is 0 for every combination of values of the variables, can be eliminated, because they have no implicants. Since it is not so efficient to check such f's in F_k's for $k > 5$, we check f's in only F_5. The check can be performed by examining whether the word expressing the $\hat{f}$ is zero. The vector compress function is effectively used for the elimination.

2. The last expansion in step 2, i. e., the expansion for x_5, and the table look-up for obtaining all prime implicants of 4-variable Boolean functions in step 3 are combined. The contents of the table indexed by the lower and the upper half-word of the corresponding $\hat{f}$ are referred to for $f(x_5 = 0)$ and $f(x_5 = 1)$ respectively, where $f \in F_5$. For $f(x_5 = *)$, the content indexed by the logical product of the two half-words is referred to. Each referenced content is stored at the correct location in P_m using the integer representation of p as an index.

3. In order to reduce the bank confliction of memory references, four copies of the table are prepared and used one after another. About

800 kilowords are required for the four tables.

4. Two adjacent iteration steps in step 4 are paired into one iteration step so as to accelerate the computation speed (loop unrolling).

All of the above techniques are also adopted for ECE/T, and (1) and (2) of the above techniques are adopted for M/T. For CE/T and M/T, 4 is chosen as m, and for ECE/T, 4 and 18 are chosen as m_1 and m_2, respectively.

2.6.2 Evaluation

Figure 2.4, 2.5 and 2.6 show the benchmark result of the programs CE/T, M/T and ECE/T, respectively, on the FACOM VP-400E. Experiments for CE/T and M/T are performed for 12- to 18-variable Boolean functions, and experiments for ECE/T are performed for 19- and 20-variable Boolean functions. Every cross designates average computation time for 10 Boolean functions of the same number of variables and truth table density. (Truth table density is the ratio of 1's in the truth table.) The truth tables of the functions are generated randomly based on Lehmer's linear congruence method using RANU2 in the scientific subroutine library SSL-II[Fuj80]. (The Boolean functions for the truth table density 0/16 and 16/16 are unique, which are called *inconsistency* and *tautology*, respectively.)

The computation time by CE/T is not much affected by the truth table density. The average computation time is about 2.3 msec for 12-variable functions, and about 1.4 sec for 18-variable functions. The computation time by M/T deeply depends on the truth table density, due to the number of table references in step 3 which are affected by the elimination of the tuples with *inconsistency*. The average computation time is about 9.7 msec for 12-variable functions, and about 5.3 sec for 18-variable func-

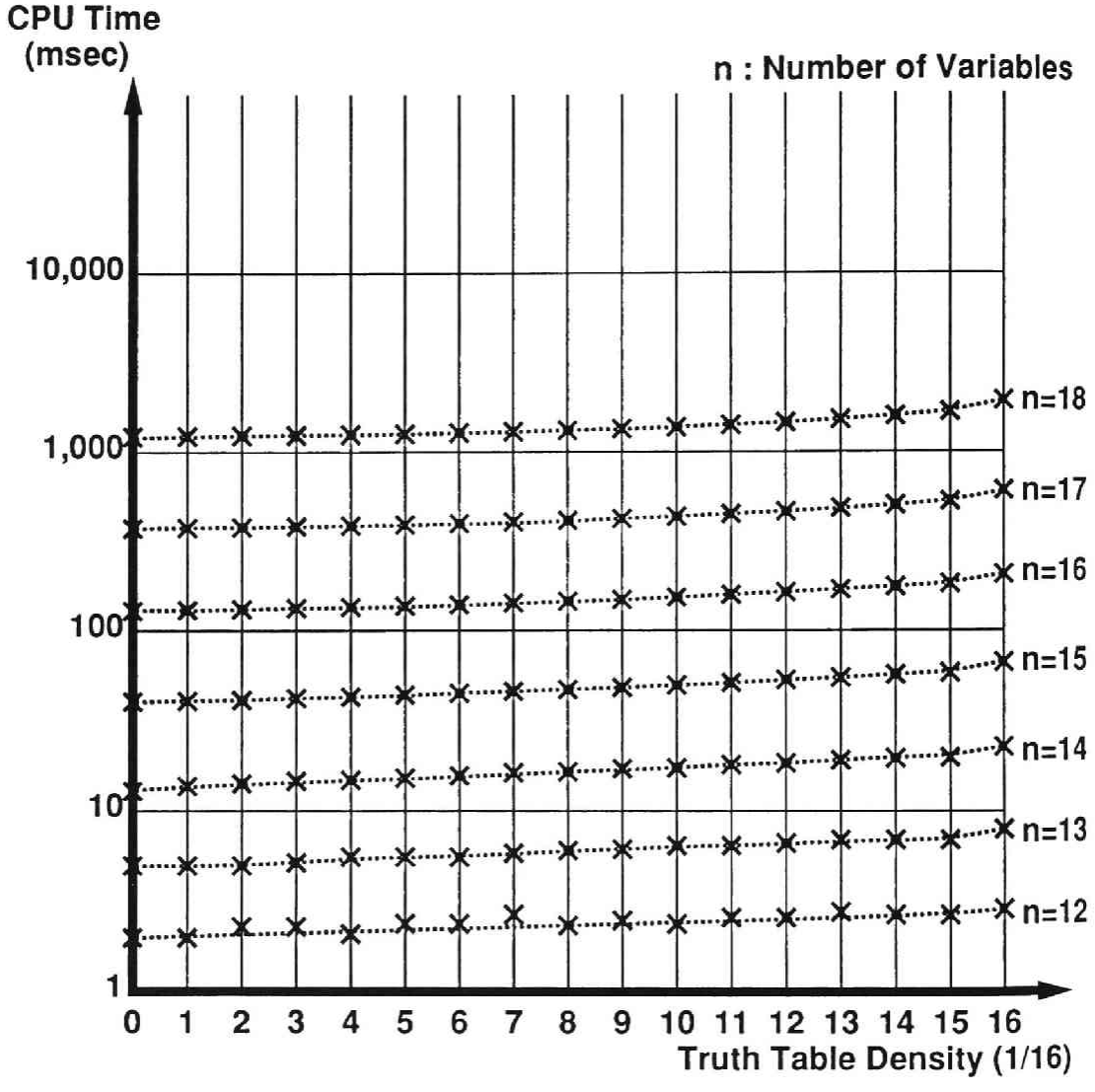


Figure 2.4: Average Computation Time of the Consensus Expansion Method with Table Look-up

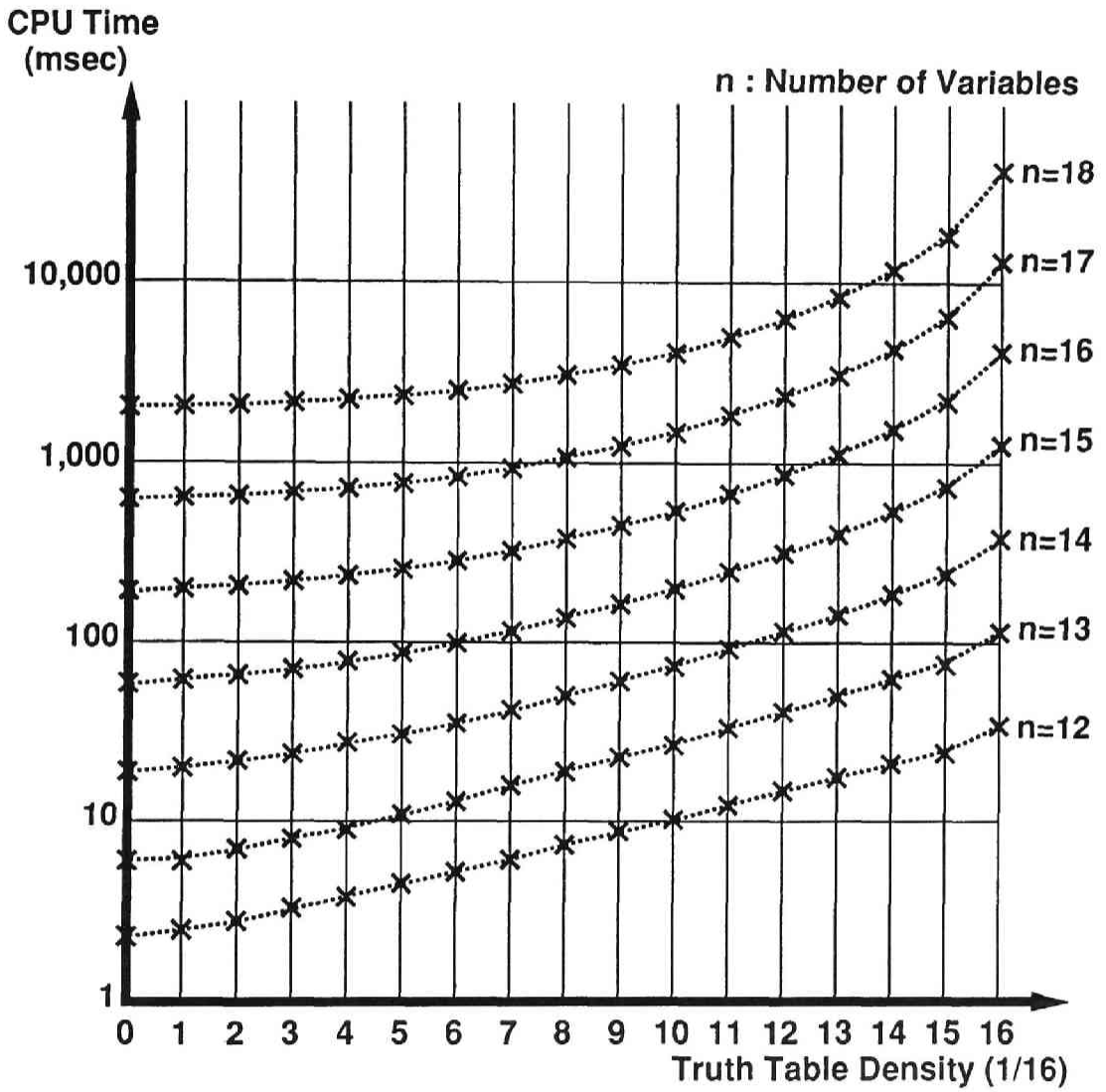


Figure 2.5: Average Computation Time of the Morreale Method with Table Look-up

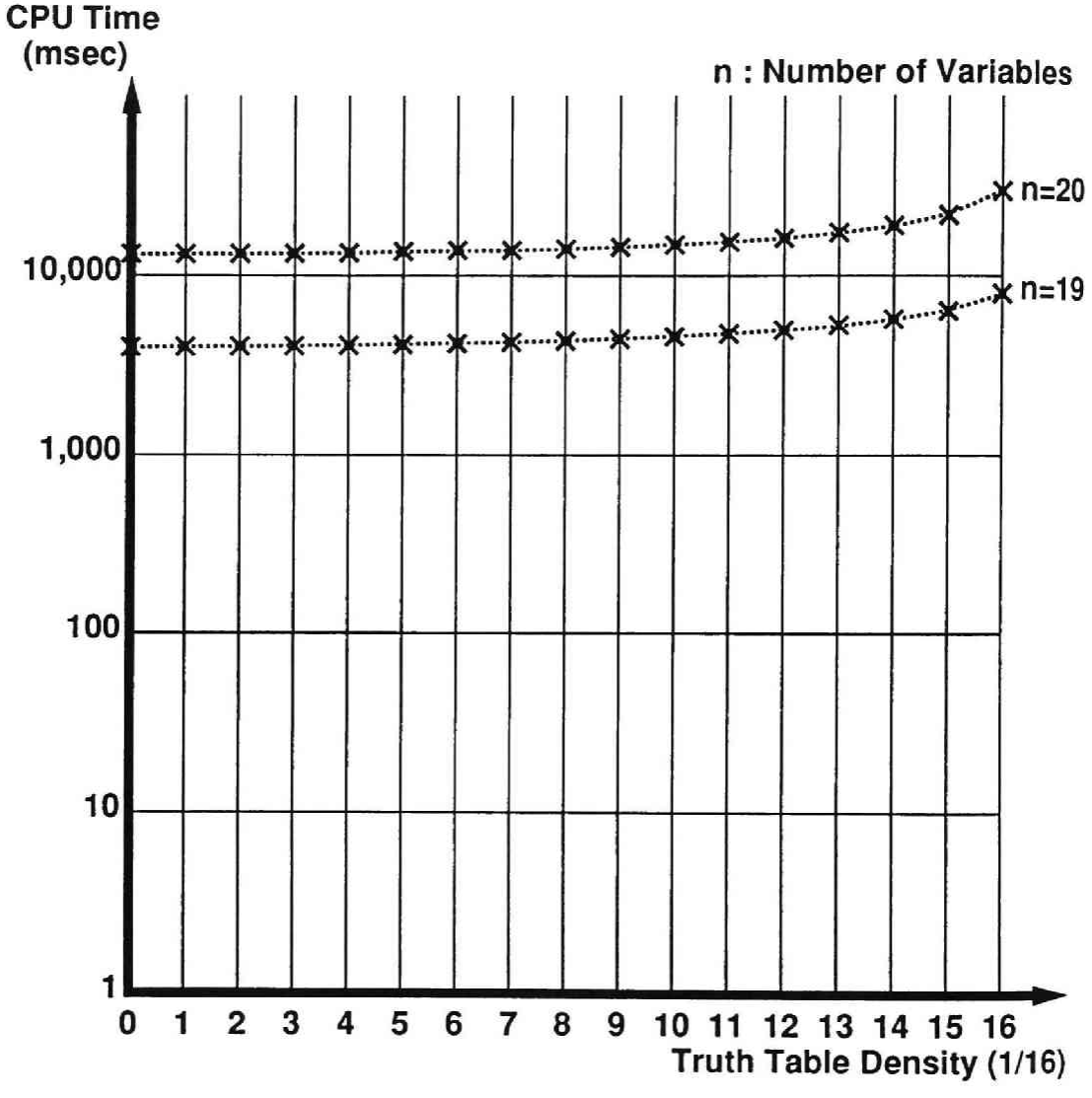


Figure 2.6: Average Computation Time of the Extended Consensus Expansion Method with Table Look-up

tions. The average computation time by ECE/T is about 15.8 sec for 20-variable functions.

For comparison, the Quine-McCluskey method and the Morreale method are also implemented on the VP-400E. We call the programs *QM* and *M*, respectively. Figure 2.7 shows the comparison of the computation time. Each dot indicates average computation time for 150 Boolean functions of the same number of variables (10 functions for each of 15 kinds of truth table density). The average computation time for 12-variable functions by a program based on the clause selection method (*CS*), that by one based on the variable-oriented expansion method (*VOE*) and that by one based on the consensus expansion method with pointers (*CE/P*) are also designated in the figure, which were evaluated on FACOM VP-200 by Kagatani et al.[Kag87].

Table 2.1 shows comparison of the vector acceleration ratio, i. e., (CPU time using scalar unit only) / (CPU time enabling vector unit). As the table indicates, the acceleration ratio of CE/T is very high. In this table, computation time of *V-versions*, i. e., programs coded to be suited for vector execution, are compared. However, it is confirmed by other experiments that *S-versions*, programs suitable for scalar execution, are at most 20% faster than the corresponding V-versions in scalar execution for these algorithms except the M (the Morreale method) whose S-version is about 2.7 times faster than the V-version.

Table 2.2 shows the required memory size for CE/T, M/T and ECE/T. This table represents the total size of declared fortran array size, and do not include the space for scalar variables or machine codes.

The developed programs are portable, and easily implemented on another vector supercomputer HITAC S-820/80 at the University of Tokyo without loss of efficiency. By the similar experiments to the VP-400E, the average computation time on S-820/80 for 18-variable functions by

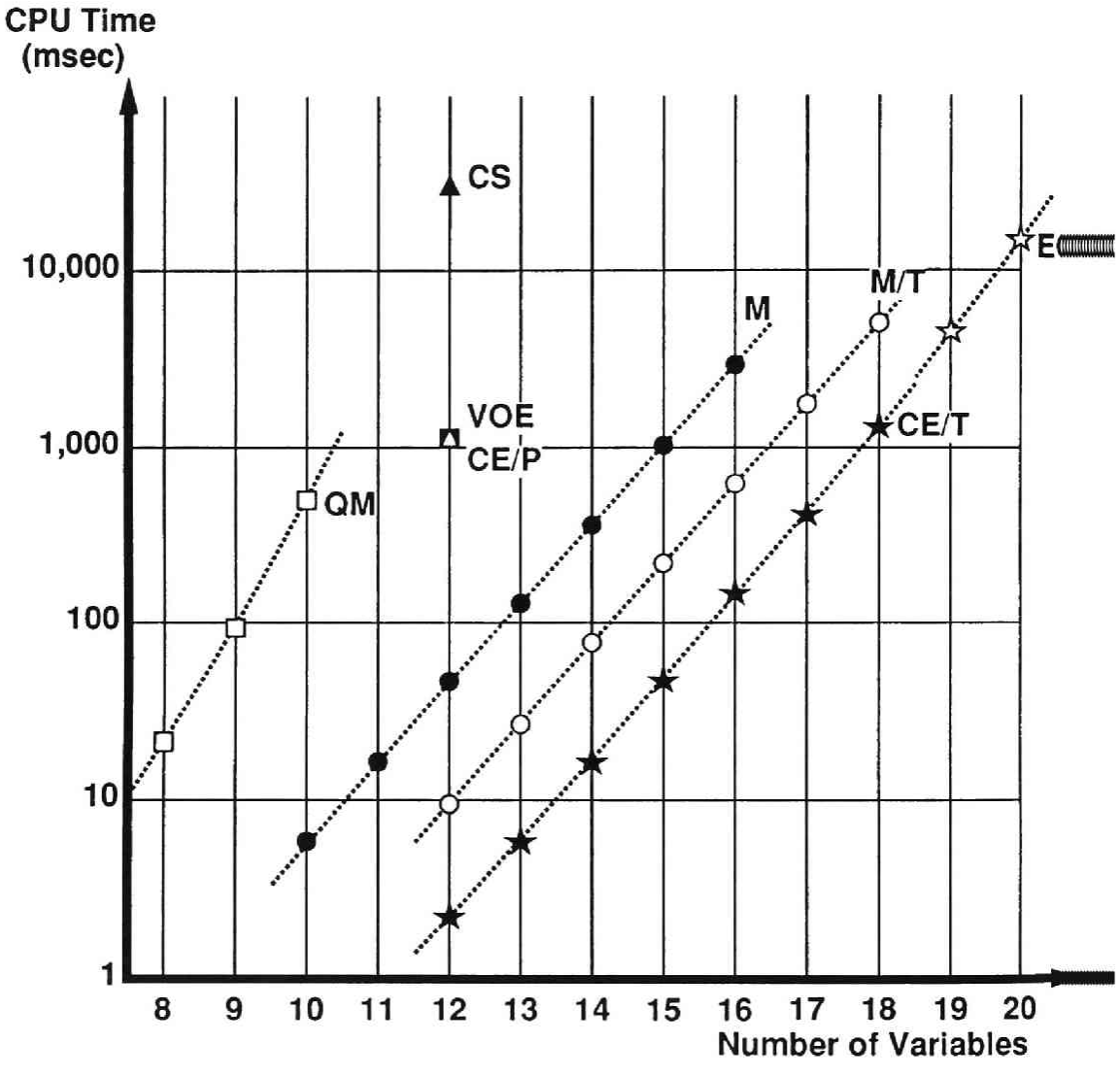


Figure 2.7: Comparison of Computation Time

Table 2.1: Comparison of Vector Acceleration Ratio

Program	n	Processor	CPU Time (msec)		Vector Acceleration Ratio (S/V)
			Scalar (S)	Vector (V)	
CE/T	12	VP-400E	33.3	2.31	14.42
M/T	12	VP-400E	65.9	9.69	6.81
M	12	VP-400E	571.4	47.5	12.03
QM	10	VP-400E	5,187	523	9.92
VOE	12	VP-200	10,851	1,186	9.15
CE/P	12	VP-200	10,579	1,183	8.94
CS	12	VP-200	46,750	32,009	1.46

Table 2.2: Required Memory Size (kilobytes)

n	CE/T	M/T	ECE/T
12	3,242	948	—
13	3,434	1,332	—
14	4,011	2,537	—
15	5,744	6,304	—
16	10,940	18,068	—
17	26,529	54,743	—
18	73,296	168,920	—
19	—	—	73,296
20	—	—	73,296

CE/T is about 0.4 sec. This program is also efficient on conventional scalar processors; for example, the average computation time on the processors such as FACOM M-360R or ACOS-850/8 for 12-variable functions by CE/T is about 0.2 sec.

2.6.3 Discussions

As Figure 2.7 indicates, programs based on the proposed methods are faster than the programs based on the conventional methods. Especially, CE/T is the fastest, and is about 20 times as fast as M which is the fastest among the programs based on previous algorithms. As Table 2.1 indicates, the programs based on the proposed methods achieves high vector acceleration ratio. As estimated in the previous sections, Figure 2.7 indicates that the computation time of CE/T, M/T and ECE/T increases approximately 3 times per variable. As Table 2.2 shows, ECE/T realizes the speediness of CE/T within the limited memory space.

In QM, each operation is simple and hence the acceleration ratio is high. However, the computation time is large due to so many operations. CS and VOE, as well as CE/P, are not so efficient because of the complex data structure. It is difficult to find effective data structure for representing the logical sum of product-of-sums' or the logical product of sum-of-products' which appears during the expansion. Thus, the algorithms based on the consensus expansion are the most suited to vector processing among the three types of algorithms. The truth table representation of a function and the bit-sequence representation of (candidates of) prime implicants are effective.

The table look-up technique is also effective to reduce the computation time and the required memory space. The difference in performance between the M/T and M is due to the data structure and the use of the table look-up technique. (The table of about 800 Kilobytes saves hun-

dreds of Megabytes of memory space required for the expansion process for the last 4 variables.)

M/T, as well as M, can be used for the generation of all prime implicants of an *incompletely specified* Boolean function, i. e., a function with *don't cares*, with a slight modification[Mor70]. We can also generate all prime implicants of such a function using CE/T twice.

Using the developed program ECE/T with $m_2 = 18$, prime implicants of a Boolean function with even 22 or more variables can be generated within the main memory of the VP-400E at the Kyoto University. For furthermore variables, the required space for the step 2 become dominant. In such case, we can execute sequentially not only step 3 but also step 2 to reduce the required space for step 2.

We implemented the step 3 of Algorithm 1 by table look-up. It is possible by adding special vector instruction (e. g. a vector instruction which computes all prime implicants of given 5-variable Boolean functions) to make Algorithm 1 faster.

2.7 Application for the Study on the Number of Prime Implicants

Related to the two-level logic minimization, various studies on the number of prime implicants of Boolean functions have been made. The best known lower bound on the maximum number of prime implicants of n-variable Boolean functions is $O(3^n/n)$ presented by Igarashi[Iga79]. Igarashi conjectured that his lower bound is optimal. The best known upper bound on the maximum number of prime implicants of n-variable Boolean functions is $O(3^n/\sqrt{n})$ presented by Chandra et al.[CM78]. The average number of prime implicants of n-variable Boolean functions is studied by Cobham et al. and Mileto et al.[CFN62, MP64].

In this section, we will make an experimental study on the number of prime implicants of Boolean functions by means of the developed program. First, we will present the maximum number of prime implicants of 5-variable Boolean functions obtained by examining the number of prime implicants of every 5-variable Boolean functions exhaustively. Next, we will show the maximum number of prime implicants of 6-variable Boolean functions using the result obtained by the above experiment. The obtained value, 32 and 92, of the maximum number of prime implicants of respectively 5- and 6-variable Boolean functions are equal to the Igarashi's lower bound. We also present some other results related to the number of prime implicants.

2.7.1 Maximum Number of the Prime Implicants of 5-Variable Boolean Functions

It has been not clear whether Igarashi's lower bound[Iga79] is tight or not, even for $n = 5$. To solve this open problem, we first examined the number of all 5-variable Boolean functions. Using high-speed program for generating all prime implicants based on the consensus expansion method with table look-up, the experiment is performed on a computer ACOS-850/8 at the Integrated Media Environment Experimental Laboratory of Kyoto University.

Table 2.3 shows the result of this experiment. This table represents the number of Boolean functions with m minterms and p prime implicants. From the observation of this table, there are 16 5-variable Boolean functions with 32 prime implicants, and there is no 5-variable Boolean functions with more than 32 prime implicants. This value, 32, is equal to the Igarashi's lower bound for $n = 5$. The 16 5-variable Boolean functions with 32 prime implicants are equivalent up to the negation of the variables. This table also indicates the average number of prime implicants

for every number of minterms.

2.7.2 Maximum Number of the Prime Implicants of 6-Variable Logic Functions

Since there are 2^{64} 6-variable Boolean functions, it is unfeasible to examine the number of prime implicants of all 6-variable Boolean functions, as could be performed for 5-variable Boolean functions. We will determine the maximum number of prime implicants of 6-variable Boolean functions by applying Lemma 1 to the results of the experiments on 5-variable functions.

Now it has been determined that the maximum number of prime implicants of 5-variable Boolean functions is 32, following corollary holds.

[Corollary 1]
If there exists a 6-variable Boolean function f which has N prime implicants, the numbers of prime implicants of both $f(x_6 = 0)$ and $f(x_6 = 1)$ are more than or equal to $(N - 64)$.

*(proof)*The numbers of prime implicants of $f(x_6 = *)$, $\overline{x_6}f(x_6 = 0)$, and $x_6 f(x_6 = 1)$ are at most 32. From Lemma 1, the sum of the numbers of prime implicants of above three 5-variable Boolean functions should be more than or equal to N. □

From Corollary 1, we can conclude that, in order to find out all 6-variable Boolean functions with N or more prime implicants, it is sufficient to examine all 6-variable Boolean functions which can be synthesized as $\overline{x_6}g + x_6 h$, where g and h are 5-variable Boolean functions with $(N-64)$ or more prime implicants.

Since Igarashi showed that there are 6-variable Boolean functions with 92 prime implicants[Iga79], we made an experiment for $N = 92$ as follows:

1. Generate a set of all 5-variable Boolean functions with 28 $(= 92-64)$

Table 2.3: Correlation of the Number of Prime Implicants and the Number of Minterms of 5-Variable Boolean Functions

number of minterms / number of prime implicants	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0
1	32	80	0	80	0	0	0	40
2	0	416	2080	1720	2240	2720	480	640
3	0	0	2880	22400	34400	37680	58240	32480
4	0	0	0	11760	134880	289440	340960	553620
5	0	0	0	0	29856	527232	1400640	1793440
6	0	0	0	0	0	49120	1507392	4539520
7	0	0	0	0	0	0	58144	3512960
8	0	0	0	0	0	0	0	85600
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
average number of prime implicants	1.000	1.839	2.581	3.275	3.955	4.643	5.346	6.065

number of minterms / number of prime implicants	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	10
2	960	1440	320	480	0	240	0	20
3	21600	16800	28800	20960	10560	10080	4320	2720
4	453760	317840	191040	265160	269120	136640	136160	77940
5	3019840	3028992	2383904	1334720	1424160	1654800	908864	832480
6	6164960	10697440	12196064	10623384	6033760	4999920	6028800	3394544
7	10866240	15328320	26670400	33307072	31141920	18292880	12545920	14710240
8	7223200	20436480	30006080	49797040	66370400	65059200	38383520	23202650
9	297280	13476640	31314240	48911680	74061920	101446880	103669280	59398720
10	960	1190848	22335296	40663776	67244800	93328640	123617600	130077424
11	0	17280	3750816	31227584	47555680	77110400	102839680	126269920
12	0	160	144800	8874584	35339360	51810240	74909920	97298000
13	0	0	2720	738400	15319840	33071520	49648960	64488960
14	0	0	0	28000	2426400	18713040	28388640	39352000
15	0	0	0	0	172320	5121120	16216384	22747296
16	0	0	0	0	3360	646240	6804800	11295914
17	0	0	0	0	0	33120	1464640	5620160
18	0	0	0	0	0	640	146240	1936000
19	0	0	0	0	0	0	8960	330400
20	0	0	0	0	0	0	32	42560
21	0	0	0	0	0	0	0	2432
22	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
average number of prime implicants	6.792	7.515	8.222	8.902	9.544	10.146	10.709	11.237

Table 2.3: Correlation of the Number of Prime Implicants and the Number of Minterms of 5-Variable Boolean Functions (Continued)

number of minterms / number of prime implicants	17	18	19	20	21	22	23	24
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	160	320	0	240	0	0	0	40
3	1920	1040	3520	1840	2880	1920	480	120
4	35520	35120	6400	16960	9600	14240	13120	6620
5	561920	261120	226720	72680	52384	22880	31680	31280
6	2989920	2113920	1054400	777280	336640	148064	28960	43280
7	7877120	6626720	4721440	2388400	1613760	727760	325760	24400
8	25892320	12984080	9911040	6790440	3280960	2174560	903520	407200
9	32469440	33615280	15822720	11183680	6794720	2915200	1811040	725700
10	71507744	36402080	33774240	14097936	9784800	5006720	2013440	979000
11	129824640	67315520	33058720	26944320	9796224	6501280	2670080	1122240
12	113637440	106442960	49773440	24982720	16346560	5903040	3168800	997760
13	77722400	88843040	72811040	28932080	16153120	6908560	3046880	1150720
14	48699040	54775040	60053600	40730720	13788000	8696160	2328960	1106720
15	27329920	31244480	34250080	35241504	17457120	5887584	2923200	872640
16	14643552	16297680	17384160	19149760	17574784	5746304	2732160	607100
17	7411264	8068800	8308960	8848480	9903360	6461280	1564800	753280
18	3175040	3800800	3683680	3491840	4214240	4617520	1616000	596520
19	1465440	1646720	1520960	1364640	1281280	2018720	1515200	311680
20	382880	629440	658080	493120	431264	568752	917760	306400
21	81280	235360	231840	178080	143200	136320	317600	238880
22	12800	71600	70720	79040	37920	41760	95040	154880
23	960	19520	30400	15680	15040	9440	17280	53600
24	0	4480	10080	6520	4000	3200	6240	20640
25	0	480	6080	1184	2272	320	0	7200
26	0	0	960	2560	0	640	800	0
27	0	0	320	480	0	0	0	320
28	0	0	0	640	0	0	0	80
29	0	0	0	0	320	0	0	0
30	0	0	0	16	0	0	0	0
31	0	0	0	0	32	0	0	0
32	0	0	0	0	0	16	0	0
average number of prime implicants	11.736	12.211	12.659	13.067	13.411	13.653	13.750	13.662

number of minterms / number of prime implicants	25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0
3	320	480	0	80	0	0	0	0
4	0	1120	1920	240	320	80	0	0
5	24160	1280	320	2360	0	160	32	0
6	32320	31840	2880	80	1280	0	0	0
7	32000	20640	21760	160	0	0	0	0
8	37280	11280	7520	8880	0	160	0	0
9	289280	26800	4800	2880	480	0	0	0
10	388320	135776	4480	0	1600	0	0	0
11	383840	128160	38432	880	0	0	0	0
12	393440	128800	36800	3400	0	0	0	0
13	355200	76960	21440	7520	0	80	0	0
14	291040	97920	17120	3840	320	0	0	0
15	257600	55200	13184	3200	0	0	0	0
16	228320	55840	12960	0	960	0	0	0
17	180640	33920	1280	0	0	0	0	0
18	120000	24640	4800	1360	0	0	0	0
19	107520	20480	4800	960	0	0	0	0
20	90784	23520	3840	120	0	16	0	0
21	57280	10560	640	0	0	0	0	0
22	35680	7840	960	0	0	0	0	0
23	29280	6080	160	0	0	0	0	0
24	17280	2320	960	0	0	0	0	0
25	8640	2304	320	0	0	0	0	0
26	4000	1600	0	0	0	0	0	0
27	1280	480	0	0	0	0	0	0
28	320	0	0	0	0	0	0	0
29	0	320	0	0	0	0	0	0
30	32	0	0	0	0	0	0	0
31	0	32	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
average number of prime implicants	13.372	12.886	12.218	11.309	9.903	7.581	5.000	1.000

or more prime implicants, say F_5 (Cartesian of F_5 is 1808).

2. Examine the number of prime implicants of all 6-variable Boolean functions synthesized by two 5-variable Boolean functions out of F_5. We used the same program and computer as used for 5-variable functions to examine the number of prime implicants of every Boolean function.

From the experiment, there are 32 6-variable Boolean functions with 92 prime implicants, and there is no 6-variable Boolean functions with more than 92 prime implicants. This value, 92, is also equal to the Igarashi's lower bound for $n = 6$. The 32 6-variable Boolean functions with 92 prime implicants are equivalent up to the negation of the variables.

2.7.3 The Number of Prime Implicants of Boolean Functions of 7 or More Variables

From Lemma 1, we can observe that the following corollary holds.

[Corollary 2]
The maximum number of prime implicants of n-variable Boolean functions do not exceed three times of the maximum number of prime implicants of $(n-1)$-variable Boolean functions.

From Corollary 2 and the fact that the maximum number of prime implicants of 6-variable Boolean functions is 92, we can obtain the trivial upper bound $92 \cdot 3^{n-6}$ on the maximum number of n-variable Boolean functions, where $n > 6$. Table 2.4 shows this upper bound compared with the Igarashi's lower bound and the upper bound of Chandra et al. Our upper bound is better than that of Chandra et al. up to 43.

At last, Figure 2.8 shows the average number of prime implicants of 8- to 20-variable Boolean functions obtained experimentally. This result

Table 2.4: Upper and Lower Bounds of the maximum Number of Prime Implicants of n-variable Boolean Functions

n	Lower Bound by Igarashi	Upper Bound by Chandra et al.	Upper Bound of This Thesis
4	13	32	13
5	32	80	32
6	92	240	92
7	218	672	276
8	576	1,792	828
9	1,698	5,376	2,484
10	4,300	15,360	7,452
20	1.33×10^8	6.35×10^8	4.40×10^8
30	5.55×10^{12}	3.15×10^{13}	2.60×10^{13}
40		1.62×10^{18}	1.53×10^{18}
50		8.46×10^{22}	9.06×10^{22}
	$O(3^n/n)$	$O(3^n/\sqrt{n})$	$O(3^n)$

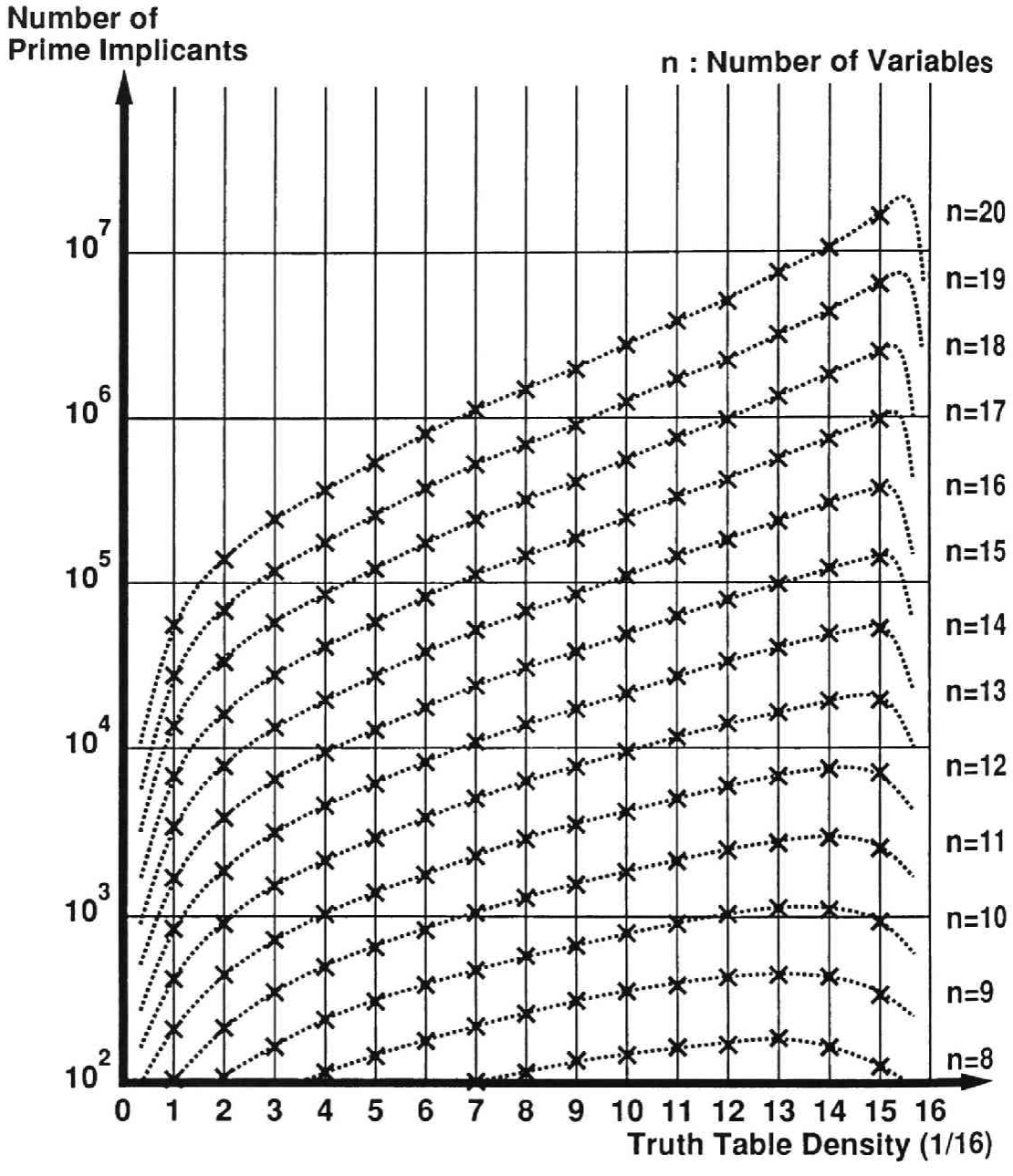


Figure 2.8: Average Number of Prime Implicants of 8- to 20-Variable Boolean Functions

is obtained by the experiments in section 2.6. From this figure, we can observe that the average number of prime implicants increases exponentially for the number of variable, and for every number of variables, the average number of prime implicants rises to the peak where the truth table density is very high.

2.8 Conclusion

In this chapter, three vector algorithms for generating all prime implicants of a given Boolean function have been proposed, and the required time and space for an arbitrary n-variable Boolean function have been shown. We have also shown that the proposed algorithms are much faster than any other conventional algorithm by benchmark results on a vector supercomputer FACOM VP-400E. It has been shown that the generation of prime implicants can be performed efficiently on a vector supercomputer by developing good vector algorithms and introducing effective data structure and a table look-up technique in their implementation.

As an application of the proposed algorithm, we have shown the results related to the number of prime implicants of Boolean functions. We have shown that Igarashi's conjecture on the maximum number of n-variable Boolean functions is true for $n = 5$ and 6.

Chapter 3

Vector Algorithms for Manipulating Binary-Decision Diagrams

3.1 Introduction

Recent progress of semiconductor technologies has enabled us to realize larger and more sophisticated logic circuits. Today, it is almost impossible to design logic circuits efficiently and correctly without using computer-aided design (CAD) systems. However, with the growth of the scale of VLSI, CAD systems have revealed its problem of increasing time and storage for computation.

In such CAD systems as design verification, test generation or logic synthesis, the major part of computation is, or can be reduced to, the manipulation of Boolean functions. A typical process of Boolean function manipulation in CAD systems are as follows:

(1) Input the description of a given instance, then encode the description into Boolean functions and represent them by an internal data structure of the system.

(2) Compute Boolean operations such as NOT, AND, OR and EXOR.

(3) Obtain the results of comparison (i. e., equivalence check) of two

Boolean functions or of substitution of 0 or 1 for variables of a Boolean function.

Primary operations in the above Boolean function manipulation are

(A) the unary operation for a Boolean function, i. e., NOT,

(B) binary operations for Boolean functions, such as AND, OR and EXOR,

(C) comparison of two Boolean functions, and

(D) substitution of 0 or 1 for a variable of a Boolean function.

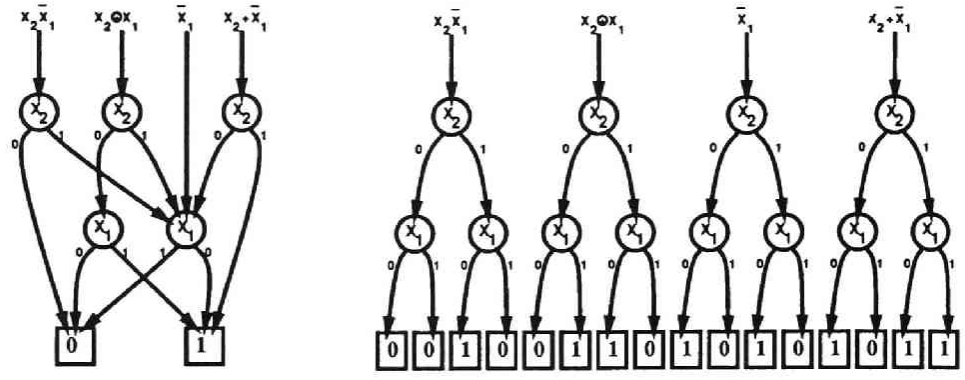
The efficiency of such Boolean function manipulation is closely connected with the internal representation of Boolean functions. For example, using truth tables as a representation of Boolean functions, (A), (B) and (C) require time proportional to 2^n for any n-variable Boolean function, while using Boolean formulas as a representation of Boolean functions, (C) is very difficult in general. Various representations of Boolean functions have been proposed for efficient Boolean function manipulation. Ordered Binary-Decision Diagram (OBDD), or simply Binary-Decision Diagram (BDD), is a graph representation of Boolean functions proposed by Akers[Ake78] and developed by Bryant[Bry86]. BDDs have excellent properties which makes (C) very easy and (A), (B) and (D) feasible in many practical cases.

At present, subroutine packages, called Boolean function manipulators, based on Shared Binary-Decision Diagram (SBDD), or multirooted BDD, are implemented on workstations which support primary operations of Boolean function manipulation of CAD systems. Several techniques for implementation of Boolean function manipulators based on SBDDs are proposed in order to reduce time and storage for manipulation, such as two kinds of hash tables[Bry86] and various attributed edges[MIY90,

BRB90]. These manipulators are now widely utilized in various applications such as design verification[FFK88, IDY90], test generation[CB89], logic synthesis[SYMF90] and so on.

Thus Boolean function manipulators based on SBDDs implemented on workstations are proven useful in CAD systems. However, according to the recent progress of the VLSI technology, it is required to manipulate larger and larger scale Boolean functions, which will exceed the computational power of workstations. In order to satisfy this requirement, the use of parallel machines or connection machines are studied[KC90]. In this chapter, an algorithm suitable for vector supercomputers is proposed. The proposed algorithm is based on so-called breadth-first manipulation to utilize the high performance of vector supercomputers, while the conventional algorithms for workstations are based on depth-first manipulation. The proposed breadth-first algorithm consists of two parts; an expansion phase and a reduction phase. In the expansion phase, new nodes sufficient to represent the resultant Boolean function are generated in a breadth-first manner from the root-node toward leaf-nodes. In the reduction phase, the nodes generated in the expansion phase are checked in a breadth-first manner from nodes nearby leaf-nodes toward the root-node. A modified algorithm which can manage efficiently SBDDs with output inverters, a kind of attributed edges, is also considered.

A Boolean function manipulator based on the proposed algorithm is implemented on the vector supercomputer HITAC S-820/80 at the University of Tokyo, and the results of the evaluations are shown in this chapter. From the experiments of constructing the SBDDs representing the Boolean functions of all the primary outputs and nets from a circuit description chosen from ISCAS'85 [BF85], the vector acceleration ratio on the S-820/80 is 5.3 to 27.8. Our manipulator on the S-820/80 is faster than that of Minato et al. on the workstation Sun3/60[MIY90] by up to



(a) A Shared Binary-Decision Diagram (b) Binary Decision Trees

Figure 3.1: Binary Decision Trees and a Shared Binary-Decision Diagram

130 times. In addition, as an example of applications of SBDDs, a design verification system based on computation tree logic (CTL) model checker is implemented and the experimental results are shown in this chapter.

In the following section, basic explanation on SBDDs and additional explanation on a vector supercomputer will be described. In section 3, a new algorithm will be proposed. In section 4, experimental results of the Boolean function manipulator will be shown. In section 5, an application to CTL model checker will be described. Section 6 will provide some concluding remarks.

3.2 Preliminaries

3.2.1 Shared Binary-Decision Diagram (SBDD)

An *Ordered Binary-Decision Diagram (OBDD)*, or simply a *Binary-Decision Diagram (BDD)*, is a directed acyclic graph which represents a Boolean function[Ake78, Bry86]. A *Shared Binary-Decision Diagram*

(SBDD) is a multirooted directed acyclic graph which represents multiple Boolean functions[MIY90, BRB90]. An example of an SBDD is shown in Figure 3.1 (a). This graph represents four Boolean functions corresponding to four *root-edges*. The node (vertex) pointed to by a root-edge of a Boolean function is referred to as the *root-node* of the Boolean function. There are (at most) two terminal nodes, *leaf-nodes*, which are labeled by 0 and 1. Every *non-terminal node*, or simply *node*, is labeled by a Boolean variable. Every node has exactly two outgoing edges (arcs). They are labeled by '0' and '1'. They are called *'0' edge* and *'1' edge*, respectively.

An SBDD is defined as the graph obtained from binary decision trees representing Boolean functions (Figure 3.1 (b)) by repeating the following transformations until they are not applicable.

(a) To share isomorphic sub-graphs.

(b) To delete every node both of whose '0' edge and '1' edge point to the same node.

Note that no Boolean variable appears more than once in every path of an SBDD, and the variables appear in a fixed order in all the paths of an SBDD. An integer number, called *level*, is assigned to every Boolean variable with respect to the ordering of the variables in an SBDD. This assignment corresponds to the ordering so that a variable nearer to the leaf-nodes has a smaller number. We denote the variable with level i as x_i.

Also note that there is no node which has either of the following property;

- *A redundant node*: A node whose '0' edge is the same as its '1' edge.
- *Non-unique nodes*: A node whose '0' edge and '1' edge are equivalent to respective those of another node of the same level.

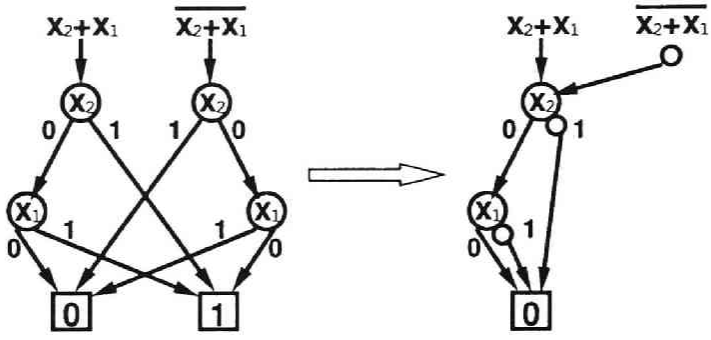


Figure 3.2: Output Inverters

SBDDs have following excellent properties:

- Canonical, i. e., there are no two root-edges of a graph which point to the different nodes and yet represent the same Boolean function. The equivalence of two Boolean functions represented by an SBDD can be tested simply by comparing the root-edges corresponding to the functions.
- The size of the graph is small for many practical Boolean functions.
- The manipulations for various operations on Boolean functions represented by an SBDD can be performed in time proportional to the number of the nodes of the graph[Bry86].

In order to reduce the number of nodes and/or the time for manipulation of an SBDD, various *attributed edges* are proposed, such as output inverters, input inverters, variable shifters, and so on[MIY90, BRB90]. Among them, *output inverter* is effective in realizing high-speed SBDD manipulation, which is the aim of this chapter. Output inverter is the attribute indicating to complement the Boolean function of the subgraph

pointed to by the edge (Figure 3.2). Employing this attribute, the number of nodes of SBDDs can be reduced to a half in the best case, NOT operations can be executed without traversing the graph and whether two given Boolean functions are complement to each other or not can be examined without traversing the graph. Abuse of output inverters break the important property of SBDDs giving unique representations of Boolean functions. The following limitations are placed in order to keep this property:

(A) Output inverters must not be used in '0' edges, i. e., output inverters are used only in '1' edges or in the root-edges.

(B) The leaf-node must be unique. In this thesis, only 0 is used as the leaf-node.

3.2.2 Conventional Algorithm for Manipulating SBDDs

The principal tasks of Boolean function manipulators are

(A) the unary operation for a Boolean function, i. e., NOT,

(B) binary operations for Boolean functions, such as AND, OR and EXOR,

(C) comparison of two Boolean functions, and

(D) substitution of 0 or 1 for a variable of a Boolean function.

If the Boolean functions are represented by an SBDD, (C) can be achieved only by comparing two root-edges of the given functions, and (A) is also easily realizable if output inverters are employed. In this section, the conventional algorithms for (B) and (D) are described. In addition, another operation *shift of variables* is described. (An operation over BDD to perform (B) is often called *APPLY*.)

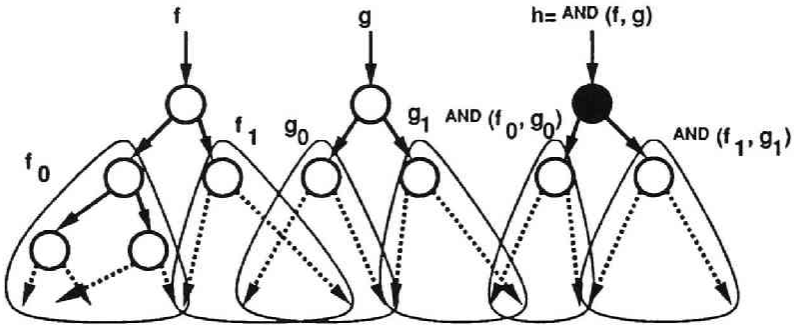


Figure 3.3: Conventional Recursive Algorithm

Binary Operations

For example, let us consider a conventional recursive algorithm[MIY90, BRB90] for generating the graph that represents the Boolean function h=AND(f, g), where f and g are Boolean functions represented by a given SBDD with two root-edges e_f and e_g. We denote the levels of the root-nodes of f and g as L_f and L_g, and let $L_h = max(L_f, L_g)$. Recall definitions of $f(x_i = 0)$ and $f(x_i = 1)$ appeared in section 2.2. We will denote them simply f_0 and f_1, respectively, if x_i is obvious from context.

[A Conventional Depth-First Algorithm for AND]
Examine the given two root-edges e_f and e_g, and execute one of the following statements:

(1) If e_f and/or e_g point(s) to the leaf-node 0, then return the edge pointing to the leaf-node 0.

(2) If e_f (e_g) points to the leaf-node 1, then return e_g (e_f).

(3) If $e_f = e_g$, then return e_f.

(4) Otherwise, compute the root-edges of $h(x_{L_h} = 0)$ = AND $(f(x_{L_h} = 0), g(x_{L_h} = 0))$ and $h(x_{L_h} = 1)$ = AND $(f(x_{L_h} = 1), g(x_{L_h} = 1))$,

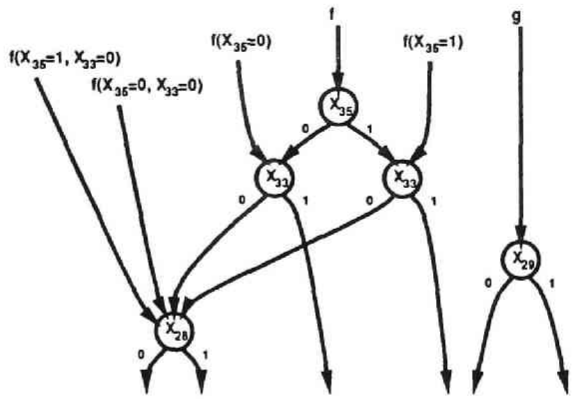


Figure 3.4: Effect of the Operation-Result-Table

recursively using this same algorithm. Then examine the obtained root-edges of $h(x_{L_h} = 0)$ and $h(x_{L_h} = 1)$ and execute either of the following statements.

(4.1) If $h(x_{L_h} = 0) = h(x_{L_h} = 1)$, then return the root-edge of $h(x_{L_h} = 0)$.

(4.2) If there exists a node whose level is L_h and whose '0' edge and '1' edge point to the root-node of h_0 and h_1, respectively, then return the edge pointing to this node.

(4.3) Otherwise, generate a new root-node for h whose level is L_h and whose '0' edge and '1' edge point to the root-node of $h(x_{L_h} = 0)$ and $h(x_{L_h} = 1)$, respectively (Figure 3.3). Return the edge pointing to this new node.

For (4.2) of the above algorithm, a hash table, *node-table*, is introduced. It manages all the nodes of the graph. The keys of the node-table are the level, '0' edge and '1' edge of a node.

Another hash table (or may be hash-based cache), called an *operation-*

result-table is introduced to avoid repetition of the same operations. The keys of the operation-result-table are a Boolean operator (e. g. AND) and given two root-edges. Every time when (4) of the above algorithm is completed, the result is registered to this table. The time for executing (4) is saved if the result is found in this table before executing the statement (4). This table is important especially when there are many reconvergences in the sub-graphs of the given functions. A simple example is shown in Figure 3.4. Let us consider the case of computing AND(f, g). According to the above algorithm, one must obtain AND$(f(x_{35} = 0), g)$ and AND$(f(x_{35} = 1), g)$. In order to obtain AND$(f(x_{35} = 0), g)$, both AND$(f(x_{35} = 0, x_{33} = 0), g)$ and AND$(f(x_{35} = 0, x_{33} = 1), g)$ are required, while in order to obtain AND$(f(x_{35} = 1), g)$, both AND$(f(x_{35} = 1, x_{33} = 0), g)$ and AND$(f(x_{35} = 1, x_{33} = 1), g)$ are required. Because $f(x_{35} = 0, x_{33} = 0)$ is equal to $f(x_{35} = 1, x_{33} = 0)$, the result of AND$(f(x_{35} = 0, x_{33} = 0), g)$ can be reused as the result for AND$(f(x_{35} = 1, x_{33} = 0), g)$ if the operation-result-table is introduced.

The other binary operations such as OR or EXOR can be done in the same way.

Substitution of 0 or 1 for a Variable

Substitution of 0 for a variable x_i of a Boolean function f, i. e., generation of an SBDD which represents Boolean function $f(x_i = 0)$ can be done by the following recursive algorithm. Generation of an SBDD which represents Boolean function $f(x_i = 1)$ can be performed similarly.

[A Conventional Depth-First Algorithm for $f(x_i = 0)$]
Examine the given root-edge e_f, and execute either of the following statements:

(1) If e_f points to the leaf-node 0 (1), then return the edge pointing to

the leaf-node 0 (1).

(2) If $i > L_f$, then return e_f.

(3) If $i = L_f$, then return the edge pointing to the node pointed to by the '0' edge of the root-edges of f.

(4) Otherwise, compute the root-edges of $g(x_{L_f} = 0) = f(x_{L_f} = 0, x_i = 0)$ and $g(x_{L_f} = 1) = f(x_{L_f} = 1, x_i = 0)$ recursively. Then examine the obtained root-edges of $g(x_{L_f} = 0)$ and $g(x_{L_f} = 1)$ and execute either of the following statements:

(4.1) If $g(x_{L_f} = 0) = g(x_{L_f} = 1)$, then return the root-edge of $g(x_{L_f} = 0)$.

(4.2) If there exists a node whose level is L_f and whose '0' edge and '1' edge point to the root-node of g_0 and g_1, respectively, then return the edge pointing to this node.

(4.3) Otherwise, generate a new root-node for g whose level is L_f and whose '0' edge and '1' edge point to the root-node of $g(x_{L_f} = 0)$ and $g(x_{L_f} = 1)$, respectively. Return the edge pointing to this new node.

As the algorithm for AND, node-table and operation-result-table is used for (4). The keys of the operation-result-table are a operation ("substitute 0" or "substitute 1") and a given root-edge.

Shift of Variables

There are some applications (such as CTL model checker described in section 3.5) which needs the operation of shifting all the subscripts of variables of a given Boolean function, i. e., generation of an SBDD which represents Boolean function $f(x_{1+c}, x_{2+c}, \ldots, x_{n+c})$ when positive constant

c and the root-edge for the Boolean function $f(x_1, x_2, \ldots, x_n)$ are given. This can be performed by the following recursive algorithm:

[A Conventional Depth-First Algorithm for Shift of Variables] Examine the given root-edge e_f, and execute either of the following statements:

(1) If e_f points to the leaf-node 0 (1), then return the edge pointing to the leaf-node 0 (1).

(2) Otherwise, compute the root-edges for $g(x_{L_f+c} = 0) = f(x_{1+c}, x_{2+c}, \ldots, x_{L_f-1+c}, 0)$ and $g(x_{L_f+c} = 1) = f(x_{1+c}, x_{2+c}, \ldots, x_{L_f-1+c}, 1)$ recursively. Then examine the obtained root-edges of $g(x_{L_f+c} = 0)$ and $g(x_{L_f+c} = 1)$ and execute either of the following statements:

(2.1) If $g(x_{L_f+c} = 0) = g(x_{L_f+c} = 1)$, then return the root-edge of $g(x_{L_f+c} = 0)$.

(2.2) If there exists a node whose level is $(L_f + c)$ and whose '0' edge and '1' edge point to the root-node of g_0 and g_1, respectively, then return the edge pointing to this node.

(2.3) Otherwise, generate a new root-node for g whose level is $(L_f + c)$ and whose '0' edge and '1' edge point to the root-node of $g(x_{L_f} = 0)$ and $g(x_{L_f} = 1)$, respectively.

As the algorithm for AND, node-table and operation-result-table is used for (2). The keys of the operation-result-table are a operation ("shift of variables"), a given root-edge and a given value c.

3.2.3 High-Speed Vector Indirect Store

See also section 2.3 of chapter 2 for basic explanation of vector supercomputers. In this section, a special feature of HITAC S-820/80 on which we have developed our Boolean function manipulator is described.

Vector supercomputer HITAC S-820/80 has a special vector instruction which enables us high-speed vector store in list vector access mode.

By normal vector indirect store instruction, it is guaranteed that the latest store is valid at the location where confliction occur. As illustrated in Figure 3.5 (a), `B[2]` will be 62 (not 23) and `B[5]` will be 91 (not 84) after the normal vector indirect store instruction.

The special vector indirect store instruction of HITAC S-820/80 is about 3 times faster than the normal vector indirect store instruction. However, if there are confliction, it is guaranteed only that one of the store is valid. For example in Figure 3.5 (b), it is not defined that `B[2]` will be 23 or 68.

This special instruction is designed to be used when user know that there is no confliction. For example, user can inform FORTRAN compiler by statement

```
*VOPTION VIST
```

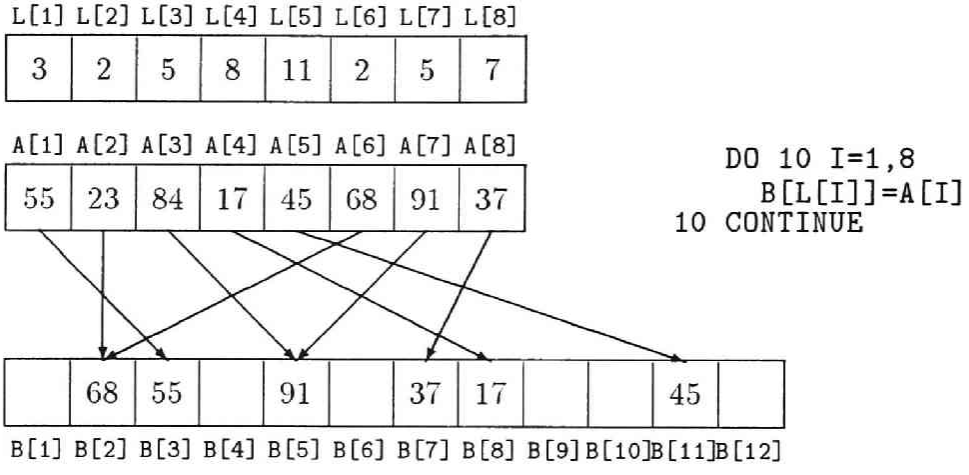
that the special instruction may be used in the succeeding `DO` loop. In this chapter, we utilize this instruction for hash tabel access where conflictions may be occur.

3.3 Breadth-First Vector Algorithm for Manipulating SBDDs

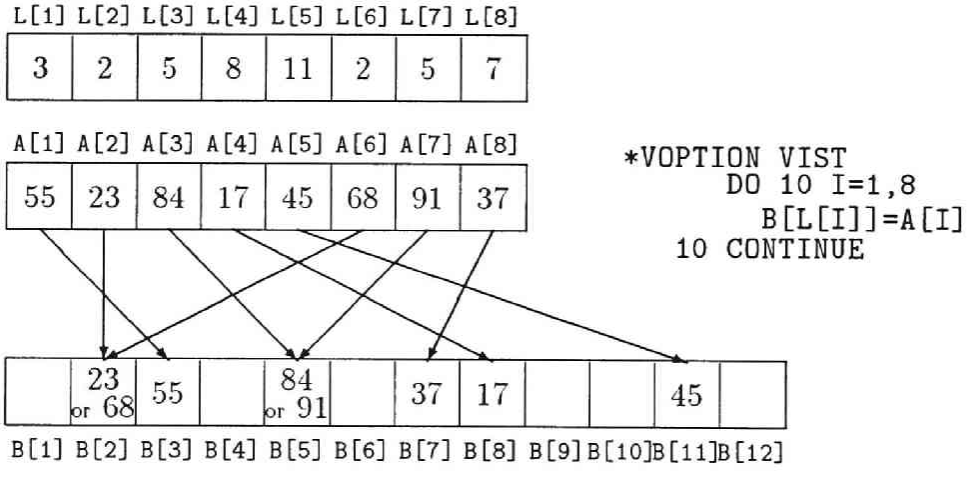
3.3.1 Basic Idea

As mentioned in the preceding section, the conventional algorithm for managing SBDDs is based on a recursive procedure (or a depth-first operation), which is not suitable for vector processing. In this section, a breadth-first algorithm for managing SBDDs is proposed.

The proposed algorithm consists of two phases: an *expansion phase*



(a) Normal Vector Indirect Store



(b) High-Speed Vector Indirect Store of HITAC S-820/80

Figure 3.5: Vector Indirect Store

and a *reduction phase*. In the expansion phase, new nodes sufficient to represent the resultant function are generated in a breadth-first manner from the root-node toward the leaf-nodes. In the reduction phase, the nodes generated in the expansion phase are checked and the redundant nodes and the non-unique nodes are removed in a breadth-first manner from the nodes nearby the leaf-nodes toward the root-node. The nodes generated in the expansion phase are called *temporary nodes*, while the nodes which already exist are called *permanent nodes*.

3.3.2 Algorithm

In this section, the breadth-first algorithm for binary operation is described. Other operations, i. e., substitution of 0 or 1 for a variable and the shift of variables, are also implemented similarly.

Expansion Phase

An input for the expansion phase is a triple $(op,\ e_f,\ e_g)$, where op is a Boolean operator to be executed, such as AND, OR or EXOR, and e_f and e_g are root-edges of argument Boolean functions represented by an SBDD. This triple is referred to as a *requirement*. A requirement $(op,\ f,\ g)$ requires to compute the root-edge for the resultant function of $op(f, g)$. During processing a requirement, new requirements are generated for computing the operations between sub-functions or sub-sub-functions ... of the argument functions. Actually a requirement corresponds to a procedure call in the depth-first algorithm. A queue called a *requirement queue* is introduced to manage these requirements, which makes our procedure breadth-first. (The procedure would be depth-first if a stack is used instead of the queue.)

For a given requirement $(op,\ e_f,\ e_g)$, a new root-node is not always

generated. A new node should not be generated if a node representing the result of $op(f, g)$ already exists. For example, if the result of $op(f, g)$ is found trivially (the cases (1) (3) in the algorithm in the depth-first algorithm), or found by looking up the operation-result-table, a new node is not generated. In these cases, the judgement can be performed immediately from e_f and e_g. However, in general, there are cases where the existence of the root-node of $op(f, g)$ cannot be determined until the whole graph for the sub-functions of $op(f, g)$ is constructed. In this breadth-first algorithm, a temporary node is generated in such cases. Whether the temporary node is actually essential or not is examined in the reduction phase.

Following procedure is the expansion phase. Initially, the requirement queue is empty, and there is no temporary node.

[Expansion Phase of Binary Operations]
Put a given requirement (op, e_f, e_g) to the requirement queue and repeat the following operations for every requirement in the queue until the queue becomes empty.

(1) If the root-node representing the result of $op(f, g)$ is trivial, then return the edge pointing to the node.

(2) If the root-node representing the result of $op(f, g)$ is found in the operation-result-table, then return the edge found in the table.

(3) Otherwise, generate a new temporary node and return the edge pointing to the temporary node. At the same time, register the edge pointing to the temporary node to the operation-result-table as the result of $op(f, g)$ and put new requirements (op, e_{f_0}, e_{g_0}) and (op, e_{f_1}, e_{g_1}) to the requirement queue, whose result will be '0' edge and '1' edge, respectively, of this temporary node.

Note that the temporary nodes must be registered to the operation-result-table in the expansion phase in order to avoid repetition of the same operation (recall the example of Figure 3.4). On the other hand, the registration to the node-table is done in the reduction phase.

Also note that the total number of requirements processed in the above procedure is exactly the same as the number of procedure calls in the depth-first algorithm in section 3.2.2 and thus there is no serious increase on the computation cost. The only drawback of our algorithm is the increase of the storage required for temporary nodes.

This procedure is suitable for vector processing because of the following reasons:

(1) High vectorization ratio. All of the repeated operations can be executed by vector instructions.

(2) Long vector length. All requirements existing in the queue can be processed simultaneously.

List vector access is utilized in the whole operations of the expansion phase by means of referring to the queue as a list vector. Trivial requirements and non-trivial requirements can be exclusively executed using conditional vector operations. New requirements are put to the queue using compress operations. Registration to the operation-result-table is also vectorizable by the technique which will be stated in section 3.3.3.

Reduction Phase

There may be redundant nodes and non-unique nodes among the temporary nodes generated in the expansion phase. The main tasks of the reduction phase are to find redundant nodes and non-unique nodes and to remove them. In our algorithm, these tasks are executed in a breadth-first manner from the nodes nearby the leaf-nodes toward the root-node.

In addition, temporary nodes which are neither the redundant nodes nor the non-unique nodes are registered to the node-table.

In practice, the removal of the redundant nodes and the non-unique nodes must be performed at the end of the reduction phase because there are edges pointing to these nodes. In our algorithm, the nodes classified as redundant nodes or non-unique nodes are marked as *useless nodes*. Every useless node has a forwarding pointer to indicate the node that takes the place of the useless node.

Following procedure is the reduction phase.

[Reduction Phase of Binary Operations]
Repeat the following operations while there are temporary nodes.

(1) For every temporary node whose '0' edge or '1' edge point to a useless node, redirect the edge so as to point to the node pointed to by the forwarding pointer of the useless node.

(2) For every temporary node both of whose '0' edge and '1' edge are not temporary nodes (i. e., permanent nodes or the leaf-nodes), execute following statements:

  (2.1) If its '0' edge and '1' edge are the same, mark the node as a useless node, and set its forwarding pointer to point to the node pointed to by its '0' edge.

  (2.2) If there is a node which is equivalent to this node and registered in the node-table, mark the temporary node as a useless node, and set its forwarding pointer to point to the node which is registered in the node-table.

  (2.3) Otherwise, register the node to the node-table, and mark it as permanent node.

This procedure is also suitable for vector processing because all temporary nodes whose '0' edges and '1' edges are not temporary nodes can be processed simultaneously, and almost all operations are vectorizable. Vectorization of registration to the node-table is discussed in section 3.3.3.

Example

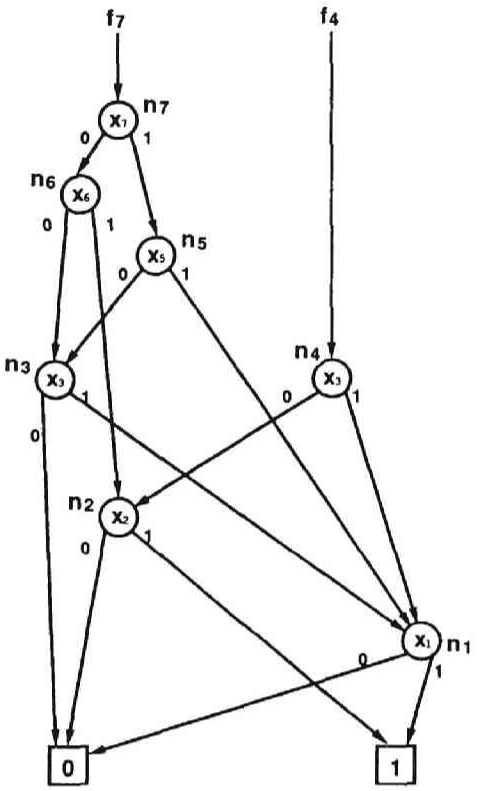
Figure 3.6 illustrates the proposed algorithm via an example. This example shows the process of AND operation whose arguments are Boolean functions represented by the root-edges e_{f_7} and e_{f_4} pointing to the nodes n_7 and n_4, respectively, of the SBDD in Figure 3.6 (a). Here we denote a Boolean function whose root-node is n_k by f_k. For simplicity, the operation-result-table is assumed to be initially empty.

At the beginning of the expansion phase, the requirement (AND, e_{f_7}, e_{f_4}) is put to the requirement queue.

Because the result of the requirement (AND, e_{f_7}, e_{f_4}) is not trivial and not found in the operation-result-table, a new temporary node n_8 is allocated as the root-node of the result of (AND, e_{f_7}, e_{f_4}). The level of the new node is $max(L_{f_7}, L_{f_4})=7$. The edge f_8 pointing to the new node n_8 is registered to the operation-result-table. The requirement (AND, e_{f_7}, e_{f_4}) is dequeued and the new requirements (AND, $e_{f_7(x_7=0)}$, $e_{f_4(x_7=0)}$)=(AND, e_{f_6}, e_{f_4}) and (AND, $e_{f_7(x_7=1)}$, $e_{f_4(x_7=1)}$)=(AND, e_{f_5}, e_{f_4}), corresponding to the '0' edge of n_8 and '1' edge of n_8 respectively, are put to the queue.

Similarly, two requirements (AND, e_{f_6}, e_{f_4}) and (AND, e_{f_5}, e_{f_4}) are processed simultaneously (Figure 3.6 (b)). Now, there are four requirements (AND, e_{f_3}, e_{f_4}), (AND, e_{f_3}, e_{f_4}), (AND, e_{f_2}, e_{f_4}) and (AND, e_{f_1}, e_{f_4}), corresponding to the '0' edge of n_9, the '0' edge of n_{10}, the '1' edge of n_9 and the '1' edge of n_{10}, respectively.

Now, the requirement (AND, e_{f_3}, e_{f_4}) corresponding to the '0' edge of n_9 is processed. Because the result of the requirement is not trivial and



Operation-Result-Table

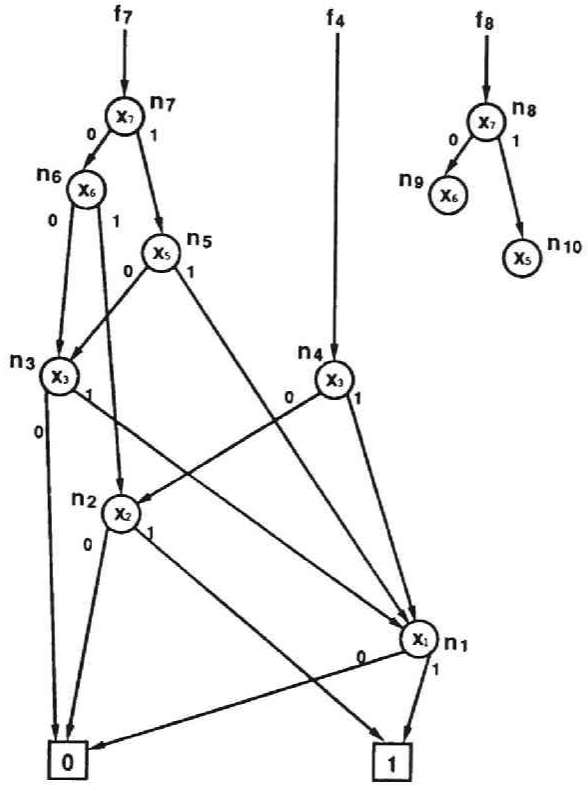
operand 1	
operand 2	
operation	
result	

Node-Table

n_7	n_4	n_2	n_6	$n_1 n_5$	n_3

(a) An SBDD before Operation

Figure 3.6: Example of the Breadth-First Manipulation



Operation-Result-Table

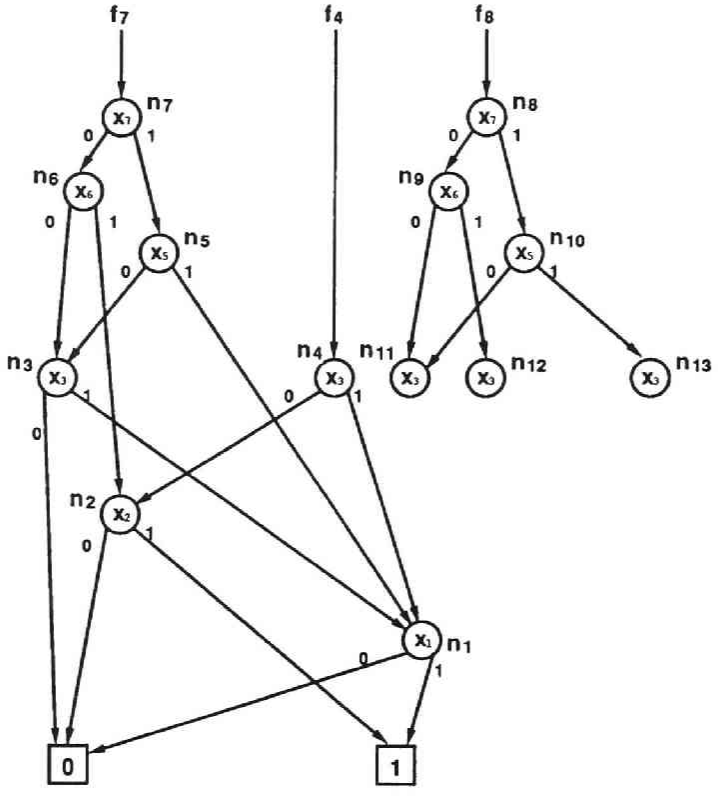
operand 1	f_5	f_7	f_6
operand 2	f_4	f_4	f_4
operation	AND	AND	AND
result	f_{10}	f_8	f_9

Node-Table

n_7	n_4	n_2	n_6	$n_1 n_5$	n_3

(b) The SBDD at the End of the 2nd Stage of the Expansion Phase

Figure 3.6: Example of the Breadth-First Manipulation (Continued)



Operation-Result-Table

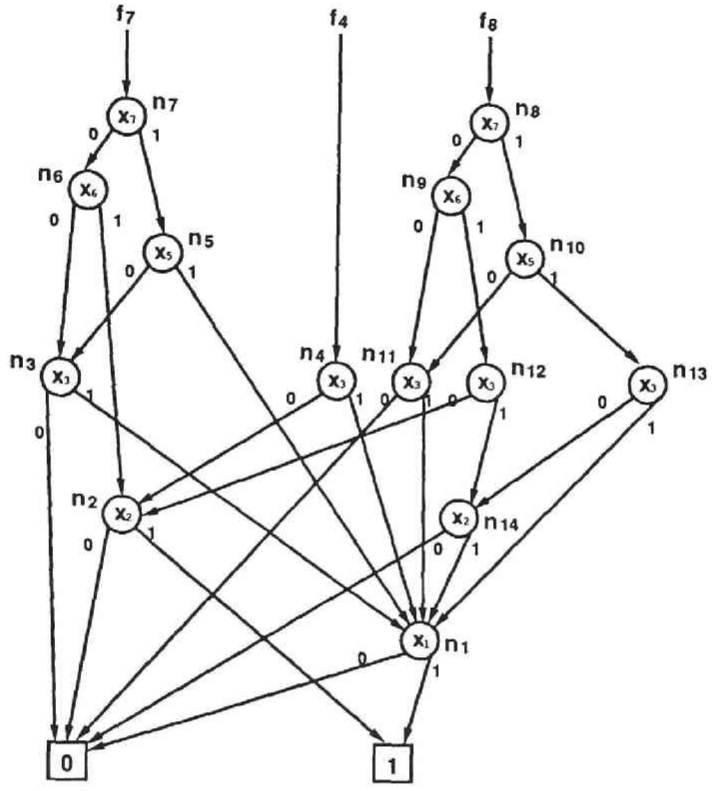
operand 1	f_5	f_2	f_7	f_3	f_6	f_1
operand 2	f_4	f_4	f_4	f_4	f_4	f_4
operation	AND	AND	AND	AND	AND	AND
result	f_{10}	f_{12}	f_8	f_{11}	f_9	f_{13}

Node-Table

n_7	n_4	n_2	n_6	$n_1 n_5$	n_3

(c) The SBDD at the End of the 3rd Stage of the Expansion Phase

Figure 3.6: Example of the Breadth-First Manipulation (Continued)



Operation-Result-Table

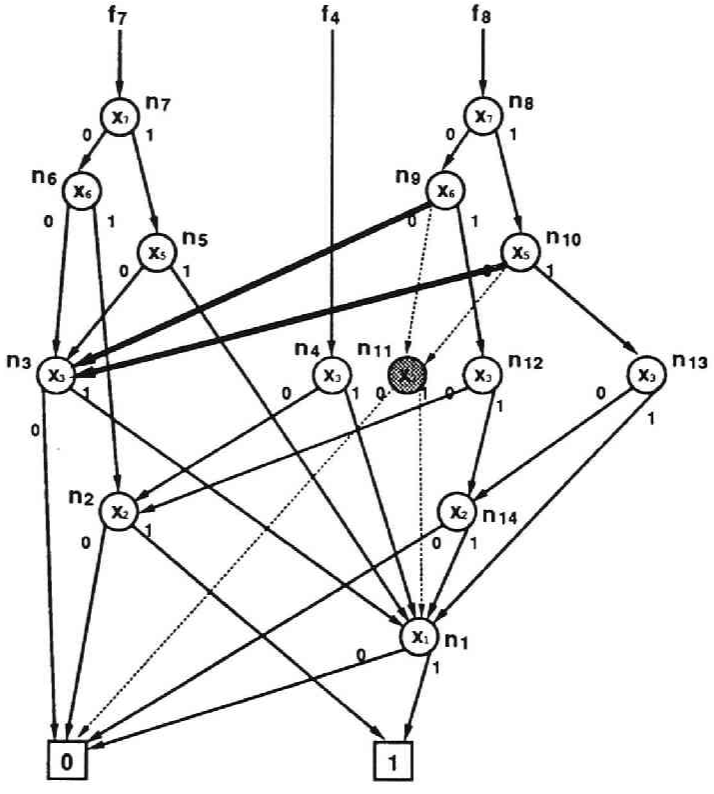
operand 1	f_5	f_2	f_7	f_2	f_3	f_6	f_1
operand 2	f_4	f_4	f_4	f_1	f_4	f_4	f_4
operation	AND	AND	AND	AND	AND	AND	AND
result	f_{10}	f_{12}	f_8	f_{14}	f_{11}	f_9	f_{13}

Node-Table

n_7	n_4	n_2	n_6	$n_1 n_5$	n_3

(d) The SBDD at the End of the Expansion Phase

Figure 3.6: Example of the Breadth-First Manipulation (Continued)



Operation-Result-Table

operand 1	f_5 f_2 f_7 f_2 f_3 f_6 f_1
operand 2	f_4 f_4 f_4 f_1 f_4 f_4 f_4
operation	AND AND AND AND AND AND AND
result	f_{10} f_{12} f_8 f_{14} f_{11} f_9 f_{13}

Node-Table

n_7 n_{13} n_4 n_{14} n_8 n_2 n_9 n_6 n_1 n_5 n_{10} n_3 n_{12}

(e) The SBDD after Operation

Figure 3.6: Example of the Breadth-First Manipulation (Continued)

not found in the operation-result-table, a new node n_{11} is generated and is registered to the operation-result-table. Next, the requirement (AND, e_{f_3}, e_{f_4}) corresponding to the '0' edge of n_{10} is processed. Note that this requirement is the same as one just processed. According to the operation-result-table, the result of this requirement is f_{11}, so that the '0' edge of n_{10} is directed to n_{11} instead of a new distinct temporary node. For the other two requirements in the queue, new temporary nodes n_{12} and n_{13}, respectively, are generated (Figure 3.6 (c)). Six requirements corresponding to '0' edges and '1' edges of three new temporary nodes are put to the queue.

The result of the requirement (AND, 0, e_{f_2}) corresponding to the '0' edge of n_{11} is trivial, i. e., 0, therefore, the '0' edge of n_{11} is directed to the leaf-node 0. The result of the requirement (AND, e_{f_2}, e_{f_2}) corresponding to the '0' edge of n_{12} is trivial, i. e., f_2, therefore, the '0' edge of n_{12} is directed to n_2. In the same way, the '1' edge of n_{11} and the '1' edge of n_{13} are directed to n_1. For the requirement (AND, e_{f_2}, e_{f_1}) corresponding the '0' edge of n_{13}, a new node n_{14} is generated, and registered to the operation-result-table. The '1' edge of n_{12} is directed to n_{14} according to the operation-result-table.

There are two requirements corresponding to the '0' edge and '1' edge of n_{14}. They are both trivial. Figure 3.6 (d) shows the SBDD at the end of the expansion phase.

Then the reduction phase begins. The temporary nodes both of whose '0' edges and '1' edges are not temporary nodes are processed, i. e., n_{11} and n_{14} are processed. n_{11} is not redundant (i. e., the '0' edge of n_{11} is different from the '1' edge of n_{11}). But according to the node-table, there is an isomorphic node, i. e., n_3 whose level, '0' edge and '1' edge are the same as n_{11}, therefore, n_{11} is marked as useless node and its forwarding pointer is set to point to n_3, and n_{11} is not registered to the node-table.

In addition, the entry of the operation-result-table of f_{11} is modified to f_3. On the other hand, n_{14} is not redundant nor isomorphic to any other permanent nodes, therefore, registered to the node-table and marked as permanent node.

Next, n_{12} and n_{13} are processed in the same way, and both of them are registered to the node-table and marked as permanent nodes.

Next, n_9 and n_{10} are processed (Before processing, their '0' edges pointing to the useless node n_{11} are redirected to n_3 according to the forwarding pointer).

Finally, n_8 is processed, and the result for the initial requirement is the SBDD with the root-edge e_{f_8} (Figure 3.6 (e)). The useless node n_{11} is removed now.

3.3.3 Vectorization of Hash Table Access

The access to the operation-result-table in the expansion phase and the access to the node-table in the reduction phase cannot be vectorized in a straightforward manner.

If the expansion phase is vectorized in a straightforward manner (i. e., refering to the operation-result-table simultaneously, generating a new temporary nodes simultaneously, then registering them to the operation-result-table simultaneously), duplication of temporary nodes occur when the same requirements are processed simultaneously. For example, consider the third stage of the expansion phase of the example of Figure 3.6. There are requirements corresponding to the '0' edge of n_9 and '0' edge of n_{10}. The former requirement is the same as the latter one. They are processed together in the next stage. When at first the operation-result-table is referred to complying with both requirements, the result is not yet written. Therefore, new temporary nodes are generated complying with the both requirements, which cause the duplication of the temporary

nodes.

This can be avoided by checking the operation-result-table again just after the registration to the operation-result-table. Those temporary nodes are removed whose registration to the operation-result-table is overwritten by the registration of another temporary node which is processed simultaneously. If there are duplicated requirements, the registrations to the operation-result-table conflict, because the values of the hash function for these requirements are the same. After the registrations to the operation-result-table, only the last registration is valid in the entry of the table where the conflict occurred. In this way, by means of the one-more check of the operation-result-table, all but one duplicated temporary nodes can be removed. Note that the valid registration in the entry of the table where the conflict occurred may be arbitrary one of the registrations to this entry. This fact enables us to employ the high-speed vector indirect store instruction of HITAC S-820/80. In addition, the operation-result-table can be implemented as a hash-based cache in practice[BRB90], therefore, the registrations and check can be simply implemented.

In the case of the access to the node-table in the reduction phase, the basic idea for vectorization is similar to the operation-result-table. However, the node-table cannot be implemented as a hash-based cache: the node-table must keep all the registered nodes. The node-table is constructed by an array T of pointers; every pointer corresponds to a value of hash function and points to the head of the linked list of the registered nodes.

The simultaneous references to the node-table according to the temporary nodes to be processed in a stage of the reduction phase are vectorized as follows:

(1) Generate an array L of the pointers to the temporary nodes to be

processed. Compute the value of the hash function for every temporary nodes and copy the pointer in T to a work area p of the temporary node.

(2) Check every p pointed to by L. If p is nil, then currently there is no registered node which is equivalent to the temporary node. Remove from L the pointers to these nodes.

(3) Refer to the permanent nodes pointed to by p's pointed to by L. Check whether each permanent node is actually equivalent to the temporary node. If so, the temporary node is marked with useless node, and set its forwarding pointer to point to the permanent node pointed to by p. If not, the temporary node and the permanent node pointed to by p have, occasionally, the common value of the hash function, but, in fact, distinct each other. For such temporary nodes, update p's by copying the link pointers of the permanent nodes pointed to by p's.

(4) Remove from L the pointers to the useless nodes. Repeat from step (2) until L become empty.

At first, the vector references to the 1st nodes in the linked lists are executed. For the temporary nodes which accessed to the distinct permanent nodes, the vector references to the 2nd nodes in the linked lists are executed, and so forth. The removal of pointers from L is vectorized with the vector compress operation.

The simultaneous registrations to the node-table and the simultaneous one-more check of the node-table are also vectorized in a similar way.

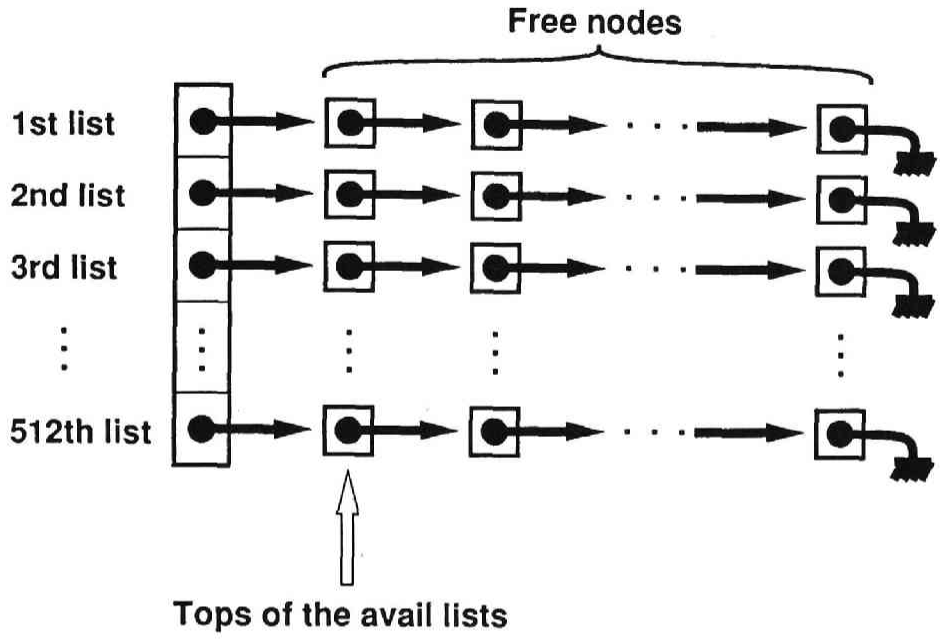


Figure 3.7: Multiple Avail Lists

3.3.4 Management of Free Nodes

As mentioned in the explanation of the reduction phase, the useless nodes are removed from the graph at the end of the operation. In order to reuse these nodes, the use of avail list is considered. There are well-known efficient procedures for a node to be inserted to and deleted from the top of a linked list. However, multiple nodes cannot be inserted to or deleted from a linked list simultaneously.

In order to vectorize the insertion and deletion of multiple nodes, we introduce multiple avail lists. Instead of single pointer for the top of an avail list, an array of 512 pointers for the tops of the 512 avail lists is introduced (Figure 3.7). Initially, all the avail lists have the same number

of the free nodes and the total number of the free nodes are set to the variable $\#free$. When $n(\leq 512)$ free nodes are required, the top nodes in the $(\mathrm{mod}(\#free - n, 512) + 1)$th through $(\mathrm{mod}(\#free - 1, 512) + 1)$th avail lists are used, then $\#free$ is decreased by n. When more than 512 free nodes are required, the vector operations of the length (up to) 512 are repeated. The insertion of the multiple useless nodes is done by the above steps backward. The number of the avail lists is chosen to be 512 because the length of the vector registers of the HITAC S-820/80 are 512.

This technique is also used in the process of the garbage collection [BRB90]. Garbage collection collects the useless nodes in the graph according to the 'free' declarations by the application program. The nodes used for the intermediate result of the application program can be reused by means of the 'free' declarations and the garbage collection.

3.3.5 Management of SBDDs with Output Inverters

In this section, a method for managing the SBDDs with output inverters is described. As mentioned in section 3.2.1, various attributed edges are proposed. The output inverters are efficient to reduce the time for manipulation. In the conventional recursive algorithm, the attributes of the edge pointing to a new node can be easily determined using the result of the recursive steps for its sub-functions. In the vector algorithm, suitable methods for applying output inverters varies among the operations.

Under the limitations of the use of output inverters mentioned in section 3.2.1, the following property holds. Methods described so far are based on this property.

[Property 1] The output inverter is attached to the root-edge of f if and only if the value of f is 1 when 0 is substituted to all the variables.

Binary Operation

In the binary operations, the output inverters of the edges can be decided without using the result of its sub-functions. Let us denote the existence of the output inverter on edge e as $oi(e)$, whose value is *true* if the output inverter exists and *false* otherwise. From Property 1 and an equation $h(0, 0, \ldots, 0) = op(f(0, 0, \ldots, 0), g(0, 0, \ldots, 0))$ where $h = op(f, g)$ and op is a binary operator such as AND, OR or EXOR, Property 2 follows.

[Property 2] oi(the root-edge of $op(f, g)$) = $op(oi$(the root-edge of f), oi(the root-edge of g))

Based on Property 2, a method is proposed by which the output inverters are computed in every stage of the breadth-first operation in the expansion phase and no more computation of output inverters is required in the reduction phase. The rules to be added to the expansion phase are as follows:

- Attach an output inverter to the root-edge of $op(e_f, e_g)$ iff $op(oi(e_f), oi(e_g))$ is *true*.
- For the requirement (op, e_{f_0}, e_{g_0}), attach an output inverter to the root-edge e_{f_0} (e_{g_0}) iff $oi(e_f)$ $(oi(e_g))$ is *true*.
- For the requirement (op, e_{f_1}, e_{g_1}), attach an output inverter to the root-edge e_{f_1} (e_{g_1}) iff $oi(e_f)$ $(oi(e_g))$ is different from oi('1' edge of the root-node of f (g)).
- Never attach an output inverter to the '0' edge corresponding to the requirement (op, e_{f_0}, e_{g_0}).
- Attach an output inverter to the '1' edge corresponding to the requirement (op, e_{f_1}, e_{g_1}) iff $op(oi(e_{f_1}), oi(e_{g_1}))$ is different from $op(oi(e_{f_0}).$ $oi(e_{g_0}))$.

Substitution of 0 and Shift of Variables

It is clear from Property 1, the output inverter of the root-edge of the Boolean function $f(x = 0)$ is the same as the output inverter of the root-edge of Boolean function f. The output inverter of the root-edge of the Boolean function $f(x_{1+c}, x_{2+c}, \ldots, x_{n+c})$ is also the same as the output inverter of the root-edge of Boolean function $f(x_1, x_2, \ldots, x_n)$. Therefore, there is no need to compute the output inverters during these operations.

Substitution of 1

The output inverter of the root-edge of the Boolean function $f(x = 1)$ cannot be decided only by the output inverter of the root-edge of Boolean function f in general. For example, output inverter is not attached to the root-edge of a Boolean function f=AND(x_2, NOT(x_1)) nor the root-edge of $f(x_1 = 1)$, while output inverter is attached to the root-edge of $f(x_2 = 1)$ (recall Property 1). Therefore, the output inverters must be computed during the reduction phase of the breadth-first algorithm.

3.3.6 Parallelization Multiple Operations

If multiple requirements of Boolean operations are given simultaneously, then they can be processed together by putting them to the requirement queue in the initialization of the expansion phase. This technique is expected to extend the vector length of both the expansion phase and the reduction phase, which will improve vector acceleration ratio.

In the case of such application as construction of an SBDD for a given Boolean formula or a given circuit description, multiple operations can be evaluated together whose maximum levels in the parse tree or in the circuit diagram are the same[KC90].

Circuit	Circuit size			#node	CPU time [msec]		Acceleration Ratio
	In.	Out.	Nets.		Scalar (S)	Vector (V)	(S/V)
c432	36	7	196	104,066	9,099	412	22.09
c499	41	32	243	65,671	2,585	163	15.86
c880	60	26	438	31,378	2,057	221	9.31
c1355	41	32	587	208,324	5,886	407	14.46
c1908	33	25	913	60,850	3,038	375	8.10
c3540	50	22	1719	1,029,210	74,834	2,692	27.80
c5315	178	123	2485	48,353	5,151	970	5.31

Table 3.1: Experimental results

3.4 Implementation and Evaluation

3.4.1 Implementation

We implemented an SBDD manipulator based on the proposed algorithm (including output inverters) on the vector supercomputer HITAC S-820/80 at the University of Tokyo. The program is coded in Fortran77. Almost all inner DO loops of the program are vectorized.

The required storage is 7 words (28 bytes) for a permanent node ('0' edge, '1' edge and level etc. and the space for the node-table and the operation-result-table) and the additional required temporary storage for a temporary node is 5 words (20 bytes). Since we can use up to 256 megabyte main memory on the HITAC S-820/80 at the University of Tokyo, we can manage an SBDD of more than 5 million nodes.

3.4.2 Evaluation

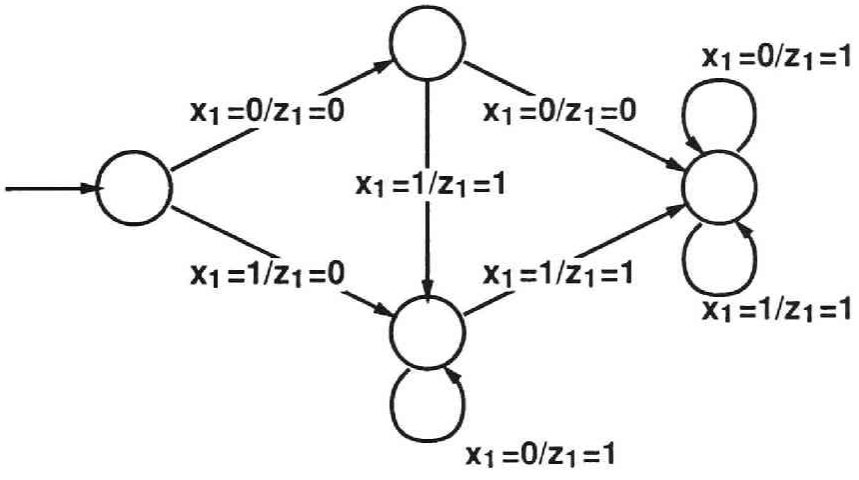
Table 3.1 shows the benchmark results on the S-820/80. This table shows the required CPU time for constructing the graph representing the Boolean functions of all primary outputs and all nets from a circuit description. For example, an SBDD for the circuit 'c432' represents 203 Boolean functions (7 primary outputs and 196 nets). The benchmark circuits are chosen from ones in ISCAS'85 [BF85]. For the ordering of the variables, the dynamic weight assignment method[MIY90] is employed (the computation time for the ordering is not contained in Table 3.1). The *vector execution time* (V) is the required CPU time using all features of the S-820/80, while the *scalar execution time* (S) is the required CPU time using only the conventional scalar processing unit of the S-820/80. A source program which is tuned for vectorization is used for both scalar execution and vector execution (i. e., two object codes with and without vector instructions are generated by the Fortran 77 compiler of HITAC S-820/80). The *vector acceleration ratio* (S/V) is the ratio of the scalar execution time to the vector execution time. *#node* is the number of the nodes of the SBDD representing the Boolean functions of all primary outputs and all the nets.

From Table 3.1, we can see that 5.3 to 27.8 vector acceleration ratio is gained. These results show how the program is suited for the vector supercomputer. Especially, the circuits with large number of nodes and small number of variables, such as c432, c499, c1355 and c3540, are highly accelerated. This is because the width of the SBDDs of such circuits are very large, i. e., there are many nodes in every level of the graph, and the vector length is very long on the average when such a graph is processed.

Compared with the results on the workstation Sun3/60 by Minato et al.[MIY90], our results are 130 times faster in the best case. For example, only 0.163 sec. and 0.407 sec. are required for c499 and c1355 respectively

$$\theta = AG(p_{x_1} \Rightarrow (AX(AXp_{z_1})))$$

(a) An Example of CTL Formula



(b) An Example of Sequential Machine

Figure 3.8: An Example of CTL Formula and Sequential Machine

in Table 3.1, while 21.5 sec. and 51.4 sec., respectively, were required in [MIY90] .

3.5 Application for CTL Model Checker

As an example of applications of the developed Boolean function manipulator, a computation tree logic (CTL) model checker is implemented on the vector supercomputer HITAC S-820/80.

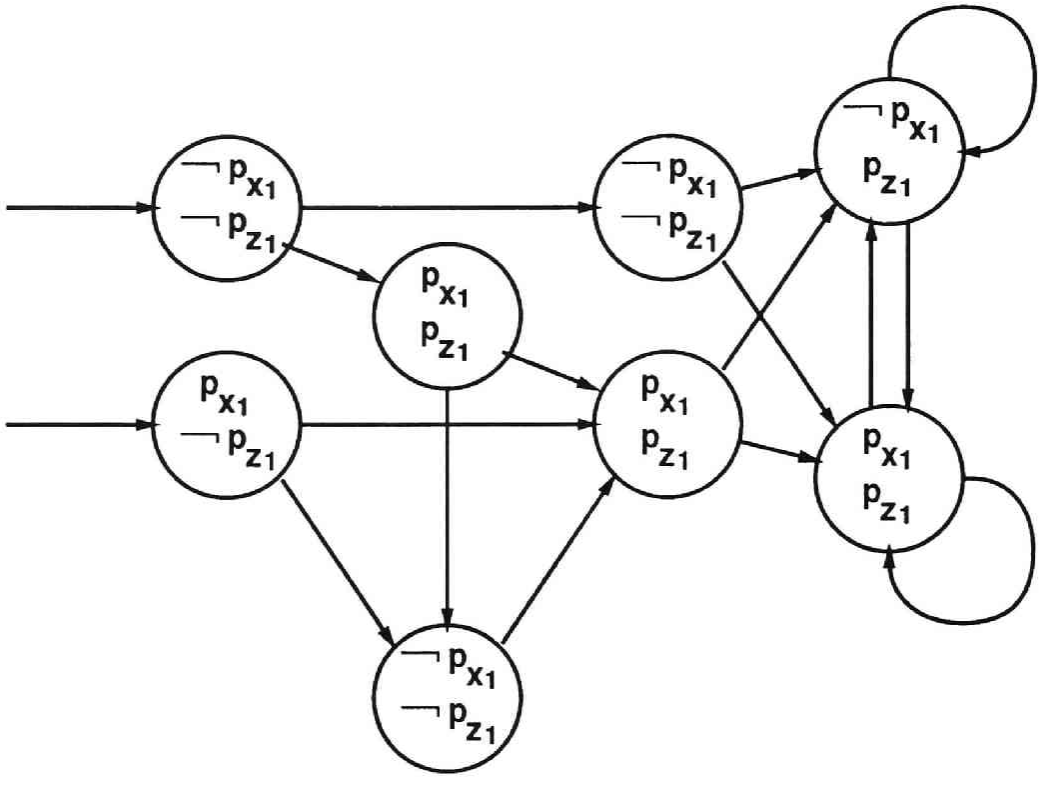


Figure 3.9: An Example of Kripke Structure

3.5.1 Outline

CTL model checking is a formal method for design verification of finite state machines such as sequential machines. Input for the model checker is a CTL formula expressing the specification of the sequential machine and a designed sequential machine. The model checker verifies whether a designed sequential machine satisfies the specification or not by computing the truth-value of the CTL formula at the initial states of the Kripke structure corresponding to the designed sequential machine.

Figure 3.8 illustrates an example of the CTL model checking. Figure 3.8 (a) is a CTL formula given as the specification of a sequential machine. Figure 3.8 (b) is a state transition diagram of the given designed sequential machine. A directed graph Figure 3.9 represents the Kripke structure corresponding to the sequential machine of Figure 3.8 (b). Every node of Figure 3.9 representing a state of the Kripke structure corresponds to an edge of Figure 3.8 (b). The truth-value of a CTL formula is determined by a state of a Kripke structure. The CTL model checker computes the truth-values of the CTL formula at the initial states of the Kripke structure. If all these truth-values are **true** then the designed sequential machine satisfies the specification.

CTL model checker has been implemented on workstations using Boolean function manipulators based on SBDDs and its efficiency has been reported [BCMD90]. In this chapter, the implementation of the CTL model checker using the vectorized Boolean function manipulator is discussed and the experimental results are shown.

3.5.2 Computational Tree Logic

Computational Tree Logic (CTL)[CES83] is a temporal logic. Let AP be a set of atomic propositions. CTL formulas are inductively defined as follows:

- If $p \in AP$, then p is a CTL formula.
- If η is a CTL formula, then so are $\neg\eta$, $EX\eta$ and $EG\eta$.
- If η and ξ are CTL formulas, then so are $\eta \vee \xi$ and $E[\eta \mathcal{U} \xi]$.

The semantics of CTL is defined over a *Kripke structure* $K = (S, R, I)$, where

- S is a non-empty finite set of states.

- $R \subseteq S \times S$ is a total binary relation on S (i. e., for $\forall \mathbf{s} \in S$, there exists $\mathbf{s}' \in S$ such that $(\mathbf{s}, \mathbf{s}') \in R$).
- $I : S \rightarrow 2^{AP}$ is an interpretation function which labels each state with a set of atomic propositions **true** at that state.

An infinite sequence of states $\pi = \mathbf{s_0 s_1 s_2} \ldots$ is called a *path* from $\mathbf{s_0}$ if $(\mathbf{s_i}, \mathbf{s_{i+1}}) \in R$ for $\forall i \geq 0$. $\pi(i)$ denotes the $i-$th state of the sequence π (i. e., $\pi(i) = \mathbf{s_i}$).

The truth-value of a CTL formula is determined by a state of a Kripke structure and $K, \mathbf{s} \models \eta$ denotes that a CTL formula η holds at a state $\mathbf{s}$ of a Kripke structure K. If there is no ambiguity, we will omit K and just write as $\mathbf{s} \models \eta$. The relation $\models$ is recursively defined as follows:

- $\mathbf{s} \models p \ (\in AP)$ iff $p \in I(\mathbf{s})$.
- $\mathbf{s} \models \neg\eta$ iff $\mathbf{s} \not\models \eta$.
- $\mathbf{s} \models \eta \vee \xi$ iff $\mathbf{s} \models \eta$ or $\mathbf{s} \models \xi$.
- $\mathbf{s} \models EX\eta$ iff there exists some next state $\mathbf{s}'$ of $\mathbf{s}$ (i. e., $(\mathbf{s}, \mathbf{s}') \in R$) such that $\mathbf{s}' \models \eta$.
- $\mathbf{s} \models EG\eta$ iff there exists some path π on K starting from the state $\mathbf{s}$ such that $\pi(i) \models \eta$ for $\forall i \geq 0$.
- $\mathbf{s} \models E[\eta \mathcal{U} \xi]$ iff there exists some path π on K starting from the state $\mathbf{s}$ such that $\exists i \geq 0$, $\pi(i) \models \xi$ and $\pi(j) \models \eta$ for $0 \leq \forall j < i$.

In addition to other Boolean operators such as conjunction ($\wedge$), the following abbreviations are often used:

- $AX\eta = \neg EX\neg\eta$.
- $A[\eta \mathcal{U} \xi] = \neg(EG(\eta \wedge \neg\xi) \vee E[(\eta \wedge \neg\xi) \mathcal{U} (\neg\eta \wedge \neg\xi)])$.

- $AF\eta = \neg EG\neg\eta$.
- $EF\eta = E[true\,\mathcal{U}\eta]$.
- $AG\eta = \neg EF\neg\eta$.

Intuitively, the first letters of temporal operators, A and E, represent universal and existential path quantifier, respectively. The other letters of temporal operators represents:

- $X\eta$ represents that η holds at the next state;
- $G\eta$ represents that η holds at every state on the path;
- $F\eta$ represents that η holds at some state on the path;
- $\eta\mathcal{U}\xi$ represents that ξ holds at some state and η holds always before that state on the path.

3.5.3 Sequential Machines

A subset S of B^n is represented by a characteristic function F such that $F(\mathbf{s}) = \mathbf{1}$ if and only if $\mathbf{s} \in \mathbf{S}$.

Let $x_i(1 \leq i \leq l)$, $y_j(1 \leq j \leq m)$ be input variables and state variables, respectively. Let $\mathbf{x}$ and $\mathbf{y}$ be vectors $x_1, x_2, \ldots x_l$ and $y_1, y_2, \ldots y_m$, respectively.

A sequential machine with l inputs, m state variables and n outputs are given in the form of Boolean functions as follows:

- Transition functions:
  $f_j(\mathbf{x}, \mathbf{y})$ $(1 \leq j \leq m)$
- Output functions:
  $z_k(\mathbf{y})$ $(1 \leq k \leq n)$ for Moore-type machines
  $z_k(\mathbf{x}, \mathbf{y})$ $(1 \leq k \leq n)$ for Mealy-type machines

In order to associate inputs and outputs of a sequential machine with atomic propositions of the Kripke structure, p_{x_i} $(1 \leq i \leq l)$ and p_{z_k} $(1 \leq k \leq n)$ are used as atomic propositions corresponding to x_i and z_k respectively. $x_i = 1$ corresponds to $p_{x_i} = true$ and so on. In addition, Boolean function $F_{init}(\mathbf{y})$ is introduced in order to represent the set of initial states of the sequential machine, i. e., $F_{init}(\mathbf{y})$=1 iff state vector $\mathbf{y}$ corresponds to an initial state.

3.5.4 Basic Algorithm

The algorithm shown in this section is based on [BCMD90].

Since the semantics of CTL is defined over Kripke structures, a given sequential machine has to be transformed to a Kripke structure for model checking.

The set of states of a Kripke structure is interpreted as the set of edges of the state transition diagram of the sequential machine.

Let $\mathbf{s}$ be $\mathbf{x}\#\mathbf{y}$, a concatenation of two vectors $\mathbf{x}$ and $\mathbf{y}$. By introducing new vectors of variables $\mathbf{x}' = x'_1, x'_2, \ldots x'_l$ and $\mathbf{y}' = y'_1, y'_2, \ldots y'_m$ corresponding to $\mathbf{x}$ and $\mathbf{y}$, $\mathbf{s}'$ is defined to be $\mathbf{x}'\#\mathbf{y}'$.

The Kripke structure K is represented by the following Boolean function F_K:

$$F_K(\mathbf{s}', \mathbf{s}) = \prod_{0 \leq j \leq m} \text{EQUIV}(y'_j, f_j(\mathbf{x}, \mathbf{y}))$$

This function means that $F_K(\mathbf{s}', \mathbf{s}) = 1$ if and only if $(\mathbf{s}, \mathbf{s}')$ is an edge of the Kripke structure obtained from the sequential machines.

$\exists x_i.f$ is defined to be $\text{OR}(f(x_i = 0), f(x_i = 1))$. $\exists \mathbf{x}.f$ is defined to be $\exists x_1.\exists x_2. \ldots .\exists x_l.f$. $\forall \mathbf{x}.f$ is defined similarly.

[Algorithm of CTL Model Checker]

- Input: a CTL formula θ and Boolean functions representing the sequential machine, f_j $(1 \leq j \leq m)$, z_k $(1 \leq k \leq n)$ and F_{init}.

- Output: *good* if the sequential machine satisfies θ and *bad* otherwise.

(1) Construct a Boolean function representing a Kripke structure F_K. Set $F_{p_{x_j}}(\mathbf{s}) = x_j$ and $F_{p_{z_k}}(\mathbf{s}) = z_k$.

(2) For each subformula θ_i of θ, compute a Boolean function F_{θ_i} representing the set of states where θ_i holds as follows. The algorithm runs in bottom up manner and finally computes F_θ:

(a) If θ_i is an atomic proposition, then return F_{θ_i}.

(b) If θ_i is $\neg\eta$, then return $\mathrm{NOT}(F_\eta(\mathbf{s}))$.

(c) If θ_i is $\eta \vee \xi$, then return $\mathrm{OR}(F_\eta(\mathbf{s}), F_\xi(\mathbf{s}))$.

(d) If θ_i is $EX\eta$, then return $F_{EX\eta}$ as follows:
$F_{EX\eta}(\mathbf{s}) = \exists \mathbf{s}'.\ \mathrm{AND}(F_\eta(\mathbf{s}'), F_K(\mathbf{s}', \mathbf{s}))$

(e) If θ_i is $EG\eta$, then return $F_{EG\eta}$ which is obtained as the fixed point of the following sequence of functions, $A_0, A_1, \ldots$:

$$\begin{aligned} A_0(\mathbf{s}) &= F_\eta(\mathbf{s}) \\ A_{i+1}(\mathbf{s}) &= \mathrm{AND}(A_i(\mathbf{s}), \exists \mathbf{s}'.\mathrm{AND}(A_i(\mathbf{s}'), F_K(\mathbf{s}', \mathbf{s}))) \end{aligned}$$

(f) If θ_i is $E[\eta \mathcal{U} \xi]$, then return $F_{E[\eta \mathcal{U} \xi]}$ which is obtained as the fixed point of the following sequence of functions, $A_0, A_1, \ldots$:

$$\begin{aligned} A_0(\mathbf{s}) &= F_\xi(\mathbf{s}) \\ A_{i+1}(\mathbf{s}) &= \mathrm{OR}(A_i(\mathbf{s}), \\ &\quad \exists \mathbf{s}'.\mathrm{AND}(A_i(\mathbf{s}'), \mathrm{AND}(F_\eta(\mathbf{s}), F_K(\mathbf{s}', \mathbf{s})))) \end{aligned}$$

(3) If $\forall \mathbf{y}.\ \mathrm{OR}(\mathrm{NOT}(F_{init}), F_\theta) = true$ then return *good* else return *bad*.

3.5.5 Implicit Manipulation of Kripke Structure

Basic idea of this section is in [HHY92].

The size of an SBDD representing $F_K(\mathbf{s}', \mathbf{s}) = \prod_{1 \le j \le m} \text{EQUIV}(y'_j, f_j(\mathbf{x}, \mathbf{y}))$ can be very large, even if the total size for f_j is small. By the following computation, the construction of SBDD representing F_K can be avoided.

The Boolean function $F_K(\mathbf{s}', \mathbf{s})$ is used only in the form: $\exists \mathbf{s}'.\ \text{AND}(C(\mathbf{s}'), F_K(\mathbf{s}', \mathbf{s}))$. This function can be calculated as follows:

[Algorithm for $\exists \mathbf{s}'.\text{AND}(C(\mathbf{s}'),\ F_K(\mathbf{s}', \mathbf{s}))$]
Obtain the $(m+1)$ functions D_i $(i = 0, \ldots, m)$ defined as follows. Return $D_m(\mathbf{s})$ as $\exists \mathbf{s}'.\text{AND}(C(\mathbf{s}'), F_K(\mathbf{s}', \mathbf{s}))$:

$$
\begin{aligned}
D_0(\mathbf{s}', \mathbf{s}) &= \exists \mathbf{x}'.C(\mathbf{s}') \\
D_{i+1}(y'_{i+2}, y'_{i+3}, \ldots, y'_m, \mathbf{s}) &= \text{OR}(\text{AND}(D_i(y'_{i+1} = 1), f_{i+1}(\mathbf{s})), \\
&\quad \text{AND}(D_i(y'_{i+1} = 0), \text{NOT}(f_{i+1}(\mathbf{s})))) \\
&\quad \text{for } i = 0, 1, \ldots, m-1
\end{aligned}
$$

Note that if D_i is independent of y'_{i+1}, then $D_{i+1} = D_i$. Whether D_i depends on y'_{i+1} or not can be tested by checking equivalence of D_i and $D_i(y'_{i+1} = 0)$. Using this technique, large part of computation can be reduced.

3.5.6 Implementation and Evaluation

The CTL model checker based on the above method was implemented on the vector supercomputer HITAC S-820/80 at the University of Tokyo. This program utilizes the Boolean function manipulator proposed in this chapter. An SBDD representing the Boolean function $F(\mathbf{s}')$ is obtained from an SBDD representing $F(\mathbf{s})$ using the operation shift of variables. The program is coded in Fortran 77.

The benchmark results are shown in Table 3.2. This table shows the required CPU time for model checking. The *vector execution time* (V), the *scalar execution time* (S) and the *vector acceleration ratio* (S/V) are

Sequential Machine	#node of Sequential Machine	CPU time [sec]		Acceleration Ratio (S/V)
		Scalar (S)	Vector (V)	
padd2	186	4.25	0.78	5.44
padd4	359	12.70	1.91	6.66
padd8	741	55.77	6.74	8.27
padd12	1,171	155.00	16.53	9.38
padd16	1,649	386.66	36.09	10.71
calu2	628	18.48	1.83	10.08
calu4	1,220	94.50	6.28	15.05
calu8	2,476	804.52	37.67	21.36
calu12	3,828	—	157.40	—
calu16	5,276	—	665.02	—

Table 3.2: Experimental results

defined in section 3.4.2. The *#node of sequential machine* is the number of the nodes of SBDD representing the designed sequential machine. Note that the maximum number of nodes required in the process of the model checking is much greater than #node of sequential machine. The sequential machines used for the benchmarks are pipelined CPU's. All results of the benchmarks are *good*, i. e., every sequential machine satisfies the specification. Experiments of the scalar execution for benchmarks 'calu12' and 'calu16' was not performed because these jobs seemed to exceed the time limit of batch jobs of the University of Tokyo (3,600 sec.).

From Table 3.2, we can see that 5.4 to 21.4 vector acceleration ratio is gained (the vector acceleration ratio of 'calu12' and 'calu16' is expected to be larger). These figures show that the program is suited for the vector supercomputer. In particular, the model checking for the sequential

machines represented by large number of nodes are highly accelerated.

3.6 Conclusion

In this chapter a vector algorithm for manipulating Boolean functions based on SBDDs has been proposed. The proposed algorithm is based on breadth-first manipulation to utilize the high performance of vector supercomputers.

The Boolean function manipulator based on the proposed algorithm is developed on the vector supercomputer HITAC S-820/80 at the University of Tokyo and benchmark results are shown. The vector acceleration ratio on the S-820/80 is 5.3 to 27.8. This manipulator on the S-820/80 is faster than that of Minato et al. on Sun3/60 up to 130 times. As an application, this manipulator is utilized for CTL model checker.

Thus, the developed algorithm has been proven to be suitable for vector supercomputers and the manipulator is proven to be faster than conventional ones. The developed technique for Boolean function manipulation is expected to be utilized for various applications of CAD systems such as design verification, test generation, logic synthesis and so on which support the design of VLSI whose scale and complexity are increasing rapidly.

Furthermore, there are many non-numerical computations other than CAD systems which manipulate Boolean functions as given data or intermediate data. For such applications, Boolean function manipulators based on SBDD may be utilized effectively. The results of this chapter, therefore, also suggests that the vector supercomputers can be utilized for various non-numerical computations using SBDDs.

Chapter 4

Algorithms for Manipulating Binary-Decision Diagrams in Secondary Memory

4.1 Introduction

As described in the previous chapter, Ordered Binary-Decision Diagrams (OBDDs), or simply Binary-Decision Diagrams (BDDs), are excellent graph representation of Boolean functions[Ake78, Bry86]. Efficient Boolean function manipulators based on the Shared BDD (SBDD, a multirooted BDD) representation have been developed[MIY90, BRB90], and they are widely used in various applications in Computer-Aided Design (CAD) of digital systems.

At present, SBDD manipulators are, in most cases, implemented on workstations. The recent progress in VLSI technologies requires them to manipulate larger-scale Boolean functions. The maximum size of the SBDDs which can be manipulated on workstations is limited by both required time and required memory. In order to reduce the computation time, the use of parallel machines or connection machines[KC90] or the use of vector supercomputers has been proposed. However, yet in many

applications we have to give up to design large-scale circuits due to the limitation of the size of main memory to store SBDDs rather than the computation time. In order to reduce the size of SBDDs, attributed edges have been proposed[MIY90, BRB90]. Variable ordering has been also studied by many researchers.

In this chapter, the use of secondary memory, such as hard disk drives of workstations, is considered in order to manipulate very large SBDDs which is too large to be stored within main memory. In contrast that the conventional depth-first algorithm causes random access of memory, the proposed method is intended to cause sequential access of memory. The main idea of our method is level-by-level manipulation of Shared Quasi-reduced BDDs (SQBDDs) upon a breadth-first algorithm. A set of nodes and hash tables of one level are recalled from secondary memory in one lot, then operations for the nodes of the level are performed within main memory, and the results of the operations for the level are stored to secondary memory all together. This algorithm is effective to reduce the overhead due to access of secondary memory, because it requires much fewer times to access secondary memory; every time a large data block is transferred between main memory and secondary memory. In addition, a garbage collection algorithm based on sliding type compaction is introduced to reduce page faults in succeeding manipulation.

We implemented and evaluated the proposed method on a workstation Sun SPARC Station 10 with 64 megabyte main memory and a one gigabyte hard disk drive connected via SCSI-2 standard interface. More than 50 million nodes can be allocated within one gigabyte virtual memory space, and as a result an SQBDD with more than 12 million nodes representing all primary outputs of a 15-bit multiplier was constructed from a circuit description in about 5.6 hours. If the conventional SBDD manipulator were used instead, it is estimated that it would take about

1,900 hours, so we can say that our manipulator achieved about 330 times improvement in elapsed time. Furthermore, we made experiments using semiconductor extended storage instead of hard disk, and showed that the required time for the 15-bit multiplier is reduced to about 2.2 hours.

In the following section, basic explanation on secondary memory and inefficiency in using it for depth-first SBDD manipulator will be described. See also section 2 of the previous chapter for basic explanation on SBDDs. In section 3, a new method will be proposed. In section 4, experimental results of the Boolean function manipulator will be shown. Section 5 will provide some concluding remarks.

4.2 Preliminaries

4.2.1 Secondary Memory

Today, almost all general purpose computers have secondary memory which have much larger capacity than main memory. In this paper, we assume the following secondary memory devices for workstations and show experimental results with them.

Hard disk is one of magnetic memory devices. The transfer rate of the hard disk drive used in experiments of section 4.4 is 10 megabytes per second via SCSI-2 ANSI standard interface. The average access time of the hard disk drive is 10 milliseconds. *Semiconductor extended storage* is a secondary memory made of semiconductor memory devices, such as DRAM's. The transfer rate of the semiconductor extended memory unit used in section 4.4 is 10 megabytes per second via SCSI-2 interface. The average access time of the semiconductor extended storage unit is 0.3 milliseconds.

No matter which device is used, every transfer between main memory and secondary memory is performed by a block transfer of contiguous

space. The minimum unit of transfer is called *page*. The page size defined in the O. S. we will use in section 4.4 is 4096 bytes. In order to utilize the maximum transfer rate, every transferred page should be filled with actually used data.

4.2.2 Problems in the Use of Secondary Memory with Depth-First Algorithm

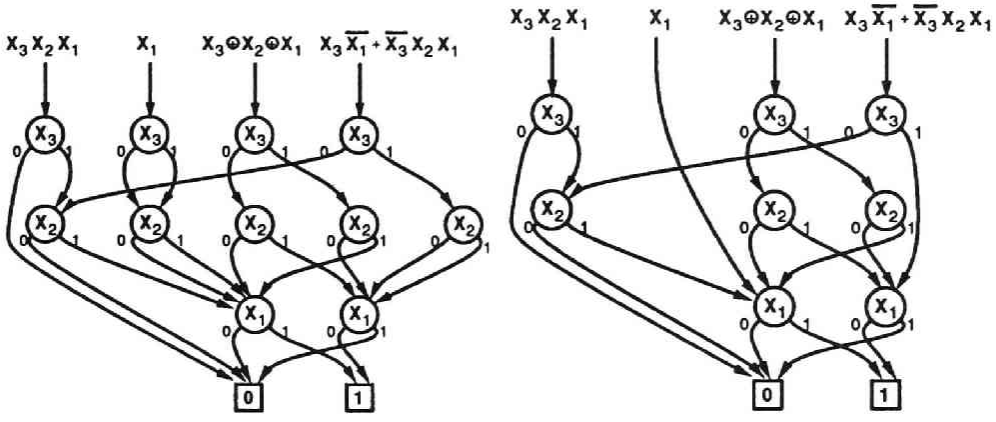
Now let us consider the use of secondary memory to enable manipulating very large SBDDs which are too large to store within main memory.

As mentioned in section 3.2.2, the conventional depth-first algorithm for manipulating SBDDs is widely used on workstations. On the memory access during the depth-first manipulation, following can be said;

- Access to nodes causes random access in some cases. In order to avoid random access during the depth-first traversals of an SBDD, adjacent nodes should be placed in neighborhood in memory space. However, it is impossible if there are many nodes which have large indegree.

- Access to the operation-result-table and the node-table causes random access, because they are hash tables.

As mentioned in section 4.2.1, a transfer between main memory and secondary memory is performed by a block transfer of contiguous space. Even if the required data in a page is only one node (about 20 bytes) or only one entry of the operation-result-table (about 12 bytes), a whole page is transferred, and the transfer time for the whole page (e. g. 4096 bytes) is required. It follows that secondary memory is not suitable to store BDD when depth-first algorithm is employed.



(a) A Shared Quasi-Reduced BDD (b) A Shared Binary-Decision Diagram

Figure 4.1: A Shared Quasi-reduced BDD (SQBDD) and SBDD

4.3 Breadth-First Algorithm for Manipulating SBDDs in Secondary Memory

4.3.1 Outline of the Proposed Method

In this section, we propose an efficient method for manipulating very large SBDDs in secondary memory. The proposed method is based on the breadth-first algorithm for manipulating diagrams level-by-level and the data structure which explicitly classifies data according to levels. The set of nodes of a level is stored in a contiguous space of secondary memory. The operation-result-table and the node-table are also constructed for every level.

To enable level-by-level manipulation, let us introduce *Shared Quasi-reduced BDDs (SQBDDs).* An SQBDD is a representation of Boolean functions using an acyclic directed graph. An example of an SQBDD is shown in Figure 4.1 (a). This graph represents four Boolean functions

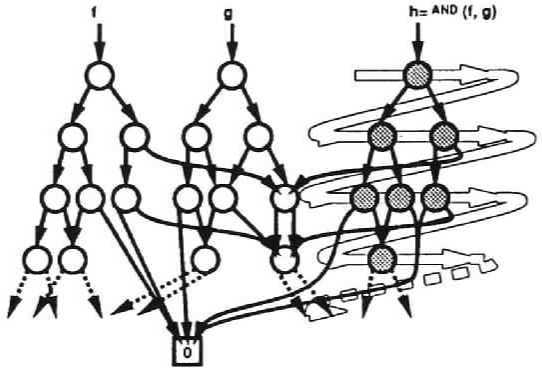


Figure 4.2: Expansion Phase of the Breadth-First Algorithm for Manipulating SQBDDs

as represented by an SBDD in Figure 4.1 (b). There is no non-unique node, but there are some redundant nodes in SQBDD. SQBDDs have redundant nodes so as to hold the following property;

- Every '0' edge and '1' edge of a level i node points to either a level $(i-1)$ node or a leaf-node.

- Root-nodes which are externally referred to by users have the common level, called *level_max*, except the root-nodes which represent 0 or 1.

Note that there is no *pseudo-leaf-node* in SQBDDs, where pseudo-leaf-node is a redundant node whose '0' edge and '1' edge point to the same leaf-node.

SQBDDs have the same excellent properties as SBDDs. SQBDDs are canonical, and small in size for many practical Boolean functions.

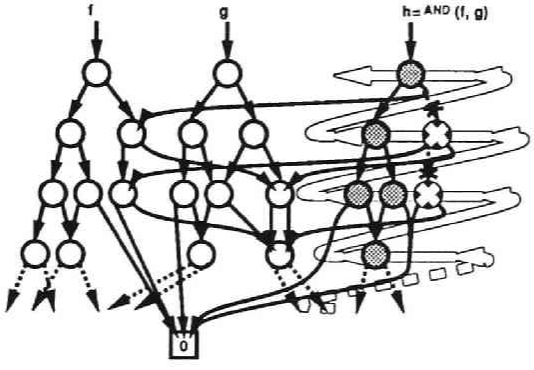


Figure 4.3: Reduction Phase of the Breadth-First Algorithm for Manipulating SQBDDs

4.3.2 Algorithm

As described in the previous chapter, the breadth-first algorithm consists of two phases; an *expansion phase* and a *reduction phase*. In the expansion phase, temporary nodes that are sufficient to represent the resultant function are generated in a breadth-first manner from the root-node toward the leaf-nodes (Figure 4.2). In the reduction phase, the temporary nodes generated in the expansion phase are checked and the redundant nodes and the non-unique nodes are removed in a breadth-first manner from the nodes nearby the leaf-nodes toward the root-node (Figure 4.3).

We will show the algorithm which is modified for manipulating SQBDDs.

The Expansion Phase

An input for the expansion phase is a requirement, which is a triple (op, e_f, e_g), where op is a Boolean operator to be executed, and e_f and e_g are root-edges of argument Boolean functions represented by an SQBDD. A

requirement $(op,\ e_f,\ e_g)$ requires to compute the root-edge of the resultant function of $op(f, g)$. During processing a requirement of level i, new requirements of level $(i-1)$ are generated for computing the operations between sub-functions of the argument functions. Similar to the previous chapter, we introduce a requirement queue in order to manage these requirements, making our procedure breadth-first, and the nodes generated in the expansion phase are called temporary nodes, while the nodes which already exist are called permanent nodes.

The following procedure is executed in the expansion phase. Initially, a requirement queue is empty, and there is no temporary node.

[The Expansion Phase of the Breadth-First Algorithm]

(1) Put the given requirement $(op,\ e_f,\ e_g)$ to the requirement queue.

(2) $lev = level\_max$

(3) Execute one of (3.1), (3.2) or (3.3) for every requirement of level lev in the requirement queue.

(3.1) If the root-node representing the result of $op(f, g)$ is found trivially, then attach the edge pointing to the node as the result of the requirement.

(3.2) If the root-node representing the result of $op(f, g)$ is found in the operation-result-table, attach the edge that is found in the table as the result of the requirement.

(3.3) Otherwise, generate a new temporary node of level lev and attach the edge pointing to the temporary node as the result of the requirement. At the same time, register the edge pointing to the temporary node to the operation-result-table as the result of $op(f, g)$ and put new requirements of level $(lev-1)$, $(op,\ e_{f_0},$

e_{g_0}) and (op, e_{f_1}, e_{g_1}), whose result will be '0' edge and '1' edge, respectively, of this temporary node, to the requirement queue.

(4) $lev = lev - 1$

(5) If the requirement queue is not empty, then go to (3).

The Reduction Phase

After the expansion phase is completed, there may be temporary nodes which are pseudo-leaf-nodes or non-unique nodes. The main tasks in the reduction phase are to find and remove such nodes. In addition, temporary nodes that are neither a pseudo-leaf-node nor a non-unique node are registered to the node-table. In this algorithm, these tasks are executed in a breadth-first manner from the nodes nearby the leaf-nodes toward the root-node.

When a pseudo-leaf-node or a non-unique node of level i is removed, a *forwarding pointer* is set to indicate the node that takes the place of the removed node. When the '0' edge or '1' edge of a temporary node of level $(i+1)$ points to a removed node of level i, the edge is redirected to point to the node pointed to by the forwarding pointer of the removed node before checking whether the temporary node of level $(i+1)$ is neither a pseudo-leaf-node nor a non-unique node. Forwarding pointers are stored in the array which were used as the requirement queue in the expansion phase.

The reduction phase is formalized as follows;

[The Reduction Phase of the Breadth-First Algorithm]

(1) $lev = 1$

(2) Execute (2.1) - (2.4) for every temporary nodes of the level lev.

(2.1) If its '0' edge or '1' edge points to a removed node, modify the edge so as to point to the node pointed to by the forwarding pointer of the removed node.

(2.2) If its '0' edge and '1' edge point to the same leaf-node, remove the node, and set its forwarding pointer to point to the leaf-node.

(2.3) If there is an equivalent node registered in the node-table, remove the temporary node, and set its forwarding pointer to point to the node found in the node-table.

(2.4) Otherwise, register the node to the node-table, and change the attribute of the node to "permanent" from "temporary".

(3) $lev = lev + 1$

(4) If $lev \leq level\_max$, then go to (2).

4.3.3 Data Structure

The above algorithm is effective for SQBDDs stored in secondary memory if all the nodes of every level are stored together in a contiguous location in secondary memory. Requirements of level i can be solved in the expansion phase only if the nodes of level i and level $(i-1)$ are in main memory. Temporary nodes of level i can be checked in the reduction phase only if nodes of level i and level $(i-1)$ are in main memory. In addition, if the two hash tables are split up according to level, we need only one operation-result-table at a time during the expansion phase and only one node-table at a time during the reduction phase, and tables of other levels can be swapped out to secondary memory.

The allocated secondary memory space for the set of nodes of every level includes free nodes for generating new temporary nodes. While there are free nodes of a level, the size of an array of the set of nodes of

the level do not change, so the array of the level is stored again in the same location of the secondary memory space as they were. If there is no free nodes in the allocated memory space for the level when a new temporary node should be generated, garbage collection (see section 3.4) is performed. If there are few nodes to be recycled anymore, then the total number of the nodes of the level is increased by twice by allocating a new location in the secondary memory space to store the new greater array for the level. As described in [BRB90], the necessary size for the operation-result-table and the node-table to keep the efficiency of the operation is 1/4 to the number of nodes. When the total number of the nodes of a level is updated, then the operation-result-table and the node-table of the level are also increased in size, and all elements are re-hashed into the larger tables. This incremental allocation strategy has the following advantages in the use of secondary memory;

- The allocated spaces of secondary memory for levels are proportional to number of nodes of the levels. It optimizes the utilization of the space of the secondary memory.
- The data density, i. e., the number of the actually used nodes per the number of allocated nodes, is kept high during manipulation. It is crucial to keep the data density to reduce the overhead of access of secondary memory (recall section 2.3).

4.3.4 Garbage Collection

Let us consider the implementation of automatic garbage collection for our SQBDD manipulator. Basic idea described in [BRB90] is applied, but some other techniques are also introduced[Coh81]. Each node has a *reference count* of the number of '0' edges and '1' edges that reference it (if the node is not $level\_max$) or the number of user formulas that reference

it (if the node is *level_max*). This count is maintained incrementally. A node with a reference count of 0 is called *dead*.

When a user formula is freed, the reference count of its root-node is decremented. If the renewed reference count of the root-node is 0, the reference counts of its children should be recursively decremented. However, decrementation of the children are not performed immediately in order to avoid extra access of secondary memory. Instead, dead root-nodes are linked to a list, a *dead list*, of *level_max*. Just before the step (3) of the expansion phase of succeeding operation, the reference counts of the nodes of level $(lev-1)$ which are referenced by the nodes in dead list of level lev is decremented and those nodes whose reference count become 0 are linked to the dead list of level $(lev-1)$. In this way, reference counts are updated in breadth-first manner during the expansion phase.

When the array of the set of nodes of a level become full, a garbage collection for the level is performed. Garbage collection consists of deleting all entries of the hash tables of the level that reference dead nodes and *compacting* all non-dead nodes of the level in one end of the array. Compaction is effective for reducing page faults in succeeding manipulation of this level. Furthermore, we choose *sliding type compaction*, i. e., non-dead nodes of the level are moved toward one end of the array without changing their linear order. Sliding type compaction keeps those nodes which has been defined in the same expansion phase placed in neighborhood in the array, which seems to be the best way to minimize the random access within a level. Compaction step of the garbage collection includes, of course, redirection of several kinds of pointers. See [Coh81] for more detail on sliding type compaction of garbage collection.

Sliding type compaction is also done in the reduction phase in order to remove pseudo-leaf-nodes and non-unique nodes.

4.4 Implementation and Evaluation

4.4.1 Implementation

We implemented the proposed method in C language on the workstation Sun SPARC Station 10 (36MHz, SunOS 4.1.3) with 64 megabyte main memory. As the secondary memory, hard disk and semiconductor extended storage are employed. Specifications of these devices are described in section 4.2.1. Secondary memory space is allocated as the swap area, which is used as the physical storage of the virtual memory space managed by the OS. We used the secondary memory devices transparently under the memory management system of the OS. This is the easiest implementation of our method.

The required space per a node is 18.25 bytes, including the space for the hash tables. Within one gigabyte virtual memory space, more than 50 million nodes can be allocated.

Minato et al. have proposed several attributed edges, including output inverters, for the purpose of reducing the number of the nodes and/or the time used for the manipulation of SBDDs[MIY90]. We employed the output inverters.

If multiple requirements of Boolean operations are given simultaneously, then they can be processed together by putting them to the requirement queue at the initial step of the expansion phase of the breadth-first algorithm. This technique is effective for parallel implementation of SBDD manipulators, because it extends the parallelism of the process [KC90]. This technique is even more effective for our implementation to use secondary memory, because it reduces the number of cycles of the expansion phase and the reduction phase. We employed this technique.

We chose multipliers as benchmarks of our manipulator in order to demonstrate manipulation of very large SQBDDs which is too large to be

Table 4.1: Bechmark Results of our Manipulator

circuit name	#node			in HD time (sec)		in SS time (sec)	
	#used	#red.	#alloc	elapsed	CPU	elapsed	CPU
mult8	10,800	236	41,392	1.21	1.21	1.22	1.22
mult9	29,851	412	98,862	3.26	3.26	3.26	3.26
mult10	82,369	639	329,004	9.48	9.47	9.49	9.48
mult11	227,655	1,083	869,674	28.69	28.68	28.74	28.69
mult12	626,859	1,870	1,592,104	80.78	80.74	80.87	80.79
mult13	1,697,928	3,089	3,608,358	390.54	284.85	310.79	284.75
mult14	4,599,659	5,312	9,107,236	5,017.75	1,095.14	2,054.35	1,114.37
mult15	12,432,897	10,121	26,924,834	20,185.67	3,487.82	8,190.48	3,730.47

stored in main memory. (Standard benchmarks such as ISCAS'85 circuits (except c6288) are too small in BDD size to use secondary memory if appropriate variable orderings are employed.) An SQBDD which represents the Boolean functions of all primary outputs of an unsigned multiplier is constructed from its circuit description. The employed variable ordering is

$$a_0 \succ b_{n-1} \succ a_1 \succ b_{n-2} \succ a_2 \succ b_{n-3} \succ \cdots \succ a_{n-1} \succ b_0$$

where a's and b's are the multiplicand and the multiplier, respectively, and a_0 and b_0 are the LSB of them. This variable ordering requires relatively small number of nodes during construction of SQBDDs for multipliers among several systematic variable orderings.

4.4.2 Experimental Results

Table 4.1 shows the experimental results of our manipulator with one gigabyte secondary memory. This table shows required CPU time (time spent by user plus time spent by system) and elapsed time for constructing

Table 4.2: Bechmark Results of the Conventional Manipulator

circuit name	#node #used	within MM time (sec) elapsed	CPU	in HD time (sec) elapsed	CPU	in SS time (sec) elapsed	CPU
mult8	10,140	2.93	2.93	1,134.89	62.85	367.56	61.50
mult9	28,833	4.27	4.27	3,417.84	114.24	757.32	96.58
mult10	80,850	8.34	8.33	12,254.06	323.05	2,061.25	193.89
mult11	225,106	21.27	21.24	47,262.49	1,152.08	6,421.09	453.84
mult12	622,221	69.29	69.22	158,026.52	3,882.65	19,271.27	1,257.62
mult13	1,689,752	(unable)		(not tried)		64,732.90	5,356.41

an SQBDD. Note that CPU time does not include idle time spent for waiting for responses from secondary memory devices. The column *#used* shows the number of nodes of the final SQBDD which represents the Boolean functions of all the primary outputs of a multiplier. The column *#red.* shows the number of redundant nodes among them (#used−#red. is the number of nodes of SBDDs). We can see that the number of nodes of an SQBDD is almost the same as the number of nodes of an SBDD. The column *#alloc* shows the number of allocated nodes, i. e., total size of the final arrays for the sets of nodes. The column *in HD* and *in SS* shows the results of the experiments with the hard disk and the semiconductor extended storage, respectively. Elapsed time is almost the same as CPU time up to 12-bit multiplier. In fact, the experiments of up to 12-bit multiplier required no physical I/O. This is because they can be performed within 64 megabyte main memory. This is yet another advantage of our implementation of incremental allocation and the use of virtual memory space managed by OS. From Table 4.1, the elapsed time for generating an SQBDD for a 15-bit multiplier is about 5.6 hours and 2.3 hours using the hard disk and the semiconductor extended storage,

respectively.

Table 4.2 shows the experimental results of the conventional depth-first algorithm. The manipulator used for these experiments is the SBDD manipulator developed by Minato et al. which supports two kinds of attributed edges, output inverters and input inverters[MIY90]. Their SBDD manipulator does not support incremental allocation of memory space; all the array space are allocated at the initialization process. The column *in HD* and *in SS* shows the result using hard disk and semiconductor extended storage, respectively, obtained by allocating (virtual) memory space for 16,777,216 nodes (372 megabytes), that is probably the least 2's power necessary to generate an SBDD for 15-bit multiplier in order to estimate the elapsed time for 15-bit multiplier (In fact, we could not make the experiments for the multiplier of 14-bit or more, because they take too long time). The column *within MM* shows the result obtained by allocating only 24 megabyte memory space. From Table 4.2, the elapsed time for generating an SBDD for a 12-bit multiplier using the conventional depth-first manipulator is almost 2 days when the SBDD is stored in hard disk.

4.4.3 Discussion

Figure 4.4 illustrates the results shown in Table 4.1 and Table 4.2.

By means of the conventional manipulator, elapsed time for a 12-bit multiplier is about 2,300 times greater when the diagram is stored in hard disk than when whole diagram is stored within the main memory. This is unbearable.

From Figure 4.4, we can estimate that it takes about 2,000 seconds for a 15-bit multiplier even when large enough main memory would be available. By means of our manipulator, elapsed time for a 15-bit multiplier using hard disk is only about 10 times greater than the above estimation.

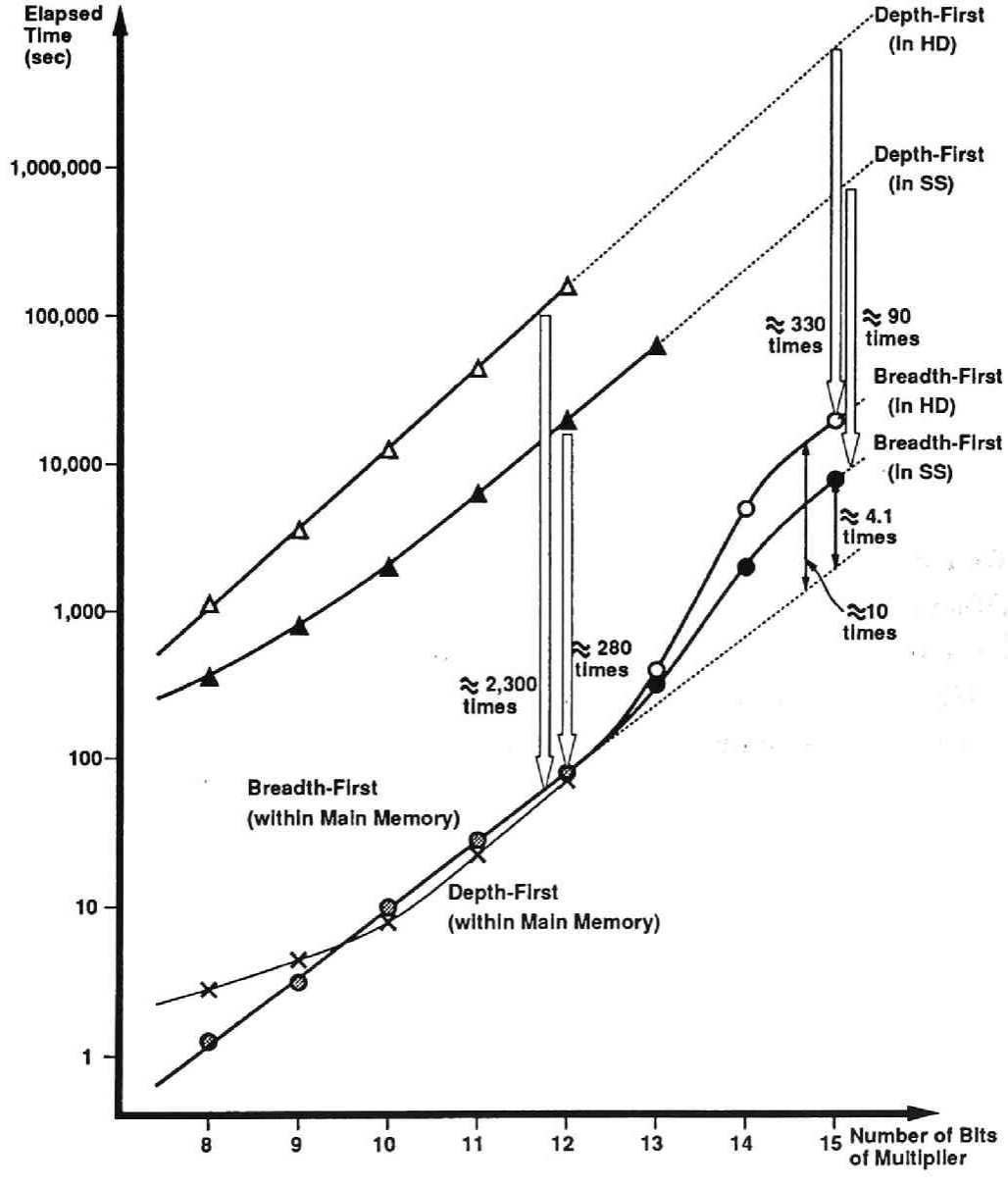


Figure 4.4: Comparison of the Elapsed Time

It is also estimated from Figure 4.4 that it takes more than 2 months for a 15-bit multiplier using the conventional manipulator and hard disk. We can say that manipulator achieved 330 times improvement on elapsed time.

Using semiconductor extended storage, elapsed time of the conventional manipulator for a 12-bit multiplier is improved about 8.2 times compared with using hard disk, but still unbearable. Elapsed time of our manipulator for a 15-bit multiplier is improved about 2.5 times and is only about 4.1 times greater than the estimated elapsed time with large enough main memory.

4.5 Conclusion

We have proposed an efficient method for manipulating very large SQBDDs in secondary memory and shown benchmark results. The developed technique for SQBDD manipulation is expected to be utilized for various CAD applications such as formal design verification, test generation, logic synthesis and so on in order to enable us much larger and more complex design which were not possible with the conventional SBDD manipulators.

Chapter 5

Conclusions

In this thesis, three topics concerning Boolean function manipulation have been discussed in order to solve very large problems in CAD of digital systems.

In chapter 2, high-speed algorithms for generating prime implicants of a given Boolean function were discussed, and the use of vector supercomputer was proposed. The proposed algorithms were based on the consensus expansion. The proposed algorithms were implemented efficiently on vector supercomputers by performing consensus expansion in breadth-first manner, and employing truth table representation of a Boolean function and map representation of a set of prime implicants. Table look-up technique was also employed to reduce the consensus expansion stages. The proposed algorithms were implemented on the vector supercomputer FACOM VP-400E at the Kyoto University Data Processing Center and compared with several other algorithms. For example, by the consensus expansion method with table look-up, all prime implicants of randomly generated 18-variable Boolean functions were generated in about 1.4 seconds on the average.

As an application of the proposed algorithm, we have shown the results related to the number of prime implicants of Boolean functions. We have

shown that the Igarashi's conjecture on the maximum number of prime implicants of n-variable Boolean functions is true for $n = 5$ and 6, i. e., the maximum number of prime implicants of 5- and 6-variable Boolean functions are 32 and 92, respectively. It is still open whether the Igarashi's conjecture is true for $n = 7$ and beyond.

In chapter 3 and chapter 4, algorithms for manipulating SBDDs were discussed in order to manipulate very large SBDDs which cannot be manipulated by conventional workstations, and the use of breadth-first algorithm was proposed. The breadth-first algorithm consists of two parts; an expansion phase and a reduction phase. In the expansion phase, new nodes sufficient to represent the resultant Boolean function are generated in a breadth-first manner from the root-node toward leaf-nodes. In the reduction phase, the nodes generated in the expansion phase are checked in a breadth-first manner from nodes nearby leaf-nodes toward the root-node.

In chapter 3, a high-speed algorithm for manipulating SBDDs which is suitable for vector supercomputers was proposed. Breadth-first algorithm was employed to vectorize manipulation, and actually almost all steps were vectorized, including hash table access which was efficiently vectorized using high-speed vector indirect store instruction of a vector supercomputer HITAC S-820/80. A Boolean function manipulator based on the proposed algorithm was implemented on the HITAC S-820/80 at the University of Tokyo, and experiments of constructing the SBDDs representing the Boolean functions of all the primary outputs and nets from a circuit description chosen from ISCAS'85 [BF85] were performed. From these experiments, the vector acceleration ratio on the S-820/80 was 5.3 to 27.8. Compared with the results on the work station Sun3/60 by Minato et al.[MIY90], our results were up to 130 times faster in the best case. In addition, as an example of applications of SBDDs, a design

verification system based on computation tree logic (CTL) model checker was implemented and the experimental results were shown.

In chapter 4, the use of secondary memory was discussed in order to manipulate SBDDs which were too large to be stored within main memory. In order to avoid random accesses to the secondary memory, level-by-level manipulation of Shared Quasi-reduced BDDs (SQBDDs) upon a breadth-first algorithm was employed. The use of garbage collection with sliding type compaction was also introduced to reduce page faults in succeeding manipulation. A Boolean function manipulator based on the proposed algorithm was implemented and evaluated on the workstation Sun SPARC Station 10 with 64 megabyte main memory and a one gigabyte hard disk drive connected via SCSI-2 standard interface. As a result an SQBDD with more than 12 million nodes representing all the primary outputs of a 15-bit multiplier was constructed from a circuit description in about 5.6 hours. If the conventional SBDD manipulator is used instead, it is estimated that it would take about 1,900 hours, so we can say that our manipulator achieved about 330 times improvement in elapsed time.

The results in chapter 2 and 3 suggests that the use of vector supercomputers is effective not only for numerical problems, but also for logical and combinatorial problems. There are also many non-numerical computations other than CAD systems. Some of the developed algorithms, data structures and techniques seems useful for such applications.

BDDs are now widely utilized in various areas, including design verification, test generation and logic synthesis for VLSI CAD, truth maintenance system of artificial intelligence, and some other mathematical problems. BDD-based prime implicants generation has been also studied [CM92]. The developed Boolean function manipulators are expected to be used for various BDD applications. Discovering new application

areas of BDDs and improving the performance of Boolean function manipulators will provide a fruitful area of research for many years to come. Especially, studies on computer architecture for Boolean function manipulation seem challenging; memory architecture and/or communication of processors are and will be the central problems. The proposed breadth-first algorithm for manipulating SBDDs seems useful for large and complex problems of Boolean functions to solve.

References

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, C-27(6):509–516, June 1978.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 46–51, June 1990.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational circuits, special session on ATPG and fault simulation. In *Proc. 1985 IEEE International Symposium on Circuit and Systems*, Kyoto, Japan, June 1985.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, August 1986.

[CB89] K. Cho and R. E. Bryant. Test pattern generation for sequential MOS circuits by symbolic fault simulation. In *Proc. 26th ACM/IEEE Design Automation Conference*, pages 418–423, June 1989.

ing for branching time temporal logic. In *Proc. Synthesis and Simulation Meeting and International Interchange 1992*, pages 243–252, April 1992.

[IDY90] N. Ishiura, Y. Deguchi, and S. Yajima. Coded time-symbolic simulation using shared binary decision diagram. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 130–135, June 1990.

[Iga79] Y. Igarashi. An improved lower bound on the maximum number of prime implicants. *Trans. IECEJ*, E62(6):389–394, June 1979.

[IYY87] N. Ishiura, H. Yasuura, and S. Yajima. High-speed logic simulation on vector processors. *IEEE Trans. Computer-Aided Design*, CAD-6(3):305–321, May 1987.

[Kag87] T. Kagatani. Vector algorithms for generating prime implicants of logic functions. Master's thesis, Department of Information Science, Faculty of Engineering, Kyoto University, Japan, February 1987.

[KC90] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. 1990 IEEE International Conference on Computer Design*, pages 220–223, September 1990.

[KOY79] Y. Kambayashi, K. Okada, and S. Yajima. Prime implicant generation of logic functions using clause selection method. *Trans. IECEJ*, J62-D(2):89–96, February 1979. (in Japanese).

[McC56] E. J. McCluskey, Jr. Minimization of Boolean functions. *Bell Syst. Tech. J.*, 35(6):1417–1444, November 1956.

[MIY90] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 52–57, June 1990.

[Mor70] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Trans. Comput.*, C-19(6):504–509, June 1970.

[MP64] F. Mileto and G. Putzolu. Average values of quantities appearing in Boolean function minimization. *IEEE Trans. Electron. Comput.*, EC-13:87–92, April 1964.

[Nel54] R. J. Nelson. Simplest normal truth functions. *J. Symbolic Logic*, 20(2):105–108, June 1954.

[Pet60] S. R. Petrick. On the minimization of Boolean functions. In *Proc. International Conference on Information Processing, 1959*, pages 422–423, Germany, 1960. Oldenbourg.

[Qui55] W. V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62:627–631, November 1955.

[SCL70] J. R. Slagle, C. L. Chang, and R. C. T. Lee. A new algorithm for generating prime implicants. *IEEE Trans. Comput.*, C-19(4):304–310, April 1970.

[SYMF90] H. Sato, Y. Yasue, Y. Matsunaga, and M. Fujita. Boolean resubstitution with permissible functions and binary decision diagrams. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 284–289, June 1990.

[Tis67] P. Tison. Generalization of consensus theory and application to the minimization of Boolean functions. *IEEE Trans. Electron. Comput.*, EC-16(4):446–456, August 1967.

Acknowledgment

I would like to express my sincere appreciation to Professor Shuzo Yajima of Kyoto University for his continuous guidance, interesting suggestions, accurate criticisms and encouragements throughout this research.

I would also like to express my thanks to Associate Professor Naofumi Takagi of Kyoto University who introduced me to the research field of logic design and has been giving me invaluable suggestions and accurate criticisms. I am also grateful for his comments and advices for the study described in chapter 2.

I would also like to express my thanks to Dr. Nagisa Ishiura of Osaka University who introduced me to the research field of supercomputing and binary-decision diagrams, and has been giving me invaluable suggestions and helpful advices throughout the study described in chapter 2 and 3.

I would also like to appreciate Professor Hiromi Hiraishi of Kyoto Sangyo University and Dr. Kiyoharu Hamaguchi of Kyoto University for their helpful suggestions and discussions. I am also grateful for their help in implementation of CTL model checker in section 3.5.

I would also like to thank Mr. Koichi Yasuoka of Kyoto University for his invaluable suggestions and comments especially for the study described in chapter 4.

I would also like to thank Mr. Shin-ichi Minato for his valuable suggestions and discussions on binary-decision diagrams. His SBDD subroutine library were indispensable for the evaluation in section 4.4.

I wish to express my gratitude to Mr. Katsutoshi Amitani for developing the vectorized prime implicant generator based on Quine-McCluskey method in section 2.6.

I would also like to thank Mr. Nobuyoshi Kaneda of Nittetsu Hokkaido Control Systems and Mr. Ichiro Okinaka of Nippon Steel for giving me an opportunity to utilize the semiconductor extended storage devices for the experiments in section 4.4.

Thanks are also due to all the members of the Professor Yajima's Laboratory for their discussions and supports throughout this research, especially to Mr. Hiroyuki Ogino, Mr. Kazuyoshi Takagi and Mr. Yasuhiro Fujiyoshi, who helped me in carrying out the experiments in section 4.4.

Appendixes

Proof of Lemma 1

[Lemma 1]
Let f be a k-variable Boolean function, and x_i a Boolean variable ($1 \leq i \leq k$). A product term p is a prime implicant of f, if and only if one of the following statements is true.

1. p is a prime implicant of $f(x_i = *)$.
2. p is a prime implicant of $\overline{x_i} f(x_i = 0)$, and does not imply $f(x_i = *)$.
3. p is a prime implicant of $x_i f(x_i = 1)$, and does not imply $f(x_i = *)$

*(proof)*It is clear that a prime implicant of f is an implicant of $\overline{x_i} f(x_i = 0)$ ($x_i f(x_i = 1)$) if $\overline{x_i}$ (x_i) appears in its representation. A prime implicant of f which is independent of x_i is an implicant of $f(x_i = *)$. Hence a prime implicant of f is an implicant of at least one of $\overline{x_i} f(x_i = 0)$, $x_i f(x_i = 1)$ and $f(x_i = *)$.

Suppose that there is an implicant of either $\overline{x_i} f(x_i = 0)$, $x_i f(x_i = 1)$ or $f(x_i = *)$ that is implied by a prime implicant of f. From the equation $f = \overline{x_i} f(x_i = 0) + x_i f(x_i = 1) + f(x_i = *)$, it follows that an implicant of either $\overline{x_i} f(x_i = 0)$, $x_i f(x_i = 1)$ or $f(x_i = *)$ is an implicant of f. Hence there is an implicant of f that is implied by another implicant of f, which contradicts the definition of prime implicant; accordingly, a prime

implicant of f is a prime implicant of either $\overline{x_i}f(x_i = 0)$, $x_i f(x_i = 1)$ or $f(x_i = *)$.

A prime implicant of $f(x_i = *)$ is independent of x_i, thus it does not imply both $\overline{x_i}f(x_i = 0)$ and $x_i f(x_i = 1)$. A prime implicant of either $\overline{x_i}f(x_i = 0)$ or $x_i f(x_i = 1)$ which is not a prime implicant of f is a non-prime implicant of f, and the prime implicant of f which is implied by it is obviously a prime implicant of $f(x_i = *)$. □

Proof of Lemma 3

[Lemma 3]
Let f be an k-variable Boolean function and $g_1, \ldots$ and g_j $(j \geq 0)$ be k-variable Boolean functions which implies f, and x_i be a Boolean variable $(1 \leq i \leq k)$. p is a prime implicant of f which implies neither $g_1, \ldots$ nor g_j, if and only if

1. p is a prime implicant of $f(x_i = *)$ which implies neither $g_1(x_i = *)$, $\ldots$, nor $g_j(x_i = *)$, or
2. p is a prime implicant of $\overline{x_i}f(x_i = 0)$ which implies neither $\overline{x_i}g_1(x_i = 0), \ldots, \overline{x_i}g_j(x_i = 0)$ nor $f(x_i = *)$, or
3. p is a prime implicant of $x_i f(x_i = 1)$ which implies neither $x_i g_1(x_i = 1), \ldots, x_i g_j(x_i = 1)$ nor $f(x_i = *)$.

*(proof)*By mathematical induction on j.

Case 1 : When $j = 0$, Lemma 3 is equivalent to Lemma 1.

Case 2 : Assume that Lemma 3 is true for $j = j_0 \geq 0$. On a Boolean function, say g_{j_0+1}, which implies f, it is obvious that

- a prime implicant of category (1) of the statement of Lemma 3 that implies g_{j_0+1} implies $g_{j_0+1}(x_i = *)$, and

- a prime implicant of category (2) ((3)) that implies g_{j_0+1} implies $\overline{x_i} g_{j_0+1}(x_i = 0)$ $(x_i g_{j_0+1}(x_i = 1))$.

Hence Lemma 3 is also true for $j = j_0 + 1$. □

List of Publications by the Author

Major Publications

1. N. Takagi, H. Ochi and S. Yajima : "Vector Algorithms for Generating Prime Implicants of Logic Functions", Proc. Third International Conference on Supercomputing, vol. 3, pp. 281-287, (May 1988).

2. H. Ochi, N. Takagi and S. Yajima : "Vector Algorithms for Generating Prime Implicants of Logic Functions Based on Consensus Expansion" (in Japanese), Trans. IEICE D-I, vol. J72-D-I, no. 9, pp. 652-659, (Sep. 1989).

3. H. Ochi, N. Ishiura and S. Yajima : "Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing", Proc. 28th ACM/IEEE Design Automation Conference, pp. 413-416, (June 1991).

4. H. Hiraishi, K. Hamaguchi, H. Ochi and S. Yajima : "Vectorized Symbolic Model Checking of Computation Tree Logic for Sequential Machine Verification", Proc. Third Workshop on Computer Aided Verification, vol. I, pp.279-290, (July 1991).

5. H. Ochi, N. Ishiura and S. Yajima : "A Vector Algorithm for Manipulating Boolean Functions Based on Shared Binary Decision Diagrams", Proc. International Symposium on Supercomputing, pp. 191-200, (Nov. 1991).

6. H. Ochi, N. Ishiura and S. Yajima : "A Vector Algorithm for Manipulating Boolean Functions Based on Shared Binary Decision Diagrams", Supercomputer 46, vol. VIII, no. 6, pp. 101-118, ASFRA, (Nov. 1991).

7. H. Ochi, K. Yasuoka and S. Yajima : "Breadth-First Manipulation of Very Large Binary-Decision Diagrams", Proc. International Conference on Computer-Aided Design 93, pp. 48-55, (Nov. 1993).

Technical Reports

1. H. Ochi, N. Takagi and S. Yajima : "High-Speed Generation of Prime Implicants of Logic Functions on Vector Processors" (in Japanese), Technical Report of IPS Japan, vol. 88, no. 10, 88-DA-41, (Feb. 1988).

2. H. Ochi, N. Takagi and S. Yajima : "On the Number of Prime Implicants of Logic Functions" (in Japanese), Technical Report of IEICE, vol. 89, no. 85, COMP89-23, (June 1989).

3. H. Ochi, N. Ishiura, N. Takagi and S. Yajima : "A Breadth-First Vector Algorithm for Manipulating SBDD", Technical Report of IPS Japan, vol. 91, no. 11, 91-AL-19, (Jan. 1991).

4. H. Ochi, H. Sawada, K. Okada, A. Uejima, H. Kambara, K. Hamaguchi and H. Yasuura : "A Microprocessor for Education of Computer Engineering and Integrated Circuit Design : KUE-CHIP2" (in Japanese), Technical Report of IPS Japan, vol. 92, no. 82, 92-ARC-96, pp. 93-100, (Oct. 1992).

5. H. Ochi, K. Yasuoka and S. Yajima : "A Secondary Storage Oriented Breadth-First Algorithm for Manipulating Very Large SBDD's", Tech-

nical Report of IPS Japan, vol. 93, no. 6, 93-ARC-98/93-DA-65, pp. 25-32, (Jan. 1993)

6. H. Ochi, K. Yasuoka and S. Yajima : "A Breadth-First Algorithm for Manipulating Very Large Shared Binary-Decision Diagrams" (in Japanese), Proc. Design Automation Symposium '93, IPS Japan, pp. 121-124, (Aug. 1993).

Convention Records

1. H. Ochi, N. Takagi and S. Yajima : "Generating Prime Implicants of Logic Functions on a Vector Processor by Consensus Expansion Method with Table Look-up" (in Japanese), Proc. 36th Annual Convention IPS Japan, vol. 3, pp. 1987-1988, (May 1988).

2. H. Ochi, N. Takagi and S. Yajima : "On the Maximum Number of Prime Implicants of 5- and 6-Variable Logic Functions" (in Japanese), 1989 Spring National Convention Record, IEICE, vol. 6, p. 92, (May 1989).

3. H. Ochi, N. Ishiura, N. Takagi and S. Yajima : "A Vector Algorithm for Boolean Function Manipulation Based on Shared Binary Decision Diagrams" (in Japanese), Record of the 1990 Kansai-Section Joint Convention of Institutes of Electrical Engineering, Japan, p. S57, (Oct. 1990).

4. H. Ochi, N. Ishiura and S. Yajima : "A Vector Algorithm for Manipulating SBDD" (in Japanese), Proc. 42nd Annual Convention IPS Japan, vol. 6, pp. 156-157, (May 1991).

5. H. Ochi, K. Yasuoka and S. Yajima : "A Breadth-First Algorithm for Efficient Manipulation of Shared Binary Decision Diagrams in the

Secondary Memory" (in Japanese), Proc. 45th Annual Convention IPS Japan, vol. 6, pp. 137-138, (Oct. 1992).

6. H. Ochi, K. Yasuoka and S. Yajima : "A Breadth-First Algorithm for Manipulating Shared Binary-Decision Diagrams in Secondary Storage" (in Japanese), Proc. 1993 IEICE Fall Conference, vol. 1, p. 60, (Sep. 1993).

Miscellaneous

1. H. Ochi, K. Yasuoka and S. Yajima : "Secondary Storage Oriented Breadth-First Manipulation of Very Large Shared Binary Decision Diagrams", KUIS Technical Report, KUIS-92-0005, Kyoto University, Japan, (Dec. 1992).

2. H. Ochi, K. Yasuoka and S. Yajima : "A Secondary Memory Oriented BDD Manipulator Using Garbage Collection Based on Sliding Type Compaction", KUIS Technical Report, KUIS-93-0007, Kyoto University, Japan, (Apr. 1993).