An Efficient Method for Generating Kernels on Implicit Cube Set Representations

Hiroshi Sawada, Shigeru Yamashita and Akira Nagoya
NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0237, JAPAN

1. Introduction

In the process of logic synthesis and optimization, it is an important operation to extract common logic parts among several logic functions. Most methods currently used are based on SOP (sum-of-products) expressions of logic functions. In such methods, special sub-expressions named kernels [1, 2] are generated, and attempts are made to find good intersections of kernels for common logic expressions.

A SOP expression can be represented by a set of cubes. Although it is a straight-forward way to enumerate cubes explicitly, there is a case where the number of cubes becomes so large that we can hardly manipulate them explicitly. The typical case is the SOP expression of an adder, which contains many XOR operations. To handle larger cube sets, methods have been proposed [3], [4] that represent cube sets implicitly on BDDs [5] or similar data structures. There are few methods [6], however, for extracting common logic expressions using implicit cube set representations.

In this paper, we propose a new method that efficiently generates all kernels for implicitly represented cube sets, and show how to apply it to the extraction of common logic expressions. The main feature is the memorization of the kernel generation process using a graph structure, which is called a kernel graph in this paper. Nodes of the graph are identified by the pair of a cube set and an integer value, and are easily accessible by a hash table. With this mechanism, we can skip the process of kernel generation that was processed before.

2. Preliminaries

2.1 Implicit cube set representation

A **literal** is a variable $(a, b, c, \cdots)$ or its negation $(a', b', c', \cdots)$. A **cube** is a product of literals $(ab'de, cde, \cdots)$. A **SOP (sum-of-products)** expression is a sum of cubes $(ab'de + cde + f)$, and can be seen as a **cube set** $(\{ab'de, cde, f\})$.

Meta-products [3] and Zero-suppressed BDDs [4] have been proposed to represent cube sets implicitly. In these structures, each cube corresponds to a path from the root node to the constant 1 node. There is a case where a very large cube set can be represented in a small number of nodes. The equivalence of two cube sets can be checked by comparing the identifiers of the root nodes because the same cube sets are represented by the same node.

```
void genKernel(Sop $K$, int $index$) {
    for ( int $i$ = $index$; $i > 0$; $i --$ ) {
        if ( literalCount$(K, l_i) \geq 2$ ) {
            Sop $childF$ = $K/l_i$;
            Sop $comCube$ = commonCube$(childF)$;
            if ( topLiteral$(comCube) < i$ ) {
                genKernel$(childF/comCube, i-1)$;
            }
        }
    }
    registerKernel$(K)$;
}
```

Figure 1. An algorithm for kernel generation

2.2 Kernel

A SOP expression is called **cube-free** [1, 2] if there exists no literal that appears in all cubes in the expression. For instance, $ab + a'c$ is cube-free, and $ab + bc$ and abc are not. For a SOP expression F and a cube c, F/c is called a **kernel** of F if it is cube-free. For an expression $F_1 = abcg' + abd + ae + bc'fg + c'fh$, $F_1/ab = cg' + d$ is a kernel, and $F_1/c' = bfg + fh$ is not. A cube used to make a kernel is called a **co-kernel**. In the previous example, co-kernel ab is used to make kernel $cg' + d$.

A well-known algorithm [2] to generate all kernels of a given expression F is as follows:

1. Divide F with literals that appear in all cubes and let the quotient be K.
2. Order all literals in K as $l_n, l_{n-1}, \ldots, l_1$.
3. Call the recursive procedure shown in Fig. 1 as *genKernel*(K, n).

In Fig. 1, *literalCount*(K, l_i) counts the number of literals l_i in K, *commonCube*(F) returns the product of literals that appear in all cubes in F, and *topLiteral*(F) returns the maximal index of all literals in F.

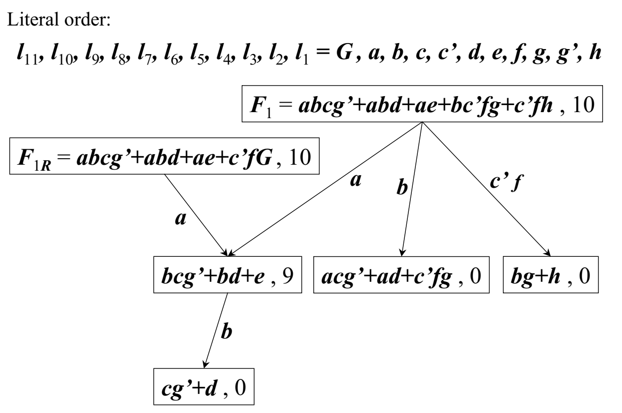
Literal order:
l11, l10, l9, l8, l7, l6, l5, l4, l3, l2, l1 = G, a, b, c, c', d, e, f, g, g', h
F1 = abcg'+abd+ae+bc'fg+c'fh , 10
F1R = abcg'+abd+ae+c'fG , 10
a
a
b
c' f
bcg'+bd+e , 9
acg'+ad+c'fg , 0
bg+h , 0
b
cg'+d , 0

Figure 2. Example 1 of a kernel graph

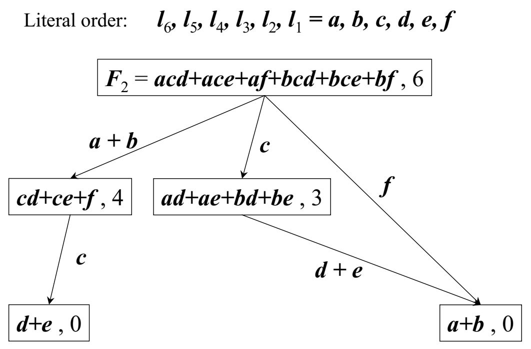
Literal order: l6, l5, l4, l3, l2, l1 = a, b, c, d, e, f
F2 = acd+ace+af+bcd+bce+bf , 6
a + b
c
f
cd+ce+f , 4
ad+ae+bd+be , 3
c
d + e
d+e , 0
a+b , 0

Figure 3. Example 2 of a kernel graph

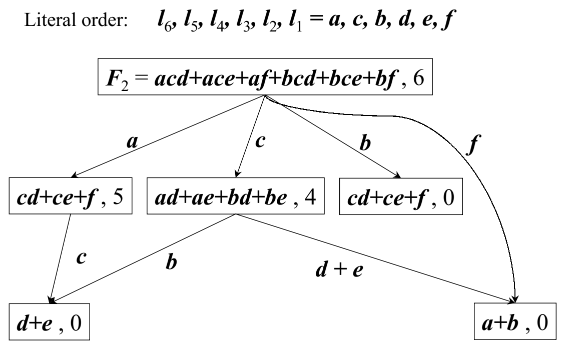
Literal order: l6, l5, l4, l3, l2, l1 = a, c, b, d, e, f
F2 = acd+ace+af+bcd+bce+bf , 6
a
c
b
f
cd+ce+f , 5
ad+ae+bd+be , 4
cd+ce+f , 0
c
b
d + e
d+e , 0
a+b , 0

Figure 4. Example 3 of a kernel graph

3 Kernel graph

In this section, we propose a graph structure that memorizes the kernel generation process.

3.1 Definition

A kernel graph consists of nodes and directed edges as defined below:
nodes having a cube set K and an integer value $topIndex$,
directed edges having a cube set D.
K is the kernel associated with the node, and $topIndex$ is the maximal index of the literals appearing in the D's of all outgoing edges. If a node has no edge, its $topIndex$ is 0. Let *node*$(K, topIndex)$ be a node having K and $topIndex$.

In *node*$(K, topIndex)$, there are edges associated with literal l_i which satisfies the following three conditions:

1. $i \leq topIndex$,
2. *literalCount*$(K, l_i) \geq 2$ and
3. *topLiteral*$(comCube) < i$, where $comCube$ = *commonCube*(K/l_i).

An edge associated with literal l_i has a cube $l_i \cdot comCube$ in its D. An edge having D and going out from a node having K points to a node having K/D.

We define that two nodes having the same K and the same $topIndex$ are equivalent. The definition of $topIndex$ leads that two equivalent nodes have the same set of edges: each edge has the same D and points to the equivalent node. Two equivalent nodes are shared and only one can exist in a kernel graph.

3.2 Generating algorithm

An algorithm for generating a kernel graph is shown in Fig. 5. This algorithm is a modification of the ordinary kernel generation algorithm shown in Fig. 1. The main differences are 1) the pointer of a node is returned as a result, and 2) whether a required node already exists or not is examined after $topIndex$ is determined. In the algorithm, the integer value i decreases one by one from the initial value $index$. Therefore, $topIndex$ is defined as the first i such that l_i satisfies the two conditions 2 and 3. If no literal satisfies the two conditions, $topIndex$ remains as 0, as an initial value.

After $topIndex$ is determined, we examine whether *node*$(K, topIndex)$ already exists or not. Every time we make a new node, we register it in a hash table. The key used in the hash table is generated from the identifier of cube set K (implicitly represented) and $topIndex$. Therefore, we can efficiently examine the existence of an equivalent node. If an equivalent node exists, the algorithm returns the node and omits the further processing. If it does not exist, the required node is made and the process continues with further recursive calls. In Fig. 5, *addChild*$(resultNode, childNode, l_i \cdot comCube)$ makes the directed edge from $resultNode$ to $childNode$ if it does not exist, and updates its SOP expression from D to $D + l_i \cdot comCube$.

Here, we show some examples. In Fig. 2, the situation is that after the kernel graph of F_1 was constructed, the expression $G = bg + h$ is selected as a common logic expression, and F_1 is revised to F_{1R} by using G. The node *node*$(bcg' + bd + e, 9)$ is shared by the graphs of F_1 and F_{1R}. This means that a portion of the process for F_{1R} can be trimmed. Figure 3 shows a case where a portion of the process can be skipped for only one expression. A node *node*$(cd + ce + f, 4)$ is pointed to by an edge having $a + b$.

```
/* KGnode*: a pointer to a kernel graph node, Sop: a SOP expression */
KGnode* genKernelGraph(Sop $K$, int $index$) {
    KGnode* $resultNode$ = 0;  /* the pointer to a result node */
    int $topIndex$ = 0;
    for ( int $i$ = $index$; $i > 0$; $i--$ ) {
        if ( literalCount($K, l_i$) $\geq 2$ ) {  /* condition 2 */
            Sop $childF$ = $K/l_i$;
            Sop $comCube$ = commonCube($childF$);
            if ( topLiteral($comCube$) $< i$ ) {  /* condition 3 */
                if ( $topIndex$ == 0 ) {  /* true only at the first time */
                    $topIndex$ = $i$;  /* the maximum because $i$ decreases */
                    $resultNode$ = lookup($K, topIndex$);  /* examine whether the equivalent node exists */
                    if ( $resultNode$ ) return $resultNode$;  /* if it exists, return it and quit */
                    $resultNode$ = makeNode($K, topIndex$);  /* make a new node */
                }
                KGnode* $childNode$ = genKernelGraph($childF/comCube, i-1$);
                addChild($resultNode, childNode, l_i \cdot comCube$);
            }
        }
    }
    if ( $topIndex$ == 0 ) {  /* no edge */
        $resultNode$ = lookup($K$, 0);  /* examine whether the equivalent node exists */
        if ( $resultNode$ ) return $resultNode$;  /* if it exists, return it and quit */
        $resultNode$ = makeNode($K$, 0);  /* make a new node */
    }
    return $resultNode$;
}
```

Figure 5. An algorithm for generating a kernel graph

This means that the required nodes are the same for both kernels F_2/a and F_2/b. Figure 4 shows the reason why the integer value $topIndex$ is needed. For the expression F_2 shown in Fig. 3, we have another kernel graph shown in Fig. 4 if we change the order of literals b and c. In this graph, two different nodes, *node*$(cd+ce+f, 5)$ and *node*$(cd+ce+f, 0)$, are needed for kernel $cd + ce + f$.

3.3 Calculation of sets of co-kernels

In a kernel graph, the set of co-kernels for a kernel can be calculated recursively. For a node N, let $E_1, \ldots, E_p$ be incoming edges and let $N'_1, \ldots, N'_p$ each be a node from which the corresponding edge goes out. Then, the set of co-kernels *CokernelSet*(N) of a node N is given by *CokernelSet*$(N) = \sum_{i=1}^{p} D(E_i) \cdot$*CokernelSet*$(N'_i)$, where $D(E_i)$ is the cube set D of an edge E_i. If there are more than one node $N_1, \ldots, N_q$ for a kernel, the set of co-kernels for the kernel is given by $\sum_{i=1}^{q}$*CokernelSet*(N_i).

For example, in Fig. 3, the set of co-kernels for kernel $d + e$ is calculated as $c \cdot (a + b) = ac + bc$. In Fig. 4, the calculation for the co-kernel is different but the result is the same. It is given by $c \cdot a + b \cdot c = ac + bc$.

4 Experiments

We performed experiments to extract common SOP expressions, using the scheme of concurrent extraction of double-cube divisors and double-literal cubes. The original method [7] has two advantages: 1) complement divisors (e.g. $a + b$ versus $a'b'$) can be recognized and 2) all double-cube divisors are calculated at the first time by taking all pairs of cubes and making them cube-free, and they can be partially updated.

We, however, could not take the second advantage because it was hard to identify each cube in an implicit cube set representation. Instead, we generated a subset of all double-cube divisors from implicitly represented kernels as follows:

1. Generate all kernels for given SOP expressions using the method proposed in Section 3.
2. For a kernel without multiple literals (a level-0 kernel), make all pairs of cubes in the kernel.
3. For a kernel with multiple literals, deduct cubes having multiple literals, and make all pairs of cubes in the resultant expression.

For instance, for a kernel $a+b+c$, we generated $a+b$, $a+c$ and $b + c$. For a kernel $ac + ad + bc + bd + e + f + g$, we generated $e + f$, $e + g$ and $f + g$, and did not make some pairs like $ac + bd$.

Table 1 shows the experimental results for several logic circuits [8] shown in the column "Circuit". From a circuit description, we constructed BDDs representing the functions of primary outputs, and generated SOP expressions from the functions using the method proposed in [9]. The column "Initial" corresponds to the initial expressions be-

Table 1. Experimental results

Circuit			Initial		Final	Time	Hash		
name	in	out	literal	co-kernel / node	literal	(sec)	hit /	look-up =	ratio
alu4	14	8	4961	16096 / 14646	1256	10.40	33625 /	83168 =	0.404
apex1	45	45	7017	6178 / 5558	1759	6.18	25595 /	46326 =	0.552
apex2	39	3	14728	14410 / 5042	358	4.91	10157 /	22651 =	0.448
apex3	54	50	4621	4888 / 4822	1718	4.05	13421 /	30405 =	0.441
apex4	9	19	7887	22235 / 21568	2598	11.25	50441 /	121497 =	0.415
apex5	117	88	7202	1321 / 946	968	2.42	4443 /	10103 =	0.440
cordic	23	2	18213	105654 / 8727	88	1.28	7976 /	17725 =	0.450
cps	24	109	6747	1458 / 1276	1415	3.36	8081 /	15088 =	0.536
dalu	75	16	24902	24976 / 9941	696	5.46	17325 /	36094 =	0.480
ex1010	10	10	4048	5891 / 5838	1892	4.88	24121 /	44922 =	0.537
frg2	143	139	27803	9212 / 1603	893	2.76	9494 /	14915 =	0.636
misex3	14	14	12101	17195 / 13185	1113	8.92	30346 /	79301 =	0.383
pair	173	137	130724	29796 / 7909	3126	37.67	85739 /	155161 =	0.553
parity	16	1	524288	42456897 / 131037	60	75.93	294292 /	523375 =	0.562
seq	41	35	17876	11615 / 8299	1972	12.14	41976 /	88383 =	0.475
spla	16	46	4608	2242 / 1436	634	1.75	3366 /	8509 =	0.396
t481	16	1	4752	4746 / 698	40	0.75	2522 /	4056 =	0.622
table3	14	14	5805	14988 / 14637	1265	9.82	22475 /	76906 =	0.292
table5	17	15	6319	14090 / 13195	1045	9.88	18820 /	63691 =	0.295

fore the extraction. Each sub-column “literal” shows the number of literals in SOP form. The sub-column “co-kernel” shows the number of all co-kernels, and “node” shows the number of nodes used in the kernel graph at that time.

The result of logic extraction is shown in the column “Final”. The column “Time” gives the CPU time in seconds on a 450MHz Pentium II system running the Linux operating system. The column “Hash” gives statistics of hash tables used in the algorithm for generating kernel graphs. The sub-column “look-up” shows the number of look-ups, and “hit” shows the number of times that a required node had already existed.

We successfully extracted common logic expressions from very large cube sets using implicit representations and a kernel graph. There are some cases where many co-kernels can be represented with considerably less nodes. In particular, our method can represent all of the co-kernels of a 16-bit parity function (“parity”), which seem very hard to be represented with other methods. The hit ratios of the hash tables were sufficiently high for most circuits, and this indicated the usefulness of memorizing the kernel generation process.

5 Conclusions

We have proposed a kernel graph, where the kernel generation process is memorized. The main advantages are that 1) the execution time can be reduced by skipping the process that was processed before and 2) the memory usage for representing kernel/co-kernel relations can be reduced by sharing nodes. In future work, we want to integrate Boolean divisions into our method.

References

[1] R. K. Brayton and C.McMullen. The Decomposition and Factorization of Boolean Expressions. In *Proc. Int'l Symp. Circuits and Systems*, pages 49–54, May 1982.

[2] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Trans. CAD*, CAD-6(6):1062–1081, November 1987.

[3] O. Coudert and J. C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proc. Design Automation Conf.*, pages 36–39, June 1992.

[4] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. Design Automation Conf.*, pages 272–277, June 1993.

[5] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, C-35(8):667–691, August 1986.

[6] S. Minato. Fast Factorization Method for Implicit Cube Set Representation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 15(4):377–384, April 1996.

[7] J. Rajski and J. Vasudevamurthy. The Testability-Preserving Concurrent Decomposition and Factorization of Boolean Expressions. *IEEE Trans. CAD*, 11(6):778–793, June 1992.

[8] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0.* MCNC, January 1991.

[9] S. Minato. Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams. *IEICE Trans. on Fundamentals*, E76-A(6):967–973, June 1993.