Fast Multiplication of the Algebraic Normal Forms of Two Boolean Functions

Subhabrata Samajder   Palash Sarkar

17th April, 2013

1 Introduction

2 Our Results

3 Conclusion

Motivation

- Multiplication of Boolean functions is a basic operation and is of interest in itself.
- Buchbergers algorithm and its improvements F_4 and F_5.
- Algebraic immunity.
- Non-linear Codes such as Reed-Muller Codes and Kerdock Codes.

Goal

- Multiplication of two *sparse* polynomials p and q having l_p and l_q terms each will have about $l_q l_q$ terms and so the usual algorithm which takes $\mathcal{O}(l_p l_q)$ time, is optimal.
- It is interesting to investigate whether this can be improved in case of *dense* polynomials, when the number of variables is at most 30 or so.

1 Introduction

2 Our Results

3 Conclusion

Basic Idea

Let, $\mathbb{R} = GF(2)\left[x_1, x_2, \ldots, x_n\right] / \left\langle x_1^2 - x_1, \ldots, x_n^2 - x_n \right\rangle$ and $p\left(x_1, \ldots, x_n\right), q\left(x_1, \ldots, x_n\right) \in \mathbb{R}$. Write,

$$\begin{aligned} p\left(x_1, \ldots, x_n\right) &= x_n \cdot p_1\left(x_1, \ldots, x_{n-1}\right) \oplus p_0\left(x_1, \ldots, x_{n-1}\right) \\ q\left(x_1, \ldots, x_n\right) &= x_n \cdot q_1\left(x_1, \ldots, x_{n-1}\right) \oplus q_0\left(x_1, \ldots, x_{n-1}\right). \end{aligned}$$

Then,

$$\begin{aligned} pq &= \left(p_1 q_1\right) x_n^2 \oplus \left(p_1 q_0 \oplus p_0 q_1\right) x_n \oplus p_0 q_0 \\ &= \left(p_1 q_1 \oplus p_1 q_0 \oplus p_0 q_1\right) x_n \oplus p_0 q_0; \qquad \left[\text{Since, } x_n^2 = x_n \text{ in } \mathbb{R}.\right] \\ &= \left\{\left(p_1 \oplus p_0\right)\left(q_1 \oplus q_0\right) \oplus p_0 q_0\right\} x_n \oplus p_0 q_0. \end{aligned}$$

Thus, the number of $(n-1)$-variate multiplications required is 2 instead of 4 at the cost of one extra addition.

Complexity Analysis

Let,

- $t(n)$ denote the time taken to multiply two n-variate polynomials.
- $e(n)$ denote the time taken to add two n-variate polynomial.

Then,

$$t(n) = 2t(n-1) + 4e(n-1).$$

Solving,

$$t(n) = 2^n t(0) + 4 \times \left\{ e(n-1) + 2 \times e(n-2) + 2^2 \times e(n-3) + \ldots + 2^{n-2} \times e(1) + 2^{n-1} \times e(0) \right\}.$$

Since, $e(n) = 2^n \cdot e(0)$, using this we get,

$$t(n) = 2^n t(0) + 4n2^{n-1} e(0),$$

where, $t(0)$ and $e(0)$ denote the time taken for bit-wise AND and XOR.

Cmplexity Analysis (Cont.)

Therefore,

$$t(n) = \mathcal{O}(n2^n) = \mathcal{O}(2^{n+\log_2 n}) = \mathcal{O}(m \log_2 m),$$

where $m = 2^n$.

- This simple observation leads to an $\mathcal{O}(n2^n)$ time recursive algorithm.
- Notice that, in "dense" polynomials, the size of the input polynomials will be about $\mathcal{O}(m)$ and so this $\mathcal{O}(m \log_2 m)$ algorithm is very attractive.
- Asymptotically, this is competitive with general purpose Fourier transform based multivariate polynomial multiplication algorithm specialized to the binary case.

Polynomial Representation

- Polynomials in $\mathbb{R}$ are representated using a sequence of bits.
- Presence of every monomial is denoted by a single bit.
- Thus 2^n bits are used to represent any n-variable polynomial in $\mathbb{R}$.

An Iterative Algorithm

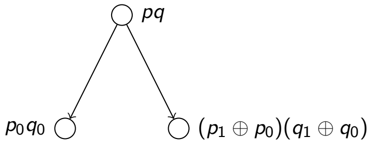

Figure: Figure depicting the basic recursion step.

- Notice that, one can compute the values of p_0, $(p_0 \oplus p_1)$, q_0 and $(q_0 \oplus q_1)$ independently and then multiply them to get the required p_0q_0 and $(p_0 \oplus p_1) \cdot (q_0 \oplus q_1)$.
- Using this idea recursively, we get two recursive tree (one each for p and q).

An Iterative Algorithm (Cont.)

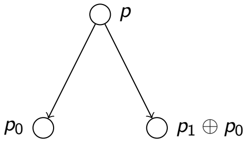

Figure: Figure depicting how each polynomials must be split.

- Then, p_0 corresponds to the first 2^{n-1} bits (LSB's) of p and p_1 the last 2^{n-1} bits (MSB's) of p.
- Hence, $p_0 \oplus p_1$ is nothing but bit-wise XOR of the 1^{st} half with the 2^{nd} half of A.
- This is repeated until $n = 1$.

An Iterative Algorithm (Cont.)

- Thus, two such trees are formed, one each for p and q.
- In the leaf, level multiplication is equivalent to bit-wise AND-ing.
- To get the final result pq, we traverse upwards from the leaves to the root by doing similar kind of operations.

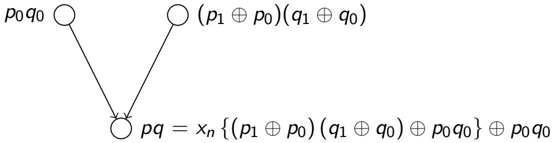
p_0q_0
$(p_1 \oplus p_0)(q_1 \oplus q_0)$
$pq = x_n\{(p_1 \oplus p_0)(q_1 \oplus q_0) \oplus p_0q_0\} \oplus p_0q_0$

Figure: Figure depicting the basic recursion step while returning back.

- Notice that the second XOR corresponds to concatenation, which is our case comes free.

An Iterative Algorithm (Cont.)

Thus, the iterative algorithm has the following subroutines :

1. PRE_PROCESS of p.
2. PRE_PROCESS of q.
3. Bitwise AND-ing of the leaves of the corresponding trees of p and q, respectively.
4. POST_PROCESS.

An Iterative Algorithm (Cont.)

Suppose, we want to multiply two 4-variable polynomial

$$p(x_1, x_2, x_3, x_4) = x_1 \oplus x_1x_2 \oplus x_2x_3 \oplus x_1x_2x_3 \oplus x_4 \oplus x_1x_2x_4 \oplus \\ x_1x_3x_4 \oplus x_2x_3x_4 \oplus x_1x_2x_3x_4$$

and

$$q(x_1, x_2, x_3, x_4) = x_2 \oplus x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_1x_2x_3 \oplus x_4 \oplus x_1x_2x_4.$$

Multiplying by hand, one can easily see that

$$pq = x_2x_3 \oplus x_1x_2x_3 \oplus x_4 \oplus x_1x_4 \oplus x_2x_4 \oplus x_1x_2x_4 \oplus x_3x_4.$$

An Iterative Algorithm (Cont.)

- The byte representation of p is 01010011 $(= 202)$, 10010111 $(= 233)$.
- And the byte representaton of q is 00101111 $(= 244)$, 10010000 $(= 9)$, where the leftmost bit entry denotes the LSB.
- Therefore, the byte representation of pq is 00000011 $(= 196)$, 11111000 $(= 31)$.

PRE_PROCESS of p

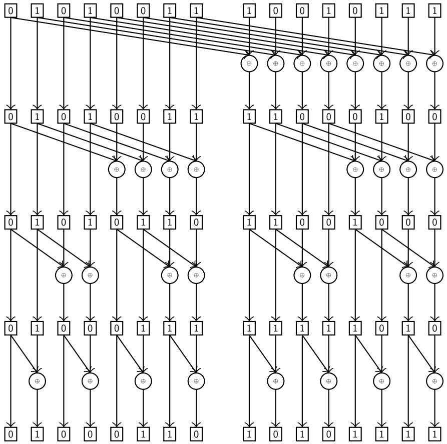

Figure: Figure depicting the PRE_PROCESS step for a 4−variate polynomial p.

PRE_PROCESS of q

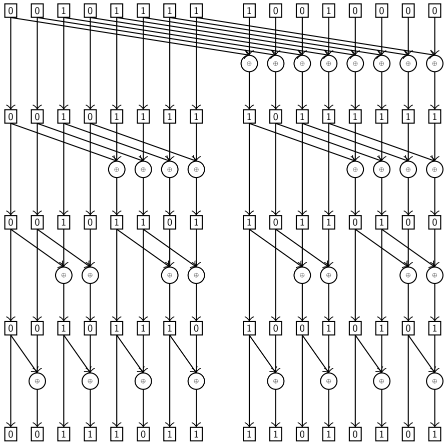

Figure: Figure depicting the PRE_PROCESS step for a 4−variate polynomial q.

POST_PROCESS

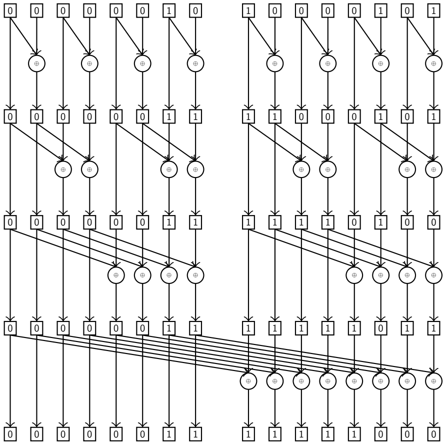

Figure: Figure depicting the POST_PROCESS step for a 4−variate polynomial and the final result pq.

Using Table Look-ups

- Extracting a bit from a byte is costly.
- Hence, we use table - lookups.
- Instead of going all the way down to the n^{th} level, we stop at level $n - \beta$.
- Use table lookups to perform multiplication of two β-variable polynomials.
- The value of β is taken to be 3, because the table corresponding to $\beta = 4$ becomes very large.
- We thus pack the polynomials p and q in byte arrays and use byte level XOR to multiply them.

An Iterative Algorithm using Table Look-ups

1. PRE_PROCESS of p till $n = 3$.
2. PRE_PROCESS of q till $n = 3$.
3. Table Look-ups to multiply two 3-variable polynomials at once.
4. POST_PROCESS.

Further Improvements

- One may use w-bit XOR instead of 8-bit, assuming the architecture allows w-bit word arithmetic, where $w = 2^k, k \geq 3$.
- The motivation is to save on the number of 8-bit XOR's.
- Thus, using one w-bit XOR, one can save $2^{\log_2 w - 3}$ many XOR's.
- However, doing it this way one can only go up to $n - \log_2 w$ level.
- Hence using w-bit words, involves, an additional task of UNPACKING and PACKING the w-bit word into bytes so that one can use the 8-bit table lookup.

1 Introduction

2 Our Results
- A w-bit Non-recursive Algorithm
- Experimental Results
- Multiplying Sparse Polynomials

3 Conclusion

A w-bit Non-recursive Algorithm

1. PRE_PROCESS of p till $n = \log_2 w$.
2. PRE_PROCESS of q till $n = \log_2 w$.
3. UNPACK p.
4. UNPACK q
5. EXTRACT_AND_LOOKUP.
6. PACK.
7. POST_PROCESS.

UNPACK

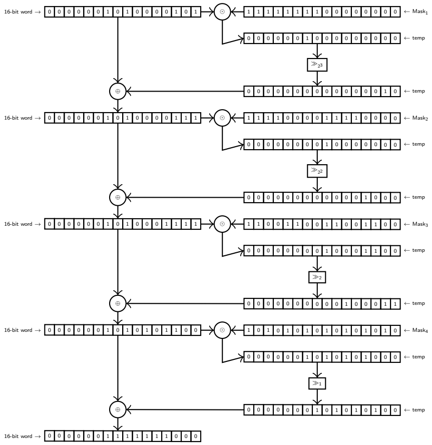

Figure: Figure depicting the UNPACKING step for a 4-bit word.

EXTRACT

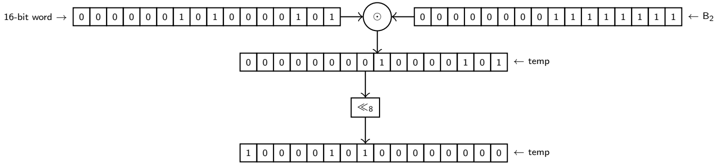

Figure: Figure depicting the extraction step for a 4-bit word.

Cost Analysis

1. $2^{\log_2 w - 3} \cdot 2^{n - \log_2 w} = 2^{n-3}$ 8-bit table look-ups.
2. $2 \cdot 2^{n - \log_2 w} \cdot 2^{\log_2 w - 3} = 2^{n-2}$ w-bit operations for table look-ups.
3. $2^{n - \log_2 w} \cdot (3 \cdot (3 \cdot (\log_2 w - 3))) = 9 \cdot (\log_2 w - 3) \cdot 2^{n - \log_2 w}$ w-bit operations for PACKING and UNPACKING.
4. $3 \cdot (n - \log_2 w) \cdot 2^{n - \log_2 w - 1}$ w-bit XOR's for the PRE_PROCESS and POST_PROCESS.

1 Introduction

2 Our Results
- A w-bit Non-recursive Algorithm
- Experimental Results
- Multiplying Sparse Polynomials

3 Conclusion

Comparison with MultANF_8, MultANF_{32} and MultANF_{64}

n	Average Cycles for 8 bit	Average Cycles for 32 bit	Speedup of 32-bit w.r.t 8 bit	Average Cycles for 64 bit	Speedup of 64-bit w.r.t 8-bit	Speedup of 64-bit w.r.t 32-bit
6	498.53	121.01	4.12	92.73	5.38	1.31
7	1138.23	428.95	2.65	199.38	5.71	2.15
8	2273.35	1032.89	2.20	1022.83	2.22	1.01
9	5013.86	1853.20	2.71	1276.61	3.93	1.45
10	11055.29	3871.94	2.86	2437.25	4.54	1.59
11	23608.47	8357.06	2.83	6010.26	3.93	1.39
12	34680.06	7711.84	4.50	5341.51	6.50	1.44
13	53976.73	16093.17	3.35	11153.91	4.84	1.44
14	103962.07	34223.26	3.04	23296.39	4.46	1.47

Table: Table showing the speed (in cycles) comparisons between 8-bit, 32-bit and 64-bit implementations.

Comparison with MultANF$_8$, MultANF$_{32}$ and MultANF$_{64}$ (Cont.)

n	Average Cycles for 8 bit	Average Cycles for 32 bit	Speedup of 32-bit w.r.t 8 bit	Average Cycles for 64 bit	Speedup of 64-bit w.r.t 8-bit	Speedup of 64-bit w.r.t 32-bit
15	221928.42	73352.13	3.03	49992.79	4.44	1.47
16	466755.57	153265.65	3.05	101450.16	4.60	1.51
17	1014411.71	321682.42	3.15	212650.40	4.77	1.51
18	2075710.70	681210.39	3.05	441465.78	4.70	1.54
19	4401203.98	1433646.38	3.07	915821.38	4.81	1.57
20	9786430.84	3132142.40	3.13	2500430.46	3.91	1.25
21	20418478.40	6441914.73	3.17	5112594.99	3.99	1.26
22	43212647.62	13552823.50	3.19	10629153.25	4.07	1.28
23	89719530.45	28183683.11	3.18	21806265.54	4.11	1.29

Table: Table showing the speed (in cycles) comparisons between 8-bit, 32-bit and 64-bit implementations.

Comparison with MultANF_8, MultANF_{32} and MultANF_{64} (Cont.)

n	**Average Cycles for 8 bit**	**Average Cycles for 32 bit**	**Speedup of 32-bit w.r.t 8 bit**	**Average Cycles for 64 bit**	**Speedup of 64-bit w.r.t 8-bit**	**Speedup of 64-bit w.r.t 32-bit**
24	190141764.33	59136263.78	3.22	45559914.11	4.17	1.30
25	401052397.73	130650693.03	3.07	106224818.55	3.78	1.23
26	838518978.22	299963811.34	2.80	272976258.05	3.07	1.10
27	1759215397.18	646245016.94	2.72	600701064.94	2.93	1.08
28	3635571731.89	1323794840.80	2.75	1239783643.15	2.93	1.07
29	7543793814.89	2735720452.18	2.76	2541063909.56	2.97	1.08
30	15606584912.85	5572652029.49	2.80	5109022401.64	3.06	1.09

Table: Table showing the speed (in cycles) comparisons between 8-bit, 32-bit and 64-bit implementations.

Comparison of MultANF$_8$ with SAGE

n	MultANF$_8$	sage
3	0.80 ns	94773.05 ns
4	1.84 ns	127928.97 ns
5	55.55 ns	197319.98 ns
6	70.78 ns	354038.95 ns
7	161.31 ns	762128.12 ns
8	718.90 ns	1700400.83 ns
9	799.88 ns	3205805.06 ns
10	1644.70 ns	7070338.01 ns
11	7151.90 ns	14413833.62 ns
12	15372.56 ns	32285171.03 ns
13	18514.16 ns	69974661.11 ns
14	36287.44 ns	162460117.1 ns
15	77486.74 ns	336447609.9 ns

Table: Comprison with SAGE. In each case, the timings are averaged over 1000 runs.

1 Introduction

2 Our Results
- A w-bit Non-recursive Algorithm
- Experimental Results
- Multiplying Sparse Polynomials

3 Conclusion

Sparse Impelmentation

- A monomial is represented by a w-bit word, where w is the minimum machine word such that $2^n \leq w$.
- The two polynomials are given as two arrays A and B of w-bit words.
- Multiplication of two monomials corresponds to the bit-wise OR of the corresponding w-bit words.
- For sparse implementation, we take the input arrays A and B and OR every element of array A with that of array B, and store them in another array C.
- The array C is then sorted using a non recursive (the process stack is simulated internally) implementation of randomized quick sort.
- Repetitions are removed by either deleting the monomial (if its number of repetitions is even) or replacing all the entries by just one entry (if the number of repetitions is odd).

Comparison with SAGE

- The experimental results show that the algorithm used by SAGE is slower than the quadratic implementation.
- It seems that the SAGE algorithm depends both on the sizes of A and B (i. e., l_p and l_q) and the number of variables involved.
- But our sparse implementation only depends on l_p and l_q.
- For example to multiply two polynomials each with 1000 monomials SAGE took 7.43 seconds for $n = 30$ and 34 seconds for $n = 63$, whereas the quadratic implementation took 0.17 seconds for both $n = 30$ and $n = 63$.

Comparison with MultANF_w

- Experimentally it was found that, if $l_p l_q < 2^{n-\alpha}$, then the quadratic algorithm performs better than $\text{MultANF}_{2^\alpha}$, where $\alpha = 3, 5, 6$.

1 Introduction

2 Our Results

3 Conclusion

Conclusion

- We have proposed a new non-recursive algorithm MultANF_w, which multiplies two Boolean functions in their ANF's.
- It tries to use the w-bit word arithmetic, if the architecture supports it.
- With this in mind, three variants of MultANF_w were proposed for $w = 8, 32$ and 64.
- It was shown that the 64-bit implementation is better than the other two.
- A comparison study of MultANF_w with a sparse implementation tells us, when one should switch from the sparse implementation to the dense implementation, i.e., MultANF_w.
- Lastly, a comparison between our implementations (sparse and dense implementations) with that of the software package SAGE shows that, our implementations are faster than SAGE.

Thank You!