Improved Computational Methods and Lazy Evaluation of the Ordered Ternary Decision Diagram

Per Lindgren
Division of Computer Engineering
Luleå Institute of Technology
S-971 87 Luleå, Sweden
pln@sm.luth.se

Abstract— We investigate the properties of the Ordered Ternary Decision Diagram (OTDD) in order to develop an efficient general OTDD package. The OTDD is a three-branched three-terminal diagram based on Kleenean strong ternary logic. The OTDD can represent functions having nontrivial don't-care sets in a single diagram and is capable of provably correct evaluation in the presence of unknown input values. We propose a number of improvements to both OTDD computational methods and data structures. Furthermore we introduce the purged form OTDD which unifies the abbreviated and full form OTDD into a single diagram. A package exploiting these OTDD specific properties is presented and we show the computational advantages of this improved package for LGSynth93 standard benchmarks.

I. Introduction

In computer engineering applications we are often confronted with complex Boolean functions containing nontrivial don't-care sets. Furthermore to accurately and efficiently handle undefined logic input values is another problem related to logic simulation applications, synthesis and testability, see [JIL94], [CP89] and [CA87]. The Ordered Ternary Decision Diagram (OTDD) as proposed in [Jen95] solves the accurate evaluation problem in the presence of unknown inputs, and is capable of representing functions having nontrivial don't-care sets in a single diagram. The OTDD is based on Kleenean strong ternary logic permitting well defined logical operations on OTDD functions. For any number of undefined input logic values the output can be evaluated in O(n) time, where n is the number of inputs. In [Jen95] the basic algorithms for building and processing OTDDs are defined. However, efficiency and optimization of the actual implementation are only briefly discussed.

This paper discusses our experience from implementing a general OTDD package. Based on our observations we propose a number of improvements to the basic computational methods proposed in [Jen95]. We introduce the *purged* form OTDD which supports lazy evaluation of the OTDD. To evaluate our refinements of the original OTDD computational methods, a number of experiments were performed on standard benchmarks (LGSynth93). The experiments involve both building the OTDDs from the PLA-descriptions in LGSynth93 and performing logical operations on them. Our results show that the proposed improvements reduce memory requirements and increase computational speed.

The paper is organized as follows: Section II gives a short tutorial on the basic properties of the OTDD. In Section III we discuss the properties of the basic OTDD package, and propose a number of improvements exploiting OTDD specific properties. Section IV motivates our choice of experiments and contains a detailed description of these. Section V shows and discusses the results obtained from computations run on a Sun SPARC ELC.

II. A Short Tutorial on the OTDD

This section gives a short tutorial on the OTDD. For an in depth description we refer the reader to [Jen95] where the theoretical aspects of the OTDD are discussed.

The OTDD was introduced to satisfy the following criteria. First: representing functions having nontrivial don't-care sets in a single diagram; second: allowing evaluation of these functions even when given (any number of) undefined logic input values; and third: allowing operations to be performed on the functions. The OTDD is an ordered decision diagram with terminal values {'0','1','U'}, where 'U' represents a don't-care output value corresponding to the first criterion. From the second criterion follows that any nonterminal must be able to handle an undefined logic input value 'U'. This problem was solved by giving the nonterminals children 'low', 'high' and 'und' corresponding to input values {'0','1','U'}.

Removal of all nonterminals where 'low' = 'high' yields a *reduced* form OTDD. If 'und' is removed from all nonterminals an *abbreviated* form OTDD is obtained. Although the resulting *abbreviated* OTDD can not directly handle 'U' input values, all necessary information to retrieve a full form OTDD is preserved. The last crite-

rion, to allow operations on the functions represented as OTDDs, is addressed by using the semantics of Kleenean strong logic for evaluation. The *abbreviated* form is suitable for representing intermediate results during calculation on OTDDs, since it requires less space and computations to obtain. Note that the *abbreviated* form OTDD is still capable of representing functions having nontrivial don't-care sets.

Figure 1 shows the identity function for variable x_i (to the left) and the function $((x_1\ and\ x_2)\ or\ x_4)$ as a *full* form OTDD (to the right). Evaluation is done by traversing the OTDD from the root, for example the input vector x_1 = 'U'; x_2 = 'U'; x_4 = '1' corresponding to (('U' *and* 'U') *or* '1'), traverses x_1, x_2, x_4 and finally arrives at the terminal '1'. Though the input vector contains undefined logic values, the output will be correctly evaluated.

III. OTDD Computational Methods

We started out by implementing a basic OTDD package as described in [Jen95]. From this exercise we gained insight into some of the properties unique to the OTDD. Exploitation of these properties resulted in the optimizations proposed in the following sections.

A. The Basic OTDD Package

Our first OTDD package implemented the computational methods suggested in [Jen95]. The package includes Kleenean AND, OR, XOR and COMPOSE as well as the OTDD specific operations OR_B_TO_U, TERNATE and ALIGNMENT. The COMPOSE operation $G|_{v_i=F}$ replaces the input variable v_i of function G with the function F, and yields a single (*full* form) OTDD. The operation OR_B_TO_U is used to create an *abbreviated* form OTDD from ON and DC set OTDDs. The TERNATE operator transforms an *abbreviated* form

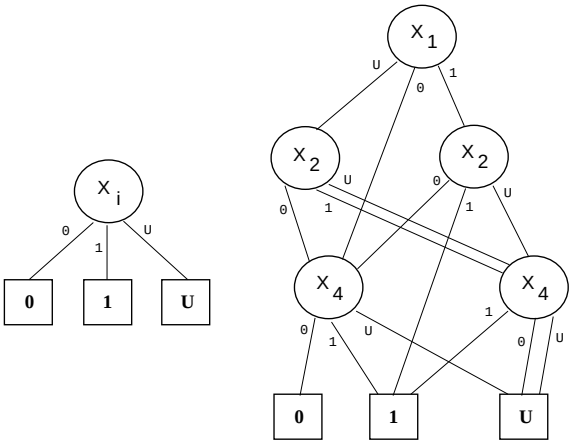

Fig. 1. Simple Ordered Ternary Decision Diagrams. To left: the identity function for variable x_i. To right: the function $((x_1\ and\ x_2)\ or\ x_4)$.

```
node {
  index:         int
  val:           {0,1,U,X}
  low,high,und: *node  }
```

Fig. 2. OTDD node description, 'U' represents a don't-care terminal.

OTDD into a *full* form OTDD by recursively visiting each node, and deriving their 'und' branches by applying ALIGNMENT on that node's 'low' and 'high' branches. We refer the reader to [Jen95] for detailed description and pseudocode for these operations.

The node description used in our basic package can be found in Figure 2. The 'index' field indicates to which input variable this specific node applies. Note that value 'X' is given to all nonterminals and that the value 'U' indicates a don't-care terminal. All nodes are cached in a unique-table as described in [BRB90].

To perform operations on the OTDDs, the *Apply_step* routine first described by [Bry86] and adapted for the OTDD in [Jen95] is used. *Apply_step* is called recursively, traversing the operands towards their terminals. *Apply_step* uses an "Op" function, which given the 'val' fields of the top node operands, can determine terminal and controlling cases, the latter preventing further traversal. In [Jen95] a flag is used to indicate whether *Apply_step* will calculate a full or an *abbreviated* form OTDD. In our package *Apply_step* yields an *abbreviated* form OTDD, a full form OTDD can be derived *postfacto* by an explicit TERNATE operation. A memory function proposed in [BRB90] insures that each unique operation is only performed once.

B. Refinement of the "Op" Function

The "Op" function in our basic package forces evaluation of the operands all the way down to the terminals if no controlling cases are found. We seek a way to prevent that kind of exhaustive evaluation.

Under Kleenean logic $(f\ and\ f) = f$, which intrigues us to consider isomorphic operands. In these cases we might be able to prevent further traversal of the operands.

The OTDD has a strong canonical form, proven in [Jen95]. This together with the unique-table representation of OTDDs allows us to detect isomorphism between OTDDs as pointer equality, that is, in constant time.

Based on these observations an improved "Op" function exploiting isomorphism is proposed in [Jen95]. The modified "Op" function is shown in pseudocode in Figure 3. The *iso* flag indicates that the two operands are isomorphic. The OR operation is modified to handle $(f\ or\ f) = f$ by adding $(iso) => return\text{`}F\text{'}$, the AND operation is modified to handle $(f\ and\ f) = f$ in the same way. The

```
char Op( a, b, opcode, iso ) {
  (opcode == OR):
     ((a == '1')||(b == '1')) => return '1' else
     ((a == 'X')||(b == 'X')):
        (iso)                 => return 'F' else
        (a == '0')            => return 'G' else
        (b == '0')            => return 'F' else
        return 'X';
     ((a == '0')&&(b == '0')) => return '0' else
     return 'U';
  (opcode == AND):
     ((a == '0')||(b == '0')) => return '0' else
     ((a == 'X')||(b == 'X')):
        (iso)                 => return 'F' else
        (a == '1')            => return 'G' else
        (b == '1')            => return 'F' else
        return 'X';
     ((a == '1')&&(b == '1')) => return '1' else
     return 'U';
  (opcode == XOR):
     ((a == 'X')||(b == 'X')):
         /* (iso) => return '0' NON-KLEENEAN */
        return 'X';
     ((a == '1')&&(b == '1')) => return '0' else
     ((a == '1')&&(b == '0')) => return '1' else
     ((a == '0')&&(b == '1')) => return '1' else
     ((a == '0')&&(b == '0')) => return '0' else
     return 'U';
  (opcode == OR_B_TO_U):
     ((b == '1')||(b == 'U')) => return 'U' else
     ((a == 'X')||(b == 'X')) => return 'X' else
     return a;
  (opcode == ALIGNMENT):
     ((a == 'U')||(b == 'U')) => return 'U' else
     ((a == 'X')||(b == 'X')):
        (iso)                 => return 'F' else
        return 'X';
     (a == b)                 => return  a  else
     return 'U'; }
```

Fig. 3. The improved "Op" function, (iso) indicates that the operands are isomorphic.

ALIGNMENT operator (used to create the 'und' branch), conforms to $(f\ alignment\ f) = f$, see [Jen95]. This allows the same modification of the ALIGNMENT operation as described for the OR operation.

Under Kleenean-strong logic the XOR operation does not conform to $(f\ xor\ f)$ = '0', hence adding $(iso) =>$ *return*'0' would violate the Kleenean semantics.

C. The Purged form OTDD

At this point we made two critical observations.

First observation: the OTDD is canonical with respect only to the 'index' of the input variable and its 'low' and 'high' branches.

Second observation: the 'und' branch can be derived for any node at any time by computing the ALIGNMENT of that node's 'low' and 'high' branches.

The first observation immediately leads to the conclusion that the very same node pointed out in the unique-table can hold both the *abbreviated* and *full* form OTDD. We introduce the *purged* form OTDD allowing any mix of *abbreviated* and *full* form OTDD nodes in a single diagram. By using this new form we hope to reduce both the number of unique OTDD nodes allocated, and the size of the unique-table. The reader must notice that although a node might contain a defined 'und' pointer it is not to be considered as a *full* form OTDD, because the 'low' and 'high' branches are in no way guaranteed to be of *full* form. Note that the ALIGNMENT operation will effect all OTDDs that contain the aligned node.

To implement the *purged* form OTDD some modifications of the basic package are necessary. We propose a modified 'uniqueness' criterion to be used for the unique-table (involving only 'index', 'low' and 'high' branches, opposed to the criteria used in [Jen95] which involves the 'und' branch). Furthermore we propose a "ut_insert" function, see [Jen95], which provides 'und' branch *updating* of existing *abbreviated* OTDD nodes. These modifications cause *Apply_step* to produce *purged* form OTDDs rather than *abbreviated* OTDDs.

The second observation is to some extent exploited in [Jen95]. The *Apply_step* function, as proposed in [Jen95], takes a parameter that indicates if the result is to be calculated as an *abbreviated* or a *full* form OTDD. Intermediate calculations can be performed producing *abbreviated* OTDDs (which require less memory and CPU resources) and the final result can be expanded to the *full* form OTDD. However *Apply_step* calculates either a *full* form OTDD or else a fully *abbreviated* OTDD.

Furthermore the second observation leads to the conclusion that the ALIGNMENT operation should not be issued until absolutely necessary, i.e. when required to perform an operation. If done earlier, 'und' nodes might be computed which are neither required for the computation, nor part of the final result. All such work is wasted !

The *Compose* algorithm, suggested in [Jen95], requires the downstream function to be represented as a *full* form OTDD (though Jennings briefly discusses the possibility of expanding the 'und' branches only when needed). This means we would expect significant improvements by using the new *purged* form OTDD for lazy evaluation.

As a result of our observations we propose an improved package including the new unique-table criteria, a node update function, and a new *Compose* algorithm, see Figure 4. The pseudocode actually holds both the original *compose_step* as proposed in [Jen95], shown as (1), and our proposed optimized function, shown as (2). Observe that (1) excludes lines corresponding to (2) and *vice versa*. The optimization can be broken down in several steps. To prevent unnecessary computation of 'und' branches we allow both functions to be represented in *purged* form, relaxing the *full* form assumption in [Jen95]. If and *only* if the 'und' branch 'vu1' is required, see (1b), we derive it from 'vl1' and 'vh1' by applying ALIGNMENT, see (2a). This operation derives 'vu1' as

```
node *compose_step(vl1, vh1, vu1, v2) {
    /* v1|m = v2 */
    (vl1.index == m) => vl1 = vl1.low
    (vh1.index == m) => vh1 = vh1.high
1a) (vu1.index == m) => vu1 = vu1.und

    /* terminal in v1 */
    (v2.val=='0') and (vl1.val terminal) =>return(vl1.val)
    (v2.val=='1') and (vh1.val terminal) =>return(vh1.val)
1b) (v2.val=='U') and (vu1.val terminal) =>return(vu1.val)
2a) (v2.val=='U') => {vu1 = apply_step(vl1,vh1,ALIGNMENT) =>
                  and (vu1.val terminal) =>return(vu1.val)}

    u.index=Min(vl1.index,vh1.index,vu1.index,v2.index)
    (v2.index==u.index)  =>{v2low=v2.low; v2high=v2.high}
                    else =>{v2low=v2;     v2high=v2}

    (vl1.index==u.index) =>{vll1=vl1.low; vlh1=vl1.high}
                    else =>{vll1=vl1;     vlh1=vl1}

    (vh1.index==u.index) =>{vhl1=vh1.low; vhh1=vh1.high}
                    else =>{vhl1=vh1;     vhh1=vh1}

1a) (vu1.index==u.index) =>{vul1=vu1.low; vuh1=vu1.high}
                    else =>{vul1=vu1;     vuh1=vu1}

    u.low  = compose_step(vll1, vhl1, vul1, v2low)
    u.high = compose_step(vlh1, vhh1, vuh1, v2high)
    r      = ut_insert(u.index, u.low, u.high)
1c) r.und  = apply(u.low, u.high, ALIGNMENT, FULL)
    return (r);
}
node *Compose(v1, v2) {
    return compose_step(v1, v1, v1, v2);
}
```

Fig. 4. The optimized compose_step function. (1) shows the original function, (2) shows the optimized function.

a *purged* form OTDD, which is exactly what we need at that time.

Furthermore we benefit from neglecting the incoming 'vu1' branch and from avoiding unnecessary calculations of 'vul1' and 'vuh1' by eliminating (1a). The second observation is further exploited by eliminating (1c), thus the result of *compose_step* will be a *purged* OTDD.

IV. Description of the Experiment

In order to measure the efficiency of our package with respect to memory usage, CPU usage etc we used test cases consisting of PLA-function descriptions from LGSynth93. The PLA descriptions contain both the ON set which evaluates to logic one and the explicit DC set (if any) which evaluates to logic 'U'. The strong Kleenean 'U' means a valid logic one or logic zero, but we don't know or don't-care which.

A. Building of the *abbreviated* form OTDDs

During parsing we initially represent the ON and DC sets as OBDDs, see [Bry86]. Building the OBDDs involves both AND and OR operations. To represent both ON and DC set in one OTDD is done by applying the OR_B_TO_U operator to the ON and DC sets. The result of the OR_B_TO_U operation is an *abbreviated* form OTDD.

Several properties are measured during the procedure:

- memory usage – the total number of nodes allocated
- wasted nodes – the number of nodes allocated that are not part of the result
- CPU time used when performing logical operations

By this experiment we hope to gain insight in the performance of the basic OTDD and furthermore to evaluate the improved "Op" function.

B. Expansion to *full* form OTDDs

This is done by applying the unary operator TERNATE, which recursively applies the ALIGNMENT operator to all nodes in the *abbreviated* form OTDD, the result is a *full* form OTDD. The ALIGNMENT operator processes the output of the 'low' and 'high' branches into the 'und' branch. We refer the reader to [Jen95] for further details.

The same measurements of memory usage and CPU load, as described above, are taken during the operations. By this experiment we seek further understanding of the properties regarding the basic and improved "Op" function.

C. Retrieval of the *full* form OTDDs

Now we can apply any logic function (AND, OR, NOT or XOR) to the OTDDs, and study the performance of our package. For this experiment we chose the NOT operator (implemented as F XOR '1'). We performed two consecutive NOT operations, thus yielding the original function. We will be able to probe the package's ability to retrieve a *full* OTDD from an *abbreviated* form OTDD in the case that TERNATE already has been applied to the original function, see B.

D. Composing OTDD functions

Any two functions F and G represented as OTDDs can be composed, $G|_{v_i=F}$, where input variable v_i of downstream function G is replaced by the function F. Since the PLA descriptions in LGSynth93 correspond to functions F with different numbers of outputs F_i, we chose only to compose the first output F_1 with the first input of the downstream function, $G_1|_{v_1=F_1}$. Of course any other combination would do as well, but this choice serves our

purposes. The ability to handle upstream functions having nontrivial don't-care sets is examined by PLAs '*spla*', '$misex3c$' and '$ex1010$'. This forces the proposed lazy *compose_step* algorithm to derive the 'und' branch while composing the functions, see (2a) in Figure 4. The input vectors to functions F and G can have any number of common variables. We test the ability to handle both completely disjoint sets of input variables as well as the case of maximum possible common variables.

V. Results of the experiments

We present our refinements of the OTDD package from our first implementation towards the final version. To probe the efficiency gain of the separate improvements and how they affect each other our package allows us to turn the features on and off.

During the experiments several measurements were taken. Table I shows the number of nodes wasted when consecutively performing experiments IV A, B and C, as well as the total number of nodes allocated during the suite. When building the *abbreviated* form OTDD, see IV A, we typically get an improvement of more than 30% (peak at 75%) with respect to the number of wasted nodes. When expanding to *full* form, see IV B, the total improvement varies from 10% to 50%. After performing the two consecutive NOT operations, see IV C, the total improvement varies from 5% to 50%.

These results fulfill our expectations for the improved "Op" function. We successfully prevent exhaustive evaluation of the OTDDs by detecting isomorphic operands and utilizing the evaluation under Kleenean strong logic.

Table II compares the total CPU time when the experiments IV A, B and C are executed on a Sun SPARC ELC. The columns marked 'lazy' indicates that our uniquetable criterion proposed in Section III C is used. The *purged* form OTDD increases computational speed by up to 30%.

From the results we conclude that the critical section is the building of the OTDDs, this result also applies to Table I. When the *abbreviated* OTDD is calculated once and for all, operations on the OTDDs are performed almost instantly. This proves the usability of the OTDD in practical applications.

Table III compares our improved *Compose* ($G|_{v_i=F}$) function to the basic implementation suggested in [Jen95].The input vectors to functions F and G can have any number of common variables. For test purposes the input sets are either disjoint or have the maximum possible common variables, see column 'disjoint inputs'. The next column 'abbrev' shows the number of nodes allocated to build the *purged* form OTDDs. The next two columns show the total number of nodes allocated and the number of ALIGNMENT entries in the memory function after performing the *compose_step* operation. The results after expanding the *purged* form OTDD to a *full* OTDD are shown in the next two columns. Finally the two rightmost columns show measurements taken on the basic OTDD package for comparison. Note that only *full* form OTDDs can be obtained from the original package.

When presented upstream functions having nontrivial don't-care sets, our proposed *compose_step* derives the required 'und' branches under lazy evaluation. '$alu4|_{1=spla1}$' and '$misex3c|_{1=ex1010}$' show that composition deriving the *purged* form only requires a fraction of the ALIGNMENT operations necessary for obtaining the *full* form. Our experimental results show that our proposed *compose_step* can derive a *purged* form OTDD mainly by using already allocated memory resources. The new *compose_step* compares favorably to the basic implementation even when expanding the *purged* form OTDD to *full* form. However, the essence of this paper is to show the advantages of using the *purged* form representation for all intermediate computations. The presented results verify this.

VI. Conclusions

We investigate the properties of The Ordered Ternary Decision Diagram (OTDD) in order to develop an efficient general OTDD package. We exploit OTDD specific properties into both improvements of the computational methods and data structures. We propose the *purged* form OTDD which allows any mixture of *abbreviated* and *full* form OTDD nodes in a single diagram, and can furthermore be utilized for lazy evaluation of the OTDD. A package exploiting these OTDD specific properties has been presented and we have demonstrated the computational advantages of this improved package on LGSynth93 standard benchmarks.

References

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings 27th DAC*, pages 40–45, June 1990.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CA87] Hongtao P. Chang and Jacob A. Abraham. The Complexity of Accurate Logic Simulation. In *Proceedings, IEEE Int'l Conf. on CAD (ICCAD '87)*, pages 404–407, November 1987.

[CP89] Susheel J. Chandra and Janak H. Patel. Accurate Logic Simulation in the Presence of Unknowns. In *Proc. Intl. Conf. on Computer-Aided Design (ICCAD '89)*, pages 34–37, November 1989.

[Jen95] G. Jennings. Symbolic Incompletely Specified Functions for Correct Evaluation in the Presence of Indeterminate Input Values. In *Twenty-Eighth Annual Hawaii Int'l Conference on System Sciences, HICSS-28, Volume I: Architecture*, pages 23–31, January 1995.

[JIL94] G. Jennings, J. Isaksson, and P. Lindgren. Ordered Ternary Decision Diagrams and the Multivalued Compiled Simulation of Unmapped Logic. In *Proceedings, 27th Annual Simulation Symposium*, pages 99–105, April 1994.

	out	PLA → *abbreviated* OTDD		*abbreviated* → *full* OTDD		2*XOR		$\sum$ nodes	
		basic	"Op"	basic	"Op"	basic	"Op"	basic	"Op"
9sym	9	2700	1765	2675	1740	2784	1849	2817	1882
5xp1	10	794	532	879	617	1157	895	1245	983
duke2	29	6294	3921	19335	16962	28327	25954	29303	26930
table5	15	6797	3867	28716	25768	42295	45225	45909	42979
alu4	8	46810	34547	66979	54716	86730	74467	88082	75819
spla	46	160711	134587	163435	137311	166374	140250	167059	140935
misex3c	14	40861	29719	58231	47089	66380	55238	68507	57365
ex1010	10	61930	57435	65134	60639	69742	65247	71669	67174

TABLE I

THE NUMBER OF WASTED NODES (NODES ALLOCATED BUT NOT PART OF THE RESULT) DURING THE EXPERIMENTS, SEE SECTION IV A, B AND C. THE TOTAL NUMBER OF ALLOCATED NODES IS FOUND IN THE RIGHTMOST COLUMN.

	out	*abbreviated*			*full* OTDD			2*XOR			retrieve *full*		
		basic	"Op"	lazy	basic	"Op"	lazy	basic	"Op"	lazy	basic	"Op"	lazy
9sym	1	0.16	0.17	0.15	0.16	0.18	0.16	0.16	0.19	0.16	0.16	0.19	0.16
5xp1	7	0.06	0.03	0.04	0.09	0.07	0.06	0.10	0.07	0.08	0.10	0.07	0.08
duke2	29	0.39	0.26	0.37	2.02	2.01	1.59	2.13	2.12	1.65	2.13	2.12	1.65
table5	15	0.62	0.63	0.57	3.53	3.61	2.59	3.61	3.69	2.66	3.61	3.69	2.66
alu4	8	2.64	2.07	2.36	6.19	5.66	4.80	6.33	5.78	4.88	6.33	5.79	4.89
spla	46	22.32	21.84	21.89	22.86	22.33	22.25	22.93	22.41	22.30	22.93	22.41	22.30
misex3c	14	2.20	2.05	2.05	4.17	4.13	3.55	4.34	4.30	3.70	4.34	4.30	3.70
ex1010	10	9.14	9.13	8.57	9.91	9.88	9.13	10.08	9.97	9.28	10.08	10.04	9.28

TABLE II

CPU TIMES IN SECONDS FOR CONSECUTIVE EXPERIMENTS, SEE SECTION IV A, B AND C, RUN ON A SUN SPARC ELC.

	disjoint inputs	abbrev	*purged* compose		*full* compose		basic compose	
			alloc nodes	mem_a	alloc nodes	mem_a	alloc nodes	mem_a
alu4|1 = 5xp1	no	183	235	0	648	911	705	1029
alu4|1 = spla	no	471	520	104	634	379	711	506
spla|1 = alu4	no	471	516	0	534	185	563	245
misex3c|1 = ex1010	no	10176	10424	413	10663	3328	10815	5014
alu4|1 = 5xp1	yes	183	183	0	384	378	384	378
alu4|1 = spla	yes	471	471	18	669	383	671	385
spla|1 = alu4	yes	471	471	8	500	67	500	67
misex3c|1 = ex1010	yes	10176	10176	272	10433	3385	10441	3476

TABLE III

COMPOSITION OF FUNCTIONS, SEE SECTION IV D, WHERE 'SPLA', 'MISEX3C' AND 'EX1010' HAVE NONTRIVIAL DON'T-CARE SETS. 'ABBREV' SHOWS THE NUMBER OF NODES ALLOCATED TO BUILD THE *abbreviated* FORM OTDDS. 'ALLOC NODES' SHOWS THE TOTAL NUMBER OF NODES ALLOCATED, CORRESPONDING TO EACH OPERATION. 'MEM_A' SHOWS THE NUMBER OF ALIGNMENT ENTRIES IN THE MEMORY FUNCTION.