Performance Evaluation of Quine-McCluskey Method on Multi-core CPU

Hoang-Gia Vu
Faculty of Radio-Electronic Engineering
Le Quy Don Technical University
Ha Noi, Vietnam
giavh@lqdtu.edu.vn

Ngoc-Dai Bui
Faculty of Radio-Electronic Engineering
Le Quy Don Technical University
Ha Noi, Vietnam

Anh-Tu Nguyen
Faculty of Radio-Electronic Engineering
Le Quy Don Technical University
Ha Noi, Vietnam

ThanhBangLe
Faculty of Radio-Electronic Engineering
Le Quy Don Technical University
Ha Noi, Vietnam

***Abstract*—The Quine-McCluskey method is an algorithm to minimize Boolean functions. Although the method can be programmed on computers, it takes a long time to return the set of prime implicants, thus slowing the analysis and design of digital logic circuits. As a result, it slows down the dynamic reconfiguration process of programmable logic devices. In this paper, we first propose a data representation for storing implicants in memory to reduce the cache misses of the program. We then propose an algorithm to find all prime implicants of a Boolean function. The algorithm aims to reuse the data available on cache, thus decreasing cache misses. After that, we propose an algorithm for step 2 of the Quine-McCluskey method to select the minimal number of essential prime implicants. The evaluation shows that our proposals achieve much higher performance than the original Quine-McCluskey method. The number of essential prime implicants is a low percentage, less than 50%, of the total prime implicants generated in step 1 of the method.**

Keywords—Quine-McCluskey, prime implicant, multithreading, Boolean function

I. Introduction

A Boolean function is a function producing a Boolean output by logical calculation of Boolean inputs. It is a key point in the analysis, design, and implementation of digital logic circuits. Minimization of Boolean functions is to optimize the algorithm of such functions to achieve a simpler structure of the algorithm. Thus it simplifies the corresponding digital logic circuit. There are two popular methods to minimize Boolean functions. 1) The Karnaugh method is based on a graphical representation of Boolean functions [1]. And 2) The Quine-McCluskey method generates prime implicant lists using the tabulation method [2]. The method was first proposed by Quine [3, 4] and then improved by McCluskey. The Quine-McCluskey method is functionally identical to the Karnaugh method. However, while the Karnaugh mapping is suitable to Boolean functions of a few input variables, the Quine-McCluskey algorithm is dedicated to Boolean functions with a large number of Boolean inputs [5]. Therefore, the Karnaugh method is often used in education. Meanwhile, the Quine-McCluskey method is practically employed in the analysis and design of digital logic circuits for real-world applications.

However, the computational complexity of the Quine-McCluskey method is $O(N^{log_2 3} log_2 N)$, N - the input length [6]. The run-time of the method also grows exponentially with the input variable number. This slows down the process of analysis, design, and verification of digital logic circuits. The problem becomes more serious in the design of dynamic run-time reconfigurable hardware architectures or adapted hardware architectures.

The Quine-McCluskey method consists of two steps:

Step 1: Finding all prime implicants of the Boolean function

Step 2: Selection of essential prime implicants that cover all the minterms of the function.

Both of the steps are memory-intensive applications since they involve many repeated memory references. We believe that both of the steps can be accelerated on cached CPU by exploiting the temporal and spatial data locality. The contributions of this paper are as follows:

1) We propose a bitarray-based data representation for implicants that consume a small size of memory. This helps to reduce the cache miss rate of the method running on the CPU.

2) We propose an algorithm for step 1 of the method to exploit data locality, thus minimizing the cache misses of the step running on the CPU.

3) We propose an algorithm for step 2 of the method to minimize the number of required prime implicants covering all the minterms of the Boolean function.

The rest of the manuscript is organized as follows: Section II discusses the related work. Section III presents the data representation of implicants. Section IV presents the proposed algorithms of the method. Section V shows the evaluation. The conclusion is summarized in section VI.

II. Related Work

Wegener et al. proved that the minimization of Boolean functions is a hard problem [7]. Prasad et al. analyzed the simplification of the Quine-McCluskey method for a different number of product terms [8]. The complexity of the Quine-McCluskey method was mathematically modelled in the following equation [8]:

$$N = a\,.\,t^{b}.e^{-tc} + 1 \qquad (1)$$

Where,

N : the number of literals

t : the number of non-repeating product terms in the Boolean function

a, b, and c : three constants depending on the number of input variables

For improving the minimization of Boolean functions, there were several works to quickly and automatically simplify Boolean functions. Dusa et al. proposed eQMC to reduce the computational complexity of the Quine-McCluskey method by taking into account only minterms that the corresponding outputs are '1' [9]. The eQMC method performs an exhaustive procedure that relies on index vectors instead of complex matrices. The method achieved higher performance and smaller memory usage compared to their implementation of the original Quine-McCluskey method. However, the performance was still low at 4.87 seconds, execution time with 15 input variables and only 20 observed configurations. Gurunath et al. introduced an algorithm for multiple output minimization [10]. Jain et al. [11] optimized the Quine-McCluskey method by introducing the concept of Reduced Mask. The new concept helped to reduce the computational complexity of the method. As a result, the execution time decreased significantly. Majumder et al. presented a technique based on decimal values to decrease the probability of an error occurrence [12].

There are a couple of works for the acceleration of the Quine-McCluskey method. Siladi et al. proposed a scheme to adapt step 1 of the Quine-McCluskey method on a parallel computing platform – GPU [13, 14]. In this works, the author presented a parallel algorithm for simplification of step 1 on the GPU. They process the implicants in multiple rounds. In each round of step 1, implicants are first partitioned by the positions of dashes in the terms. Each partition is then scanned for merge-able terms. In their merging algorithm, the list of terms is first converted to bitmap representation, thus each term is a bit set. Dashes in the term are treated as zeroes when calculating the bit indexes. The algorithm achieved not much higher performance compared to the implementation on the CPU for large numbers of input variables. Small instances running on the GPU is even slower than those running on the CPU. In these works, they did not take into account the output value '*x*' – *don't care* of the Boolean function.

III. BITARRAY-BASED DATA REPRESENTATION

To reduce the memory usage and the cache misses of the Quine-McCluskey method, we propose to represent implicants in form of bit arrays instead of ASCII characters. Particularly, symbols '1', '0', '-' are represented as follows:

Symbol '1' → bit array '01'

Symbol '0' → bit array '00'

Symbol '-' → bit array '10'

As a result, implicant (10-00-10) will be represented in the following form:

Algorithm 1: Comparison of two bit arrays

```
1:  def  compare_2(a, b)
2:         temp = a ^ b
3:         if  temp.count(1) == 1:
4:             return  temp.find(1)
5:         else:
6:             return -1
```

(10-00-10) → bit array '0100100000100100'

The implicant (10-00-10) consumes 8 bytes if it is represented in the form of ASCII characters. The implicant uses 2 bytes (16 bits) if represented in a bit array. Therefore, the memory utilization of the bitarray-based representation is four times efficient than that of the ASCII-based representation.

For comparison between two bit arrays to find out if they can be combined into a new implicant, we propose to use the operator XOR as in Algorithm 1. If the two implicants differ in only one symbol, then the XOR operation of two corresponding bit arrays will return a bit array including only one bit '1'. It is noted that the comparison is only for implicants having the same number of dashes. In the function, the XOR operation of the two bit arrays is first executed, followed by counting the number of bits '1' in the result. If the number of bit '1' is equal to one, the position of the bit '1' is returned. Otherwise '-1' will be returned.

IV. THE ALGORITHMS FOR THE QUINE-MCCLUSKEY METHOD

In this paper, we focus on optimizing both step 1 and step 2 of the Quine-McCluskey method.

A. Algorithm for Step 1: Finding all Prime Implicants

In step 1 of the Quine-McCluskey method, the major operations are memory access to read all the implicants and comparison of implicants. We believe that the performance bottleneck in this step is the huge number of memory references repeated again and again. In this part, we propose an algorithm to exploit temporal data locality on cache memory. The Pseudo code is described in Algorithm 2.

• Let *m* be the number of input variables in the Boolean function.

• Let *list-1* be the list of all minterms that the corresponding output is evaluated to '1'.

• Let *list-x* be the list of all minterms that the corresponding output is evaluated to 'x' – don't care.

• Let *implicant-list* be the list of all implicants in each round of comparison.

• Let *prime-list* be the list of all prime implicants found out after step 1 of the method.

• Let *new-implicant-list* be the new imlicant list generated after each round of comparison and merging.

• Let *combined* be the Boolean variable indicating if there is any combination between two implicants. It starts at *True* value.

In the *while* loop, *combined* is first assigned to *False*. Then all the implicants are classified as in Line 10. Implicants belonging to the same group have the same number of symbols

Algorithm 2: Finding all prime implicants

```
1:  m = # input variables
2:  list-1 = [minterms]
3:  list-x = [d-terms]
4:  implicant-list = list-1.append(list-x)
5:  prime-list = []
6:  new-implicant-list = []
7:  combined = True
8:  while (combined):
9:      combined = False
10:     groups = make_groups(implicant_list)
11:     for i in range(m):
12:         reversed = False
13:         for x1 in groups[i]:
14:             if reversed:
15:                 range-litst = groups[i+1].reverse()
16:             else:
17:                 range-litst = groups[i+1]
18:             for x2 in range-list:
19:                 pos = compare_2(x1, x2)
20:                 if pos != -1:
21:                     combined = True
22:                     new-term = x1
23:                     new-term[pos -1] = 1
24:                     new-term[pos] = 0
25:                     x1.used = True
26:                     x2.used = True
27:                     new-implicant-list.append(new-term)
28:             reversed = not reversed
29:     for impl in implicant-list:
30:         if impl.used:
31:             implicant-list.remove(impl)
32:     prime-list.append(implicant-list)
33:     implicant-list = new-implicant-list
```

'1'. Therefore, there are at most m+1 groups. Implicants in each group are then compared with the implicants of the consecutive group to find if they can be merged into a new implicant as in Line 11 to Line 19. If there is any combination, the variable *combined* is marked as *True* in Line 21. That means at least one new implicant is generated, and the next round of the *while* loop is required. At the same time, the two combined implicants are marked as used.

It is noted that in the first *for* loop, a variable named *reversed* is used to indicate the third *for* loop should be iterated in the reversed order or the normal order. This variable starts at *False* as in Line 12. Iterating the loop in the reversed order helps to utilize the data in cache memory that are fetched recently from the lower-level memory before the data are replaced on the cache memory. As a result, the cache miss rate will be decreased. That is also the key point in Algorithm 2.

All the marked-as-used implicants are then removed from the *implicant-list* before the list is updated into the *prime-list* as in Line 32. After that, the *implicant-list* is assigned to the *new-implicant-list* before going to the next round of the *while* loop.

Algorithm 3: Selection of Essential Prime Implicants

```
1:  final-prime-list = []
2:  victim = None
3:  max-len = 1
4:  while max-len != 0:
5:      max-len = 0
6:      for prime in prime-list:
7:          prime.val-1.difference_update(victim.val-1)
8:          if len(prime.val-1) > max-len:
9:              max-len = len(prime.val-1)
10:             temp = prime
11:     prime-list.remove(temp)
12:     victim = temp
13:     final-prime-list.append(temp)
```

B. *Algorithm for Step 2: Selection of Essential Prime Implicants*

In step 2 of the Quine-McCluskey method, essential prime implicants will be selected among the full set of prime implicants from step 1. In this step, we aim to choose the smallest number of prime implicants that cover all the minterms of the Boolean function. For that perspective, we propose Pseudo code as in Algorithm 3. After step 1 of the Quine-McCluskey method, we have *prime-list* including all prime implicants of the Boolean function.

• Let *final-prime-list* be the final list of essential prime implicants covering all the minterms of the function.

• Let *victim* be the prime that is selected after each round of the *while* loop. *victim* is the prime implicant that is merged from the largest number of minterms.

• Let *max-len* be the maximum length of all minterm sets of prime implicants.

• Let *val-1* be the minterm set of a prime implicant.

In the *while* loop, *max-len* is assigned to zero in Line 5. *max-len* is found by iterating *prime-list* and comparing the length of the minterm sets of all prime implicants. It is noted that after finding out a victim in each round of the while loop, the victim is removed from *prime-list* as in Line 11. Then, the minterm set of each prime implicants is also updated as in Line 7 before finding *max-len*.

V. EVALUATION

Table 1 shows the experimental setup we used to evaluate the proposed algorithms. In this evaluation, the number of input variables in the Boolean function is scaled from 10 to 20. For the k-input Boolean function, the number of possible input vectors is 2^k , which evaluate to '0', '1', or 'x' – *don't care*.

• Let *fill-factor-1* be the ratio between the number of input vectors that evaluate to '1' and the total number of input vectors 2^k.

TABLE I. EVALUATION SETUP

CPU	Intel Core i3-7100
Number of cores	4
CPU operation frequency	800 MHz
L1 Dcache	32 KB
L1 Icache	32 KB
L2 cache	256 KB
L3 cache	3 MB
Block cache size	64 Bytes
Operating system	Ubuntu 18.4
Cache profiling tool	Perf 5.4.143

TABLE II. CACHE MISS FOR DIFFERENT FILL FACTORS

fill-factor	L1-load-misses (Original)	Miss-rate	L1-load-misses (Our proposal)	Miss-rate
0.1	3,612,183	4.12%	3,363,111	3.45%
0.2	4,194,506	0.54%	4,153,117	1.15%
0.3	45,486,911	0.15%	36,471,029	0.37%
0.4	22,437,678,570	0.12%	19,667,227,871	0.35%

TABLE III. CACHE MISS FOR DIFFERENT NUMBER OF INPUT VARIABLES

Inputs	L1-load-misses (Original)	Miss-rate	L1-load-misses (Our proposal)	Miss-rate
10	3,612,183	4.12%	3,363,111	3.45%
15	203,862,212	0.21%	166,546,860	0.55%
20	321,717,266,536	0.19%	225,112,520,565	0.52%

• Let *fill-factor-x* be the ratio between the number of input vectors that evaluate to '*x*' and the total number of input vectors - 2^k.

In this section, we describe the cache performance, execution time, and the number of essential prime implicants in our algorithms. For the number of input variables 10, we scale the *fill-factor-1 and fill-factor-x from 0.1 to 0.4.*

A. Cache Performance

Table II shows the cache load misses of L1 data cache and the corresponding miss rate for the original Quine-McCluskey method and for our proposal evaluated in a Boolean function with 10 input variables while scaling the fill factor. As can be seen from the table, the number of L1 cache load misses in our proposal is quite smaller than that of the original Quine-McCluskey method. The reduced number of cache load misses is a consequence of the proposed small data representation described in Section III and the algorithm presented in Section IV.A. The same situation is in Table III when the number of input variables is scaled and the fill factor is fixed at 0.1. Although the miss rate is basically higher in our proposal, except for the function with 10 input variables and the fill factor at 0.1, the larger number of L1 data cache load misses will cause the longer total miss penalty in the original Quine-McCluskey method. As a result, the execution time in our proposal is expected shorter than that of the original Quine-McCluskey.

TABLE IV. EXECUTION TIME OF STEP 1 AND STEP 2 FOR DIFFERENT FILL FACTOR

fill-factor	Step 1 (Original)	Step 1 (Our proposal)	Step 1 (Multi-thread)	Step 2
0.1	0.0379	0.0043	0.0057	0.0014
0.2	0.1806	0.0641	0.0588	0.0116
0.3	7.2595	2.4564	2.0784	0.1597
0.4	4,301.5054	1,453.5304	1,235.8151	59.5760

TABLE V. EXECUTION TIME OF STEP 1 AND STEP 2 FOR DIFFERENT NUMBER OF INPUT VARIABLES

Inputs	Step 1 (Original)	Step 1 (Our proposal)	Step 1 (Multi-thread)	Step 2
10	0.0379	0.0043	0.0057	0.0014
15	22.3329	6.0953	5.1162	2.1205
20	41629.7808	8994.4525	7503.7638	3101.154

B. Execution Time

Table IV shows the execution time of step 2 and step 1 of the Quine-McCluskey method for different fill factors. In this experiment, we executed step 1 in the original method in a single core. Then step 1 was executed with our proposal in Section III and Section IV.A in a single core. After that step 1 with our proposal was executed in multiple threads running in the multi-core CPU. The results show that our proposal achieved much higher performance than the original Quine-McCluskey did. The same situation is presented in Table V for scaling the number of input variables. This higher performance in our proposal comes from the smaller numbers of cache load misses revealed in Section V.A.

As can be seen in the two tables, the method with our proposal running in multiple threads achieved not much higher performance compared to running on a single thread, apart from the experiment with 10 input variables and the fill factor 0.1. This is because the Quine-McCluskey method is a memory-intensive application. The majority of the execution time is consumed by the memory access, but not the computation in the CPU cores. The two tables also show that the execution time of step 2 is quite smaller than that of step 1. However, it is noted that the proposed algorithm for step 2 do not focus on improving the performance, but focus on reducing the number of essential prime implicants.

TABLE VI. NUMBER OF ESSENTIAL PRIME IMPLICANTS FOR DIFFERENT FILL FACTOR

fill-factor	Minterms	Primes	Essential primes
0.1	104	146	72 (49.3%)
0.2	205	673	106 (15.8%)
0.3	308	7,952	110 (1.4%)
0.4	411	3,485,799	97 (0.003%)

TABLE VII. NUMBER OF ESSENTIAL PRIME IMPLICANTS FOR DIFFERENT NUMBER OF INPUT VARIABLES

Inputs	Minterms	Primes	Essential primes
10	104	146	72 (49.3%)
15	3,278	6919	2048 (29.6%)
20	104,857	296,035	60,682 (20.5%)

C. Number of Essential Prime Implicants

Table VI and Table VII show the number of prime implicants before and after step 2 of the Quine-McCluskey method. As can be seen from the tables, the number of essential prime implicants achieved after step 2 decreases significantly compared to the number of prime implicants. The number of essential prime implicants is always smaller than the number of minterms and much smaller than the number of total prime implicants. Particularly, the number of essential prime implicants is less than 50% of the total number of prime implicants. For the high fill factors, 0.3 and 0.4, the percentages are even lower than 2%, 1.4% at the fill factor 0.3 and 0.003% at the fill factor 0.4. Table VI reveals that the higher fill factor the lower percentage of total prime implicants that are essential.

VI. CONCLUSION

In this paper, we address the performance bottleneck of the Quine-McCluskey method to the memory access since the application is memory-intensive. We propose a bitarray-based data representation for implicants of Boolean functions and an algorithm to find all the prime implicants. The proposals exploit the data locality of cache memories. The experimental results show that our proposal causes fewer cache load misses than the original Quine-McCluskey method does, leading to higher performance. In this work, we also minimize the number of essential prime implicants. The result shows that the percentage of prime implicants that becomes essential is quite small, less than 50%. This helps to reduce the hardware cost in the implementation of the Boolean function. In future work, we will take into account the minimazation of multi-output Boolean functions as well as a framework for design and verification of Boolean functions.

References

[1] M. Karnaugh, "The map method for synthesis of combinational logic circuits," Transactions of the American Institute of Electrical Engineers, vol. 72 part I, pp. 593–598, 1953.

[2] E. J. McCluskey, "Minimization of boolean functions," Bell System Technical Journal, vol. 35, Issue 6, pp. 1417–1444, 1956.

[3] W. V. Quine, "The problem of simplifying truth functions," The American Mathematical Monthly, vol. 59, no. 8, pp. 521–531, Mathematical Association of America, 1952.

[4] W. V. Quine, "A way to simplify truth functions," The American Mathematical Monthly, vol. 62, no. 9, pp. 627–631, Mathematical Association of America, 1955.

[5] M. Petrík, "Quine–McCluskey method for many-valued logical functions," Soft Computing, vol. 12, Issue 4, pp. 393–402, Springer-Verlag, 2007.

[6] S. P. Tomaszewski, I. U. Celik, G. E. Antoniou, "WWW-based boolean function minimization," International Journal of Applied Mathematics and Computer Science,vol. 13, no. 4, pp. 577–583, 2003.

[7] I. Wegener, "The complexity of boolean functions," John Wiley & Sons, Inc. New York, NY, USA,1987.

[8] P. W. Chandana Prasad, Azam Beg, Ashutosh Kumar Singh, "Effect of Quine-McCluskey simplification on boolean space complexity," In: Innovative Technologies in Intelligent Systems and Industrial Applications, CITISIA 2009, IEEE, 2009.

[9] A. Dusa, A. Thiem, "Enhancing the minimization of boolean and multivalue output functions with eQMC," The Journal of Mathematical Sociology, 39:2, pp. 92–108, 2015.

[10] B. Gurunath, N.N. Biswas, "An algorithm for multiple output minimization," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, Issue 9, pp. 1007–1013, IEEE, 1989.

[11] T. K. Jain, D. S. Kushwaha, A. K. Misra, "Optimization of the Quine-McCluskey method for the minimization of the boolean expressions," Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08), Gosier, 2008, pp. 165–168.

[12] A. Majumder, B. Chowdhury, A. J. Mondai, K. Jain, "Investigation on Quine McCluskey method: A decimal manipulation based novel approach for the minimization of boolean function," International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV), 2015, DOI: 10.1109/EDCAV.2015.7060531.

[13] V. Siládi, T. Filo, "Quine-McCluskey algorithm on GPGPU," In: AWER Procedia Information Technology and Computer Science, vol. 4 3rd World Conference on Innovation and Computer Science (INSODE-2013), pp. 814–820, 2013.

[14] V. Siládi, M. Povinsky, L. Trajtel, "Adapted parallel Quine-McCluskey algorithm using GPGPU," 14th International Scientific Conference on Informatics, pp. 327-331, 2017.