Regular Article

Performance Analysis of Quine-McCluskey Method on CPU

Hoang-Gia Vu[1], Thanh Bang Le[1], Dai-Do Tran [2], Do Xuan Tien [3]

[1] Faculty of Radio-Electronic Engineering, Le Quy Don Technical University, Ha Noi, Vietnam
[2] Faculty of Basics Training, Telecommunications University, Khanh Hoa, Vietnam
[3] Electric Power University, Ha Noi, Vietnam

Correspondence: Thanh Bang Le, banglt@lqdtu.edu.vn
Communication: received 03 September 2024, revised 22 October 2024, accepted 27 October 2024
Online publication: 15 December 2024, Digital Object Identifier: 10.21553/rev-jec.385

Abstract– **The Quine-McCluskey method is a widely used procedure for minimize Boolean functions. Although the method can be programmed on computers, it takes a long time to return the set of essential prime implicants, thus slowing the analysis and design of digital logic circuits. In this paper, we first propose two methods of data representation for prime implicants in memory, followed by our performance analysis for each representation. We then propose a multithreading scheme to find all prime implicants of a Boolean function. This scheme aims to accelerate step 1 of the method on multicore platforms. 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 the mask-based representation achieves the highest performance when the input number is small. When the input number is 20 or more, the best data representation is bitarray-based. The bitarray-based representation achieves a 5x higher performance than the ASCII-based representation when the input number is 24 and the fill factor is 0.002. The number of essential prime implicants can be reduced by up to 45% of the total prime implicants generated in step 1 of the method for a 16-input Boolean function at a fill factor of 0.05.**

Keywords– **Quine-McCluskey, prime implicant, multithreading, Boolean function.**

1 Introduction

A Boolean function is a mathematical expression consisting of binary variables and logical operators. These functions are the basic components for designing and developing digital circuits and systems. Since the main goal of design is to keep the number of logic gates as low as possible, this reduces the cost of manufacturing these systems. The complexity of logic design is directly related to the complexity of Boolean functions. Therefore, designers aim to create the simplest Boolean function. Simplifying the algebraic formulas of any given Boolean function is known as minimization. The two most commonly used methods in practice for minimizing a Boolean function are the Karnaugh map (K-map) and the Quine-McCluskey (QM). The K-map was first proposed by Veitch [1] and later modified by Karnaugh [2], which gives a simple, fundamental procedure for reducing Boolean functions. However, if there are more than five or six variables, the K-map approach becomes inconvenient. The QM approach is more appropriate when more than six input variables are used in the Boolean function. This method generates prime implicant lists using the tabulation technique, which was first introduced by Quine [3, 4] and then improved by McCluskey [4]. Similar to the K-map, the QM technique looks for entries that differ by only a single bit in order to collect product terms. The only difference between the two methods is that the QM performs the searching process instead of the mapping. Basically, the QM Method consists of the following two steps:

Step 1: Identify all prime implicants of the Boolean function.

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

The primary benefit of the QM approach is its ability to be used algorithmically in software. However, its drawback is that computational complexity still remains high, which can be theoretically calculated as a function $O(N^{\log_2 3} \log_2 N)$, N - the input length [5]. This procedure indicates that the execution time of the QM method increases exponentially with the number of input variables. Consequently, it slows down digital logic circuit analysis, design, and verification, especially when designing reconfigurable hardware architectures. There have been many studies on the computational complexity of the QM method. In an early work, Mileto et al. presented mathematical formulas for the average number of comparison operations among prime k-cubes occurring in Quine's and Mc-Cluskey's method [6]. Prasad et al. analyzed the behavior of QM simplification for various numbers of product terms and also introduced a mathematical model to forecast the Boolean space complexity as shown in the following equation [7]:

$$N = a \cdot t^b \cdot e^{-tc} + 1, \tag{1}$$

where N is the number of literals; t is the number of non-repeating product terms in the Boolean function; a, b and c are three constants depending on the number of input variables.

Several studies have focused on the rapid and automated simplification of Boolean functions to enhance

the process of minimizing these functions. Dusa et al. introduced eQMC, a method designed to reduce the computational complexity of the QM algorithm [8]. The approach relies on two seemingly insignificant observations. First, every positive-complete product is a (proper) subset of at least one prime implicant. Secondly, no subset of any prime implicant is a negative-complete product. Based on these two related facts, eQMC performs an exhaustive procedure that relies on index vectors instead of complex matrices. The simulation results show reduced memory consumption and completion time. As the set of configurations explicitly excluded from the minimization process increases, eQMC slows down until its advantage over QMC disappears. Gurunath et al. introduced an algorithm for multiple output minimization [9]. This study presented a novel category of selective prime cubes, termed valid selective prime cubes. The new class has demonstrated significant utility by directing the algorithm toward the minimal set of selective prime cubes when faced with single or connected cyclic sequences of selective prime cubes. In numerous instances, this approach helps to avoid the need for branching, which is a computationally intensive process. In [10], Jain et al. optimized the QM method by introducing the concept of Reduced Mask, which reduces the run time complexity of the algorithm by proposing an efficient algorithm for the determination of prime implicants. The Reduced Mask has bits set corresponding to the literal reduction, and as a result, the execution time decreased significantly.

The other approach to improve the performance of Boolean functions is minimization based on decimal manipulations [11]. The study demonstrates that as the number of variables in a Boolean function increases, the proposed method exhibits a significantly lower rate of increase in the number of comparisons compared to the original QM method. For instance, for a Boolean function with ten variables, the proposed method requires 32.80 times fewer comparisons than the QM method. To enhance the efficiency of the QM technique, Siládi et al. proposed a scheme to adapt the first step of the method for parallel execution on a GPU computing platform [12, 13]. In these works, the proposed algorithm processes implicants across multiple rounds. During each round of the first step, implicants are initially partitioned based on the positions of dashes within the terms. Each partition is then examined for terms that can be merged. In the merging process, the list of terms is first transformed into a bitmap representation, where each term is represented as a bit set. Initially, the bitmap is loaded into shared memory, and the usage flag for each minterm representation is initialized to zero. Following thread synchronization, the minterms are combined with other minterms utilizing the instructions specified in rows. These sections of the program code represent the inherently parallel implementation discussed earlier, utilizing CUDA. The kernel executes on the GPU (device), while the remaining components of the algorithm operate on the CPU (host). This process necessitates the transfer of data between the host and the device. While the proposed parallel algorithm demonstrates greater efficiency in utilizing device memory compared to previously published parallel algorithms, it remains suboptimal for implementation on GPUs. Adapting the inherently parallel aspects of the QM algorithm for GPU execution does not substantially enhance the speed of the reduction process. Furthermore, this work does not consider the Boolean functions don't care (x) output values.

This work was partially presented in [14], where we introduced a bitarray-based data representation for storing implicants in memory to minimize cache misses in the program. In this paper, we make two new contributions. Firstly, we present a new data representation, named *mask-based*, for implicants in the Quine-McCluskey method. Secondly, we introduce a scheme for step 1 of the method allowing the comparisons among groups executed on multiple threads. Totally the contributions of this study are as follows:

1) We present two data representations for prime implicants of a Boolean function. We then analyze the performance of the Quine-McCluskey method based on each of the two data representations. For Boolean functions with large numbers of input variables, we propose to use the bitarray-based data representation for implicants to reduce the cache misses when the method running on the CPU.

2) We propose a scheme for step 1 of the method executed on multiple threads. We evaluate the performance of the multithreading scheme and identify the bottleneck of the scheme.

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 remainder of the manuscript is structured as follows: Section 2 briefly introduces the QM method. Section 3 presents the data representations for prime implicants. Section 4 details the proposed algorithms for the method. Section 5 discusses the evaluation results, and Section 6 concludes with a summary of the findings.

2 The Quine-Mccluskey Method

The Quine-McCluskey method is a simple and systematic approach to minimizing Boolean functions. The basic idea behind this tabulation procedure is that by repeatedly applying the associativity theorem $XY + XY' = X$ (where X is the set of literals) on all adjacent pairs of terms. We obtain the set of all prime implicants from which the smallest sum may be chosen. The Quine-McCluskey method comprises two major steps:

Step 1: Identify the prime implicants, which are the products where the function evaluates to logic 1 or has an indeterminate value, along with the variable components that cannot be further simplified. This step is divided into four sub-steps:

1.1) Focus exclusively on combinations of variables for which the output function $y = 1$ or has an indeterminate value, and represent these combinations using binary code.

1.2) Construct a Quine-McCluskey table by organizing the combination blocks into adjacent layers, ordered by the increasing number of 1 bit.

1.3) Compare each combination in the i-th class with the combinations in the $(i+1)$-th class. If the two variable combinations differ by only a single bit, replace both combinations with a new combination in which the differing bit is substituted with a dash (-), and mark the two original combinations with an asterisk (*).

1.4) After completing all comparisons, repeat step 1.3 until no further combinations can be generated. The set of combinations in the final column, along with the unmarked combinations from the preceding columns, constitutes the prime implicants of the function.

Step 2: Identify the essential prime implicants, which are the prime implicants of the logic function that encompass the minimum number of functions while preserving the original function's behavior. Only the combinations of variables for which output $y = 1$ are of significance. Consequently, the table is structured with columns representing these variable combinations and rows corresponding to the minimum products of the function. Cells are marked with an 'x' if the minimum product of the function encompasses the corresponding column associated with that cell. Rule for selecting essential prime implicants: Select the columns marked with an 'x' and eliminate the corresponding rows and columns associated with that row. If no such columns exist, then prioritize rows containing multiple 'x's, ensuring that at least one minimum product is selected. Continue this process until all elements are covered using the minimum number of positive products.

In general, the Quine-McCluskey method offers a more effective approach for function simplification compared to the K-map. However, it remains a challenging problem and becomes impractical for large input sizes due to its exponential complexity.

3 Data Representation

In this section, we present three ways of data representation for the Quine-McCluskey method. The first way uses ASCII characters to represent symbols, called *ASCII-based* data representation. The second representations is based on the bit array format, called *bitarray-based* data representation. The third way is based on a mask to indicate whether symbols are don't care, called *mask-based* data representation. It is noted that the original Quine-McCluskey method uses the first way, ASCII-based data representation, to express implicants.

3.1 ASCII-based Data Representation

Symbols in a prime implicant can be written by ASCII characters. Particularly, they are represented as follows:

$$\begin{aligned}\text{Symbol '1'} &\rightarrow \text{ ASCII character '1'},\\ \text{Symbol '0'} &\rightarrow \text{ ASCII character '0'},\\ \text{Symbol '-'} &\rightarrow \text{ ASCII character '-'}.\end{aligned}$$

As a result, implicant (10-00-10) will be represented as a 16-bit array as follows:

$$\text{(10-00-10)} \rightarrow \text{ ASCII string '10-00-10'}.$$

This representation requires 8 bytes, and it is actually allocated 8 bytes in the main memory.

3.2 Bitarray-based Data Representation

To reduce the memory allocation 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:

$$\begin{aligned}\text{Symbol '1'} &\rightarrow \text{ bit array '01'},\\ \text{Symbol '0'} &\rightarrow \text{ bit array '00'},\\ \text{Symbol '-'} &\rightarrow \text{ bit array '10'}.\end{aligned}$$

As a result, implicant (10-00-10) will be represented as a 16-bit array as following:

$$\text{(10-00-10)} \rightarrow \text{bit array '0100100000100100'}.$$

The implicant (10-00-10) consumes 8 bytes if it is represented in the form of ASCII characters. The implicant requires 2 bytes (16 bits) if represented in a bit array. Therefore, the memory requirement of the bit array-based representation is four times more efficient than that of the ASCII-based representation. In fact, bit array variables are often allocated a given number of bytes in the memory. If each bit array variable is allocated 64 bits, then the implicant (10-00-10) will consume 8 bytes in the memory, the same as the ASCII-based representation. However, when the number of input variables increases, the bitarray-based representation may be more memory-efficient than the ASCII-based one. For example, a 16-symbol ASCII-based implicant consumes 16 bytes in memory, more than only 8 bytes for that 16-symbol implicant in the bitarray-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.

3.3 Mask-based Data Representation

Beside the bitarray-based data representation, we propose a mask-based data representation that uses two-bit arrays for an implicant. The first bit array shows the value of the implicant, called the *value*. The second indicates the position of dash '-' symbols in the implicant, which is called the *mask*. The combination of the *value* and the *mask* fully represents the implicant.

Algorithm 1 Comparison of two bitarray-based implicants

```
1: Input: a, b: bitarray-based implicants
2: Output: position of the symbol making 2 implicants different
3: temp ← a OR b
4: if temp contains one bit '1' then
5:     return position of the bit '1' in temp
6: else
7:     return -1
8: end if
```

For the value, symbols '1', '0', and '-' are represented as follows:

$$\begin{aligned}\text{Symbol '1'} &\rightarrow \text{ bit array '1'},\\ \text{Symbol '0'} &\rightarrow \text{ bit array '0'},\\ \text{Symbol '-'} &\rightarrow \text{ bit array '0'}.\end{aligned}$$

For the *mask*, symbols '1', '0', and '-' are represented as simple as follows:

$$\begin{aligned}\text{Symbol '1'} &\rightarrow \text{ bit array '1'},\\ \text{Symbol '0'} &\rightarrow \text{ bit array '1'},\\ \text{Symbol '-'} &\rightarrow \text{ bit array '0'}.\end{aligned}$$

As a result, implicant (10-00-10) will be represented by an 8-bit value and an 8-bit mask as follows:

$$\begin{aligned}(10-00-10) \rightarrow &\text{ value '10000010'},\\ &\text{mask '11011011'}.\end{aligned}$$

Totally, the data representation in this way also requires 16 bits (2 bytes). In fact, if each bit array value is allocated 64 bits, then this mask-based representation will consume 128 bits (18 bytes) instead of 8 bytes as the ASCII-based representation. However, it takes less time to count the number of bits '1' in the 8-bit value than doing that in a 16-bit bit array. Therefore, the mask-based data representation is expected to be better than the bit array-based representation in terms of comparison time between two implicants.

For comparison between two mask-based implicants to find out if they can be combined into a new implicant, we first check whether their masks are equal or not. If they are equal, then the operator XOR is used between their values as in Algorithm 2. If the two implicants differ in only one symbol, then the XOR operation of two corresponding values will return a bit array including only one bit '1'. If the number of bits '1' is equal to one, the position of the bit '1' is returned. Otherwise, '-1' will be returned.

Algorithm 2 Comparison of two mask-based implicants

```
1: Input: a, b: mask-based implicants
2: Output: position of the symbol making 2 implicants different
3: if a.mask = b.mask then
4:     temp ← a.value OR b.value
5:     if temp contains one bit '1' then
6:         return position of the bit '1' in temp
7:     end if
8: end ifreturn -1
```

4 Algorithms for The Quine-Mccluskey Method

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

4.1 Algorithm for Step 1: Finding all Prime Implicants

In step 1 of the Quine-McCluskey method, the major operations are memory accesses to read all the implicants and the comparison among implicants. We believe that the performance bottleneck in this step is the huge number of memory references. In this part, we propose an algorithm for step 1 of the method. The Pseudo code is described in Algorithm 3.

Algorithm 3 Finding all prime implicants

```
1: Input: m = # input variables
2: list-1 = [minterms]
3: list-x = [d-terms]
4: Output: prime-list
5: implicant-list = list-1 ∪ list-x
6: prime-list = []
7: new-implicant-list = []
8: combined = True
9: while (combined) do
10:     combined = False
11:     groups = make_groups(implicant-list)
12:     for i = 0 to (m − 1) do
13:         for x1 in groups[i] do
14:             for x2 in groups[i + 1] do
15:                 pos = compare(x1, x2)
16:                 if pos ≠ −1 then
17:                     combined = True
18:                     new-term = x1
19:                     Update new-term with a new dash
20:                     x1.used = True
21:                     x2.used = True
22:                     new-implicant-list.append(new-term)
23:                 end if
24:             end for
25:         end for
26:     end for
27:     for impl in implicant-list do
28:         if impl.used = False then
29:             implicant-list.remove(impl)
30:         end if
31:     end for
32:     prime-list.append(implicant-list)
33:     implicant-list = new-implicant-list
34: end while
```

- 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 implicant 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*.

In the *while* loop, *combined* is first assigned to *False*. Then all the implicants are classified as in Line 11. Implicants belonging to the same group have the same number of symbols '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 13 to Line 19. If there is any combination, the variable *combined* is marked as True in Line 17. 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 in Line 20,21. Then the new generated implicant, called *new-term*, is added to *new-implicant-list*.

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, *new-implicant-list* is assigned to implicant-list before the next round of the *while* loop. The *while* loop will end up if *combined* = *False*. That means the loop will terminate if no implicants can be combined to a new one.

In Algorithm 3, group[i], also called group i, is compared with group $i-1$ and group $i+1$. The comparison between group i and group $i+1$ consists of all comparisons among each implicant of group i and each implicant of group $i+1$. There are total $m+1$ groups in each round of the *while* loop. Therefore, there are m group comparisons, including the comparisons of group 0 and group 1, group 1 and group 2, group 2 and group 3, ..., and group $m-1$ and group m as in Figure 1. In each group comparison, many comparisons are made up from each pair of implicant groups. Each comparison requires two memory accesses to the two implicants. In this work, we propose to map each group comparison onto a thread. As a result, m threads are created and executed as in Figure 1. These threads are then mapped onto computing cores of the CPU. In a multi-core CPU, multithreading often achieves a higher performance than single thread because multiple cores can execute and access memory simultaneously.

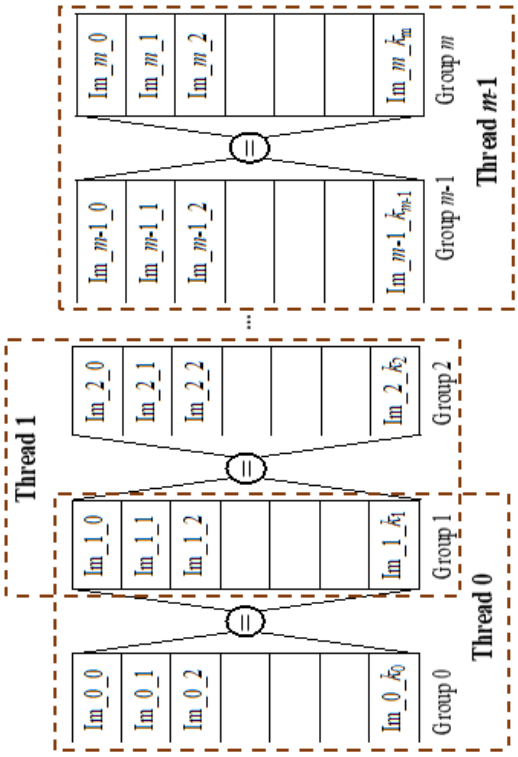

Figure 1. Mapping group comparisons onto threads.

Algorithm 4 Selection of Essential Prime Implicants

```
1:  Input: prime-list
2:  Output: final-prime-list
3:  final-prime-list = []
4:  victim = None
5:  max-len = 1
6:  while max-len ≠ 0 do
7:      max-len = 0
8:      for prime in prime-list do
9:          prime.val-1.difference_update(victim.val-1)
10:         if len(prime.val-1) > max-len then
11:             max-len = len(prime.val-1)
12:             temp = prime
13:         end if
14:     end for
15:     prime-list.remove(temp)
16:     victim = temp
17:     final-prime-list.append(temp)
18: end while
```

4.2 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. We aim to choose the smallest number of prime implicants that cover all the minterms of the Boolean function. For that perspective, we propose a function in Pseudo code to select essential prime implicants that cover the most minterms as in Algorithm 4. 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 7. *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 15. Then, the minterm set of each prime implicant is also updated as in Line 9 before finding *max-len*.

5 Evaluation

Table 5 shows the experimental setup we used to evaluate the proposed data representations and algorithms. Our proposals do not have any limitation on the number of input variables. However, the execution time increases exponentially when scaling the variable number. In this evaluation, the number of input variables in the Boolean function is scaled from 10 to 24. For the number more than 24, the execution time can last hours or days. For the number less than 10, the execution time can last several milliseconds that we do not need to accelerate. 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. *fill-factor-1* must be not more than 1.
- Let *fill-factor-0* be the ratio between the number of input vectors that evaluate to '0' and the total number of input vectors 2^k. *fill-factor-0* must be not more than 1.
- 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. *fill-factor-x* must be less than 1.

In this section, we evaluate the cache performance, execution time, and the number of essential prime implicants in our proposals. We then compare our two data representations with the ASCII-based data representation of the original Quine-McCluskey method. For the number of input variables 10, we scale the *fill-factor-1* and *fill-factor-x* from 0.1 to 0.4. It is noted that the sum of *fill-factor-1*, *fill-factor-0*, and *fill-factor-x* must be equal to 1. Therefore, we cannot scale both *fill-factor-1* and *fill-factor-x* to more than or equal to 0.5.

Table I
Evaluation Setup

CPU	Intel Core i9 – 12900K
Number of cores	16
L1 Dcache	640 KB (16 instances)
L1 Icache	768 KB (16 instances)
L2 cache	14 MB (10 instances)
L3 cache	30 MB (1 instance)
Main memory	32 GB
Block cache size	64 Bytes
Operating system	Ubuntu 22.4
Cache profiling tool	Perf 5.4.143

5.1 Cache Performance

Table II shows the load misses of L1 data cache for the Quine-McCluskey method evaluated on the three data representations 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 the mask-based representation is the largest number for each fill factor. The highest number of cache load misses comes from the most memory-consuming data representation. The smallest number of L1 cache misses belongs to the bitarray representation running on multiple threads. The same situation is in Table III and Table IV for the L2 cache misses and the L3 cache misses. As a result, the execution time for the mask-based and bitarray-based representation is expected to be shorter than that for the ASCII-based representation.

Table II
L1 Cache Miss for Different Fill Factors

Fill-factor	L1-misses (L1-loads) (ASCII)	L1-misses (L1-loads) (Mask)	L1-misses (L1-loads) (Bitarray)	L1-misses (L1-loads) (Bitarray) (Multi-thread)
0.1	1,658K (188,257K)	1,913K (117,101K)	1,638K (115,767K)	1,450K (79,046K)
0.2	2,322K (789,408K)	2,452K (377,481K)	2,076K (377,196K)	2,119K (268,923K)
0.3	7,984K (5,418M)	15,145K (3,866M)	10,173K (5,596M)	6,603K (1,593M)

Table III
L2 Cache Miss for Different Fill Factors

Fill-factor	L2-misses (L2-loads) (ASCII)	L2-misses (L2-loads) (Mask)	L2-misses (L2-loads) (Bitarray)	L2-misses (L2-loads) (Bitarray) (Multi-thread)
0.1	10,086 (250,962)	12,661 (309,699)	10,038 (226,533)	3,498 (81,379)
0.2	24,154 (303,621)	15,771 (383,583)	11,680 (353,855)	11,811 (167,849)
0.3	12,906 (356,449)	12,397 (237,830)	19,414 (324,124)	23,296 (386,820)

Table IV
L3 Cache Miss for Different Fill Factors

Fill-factor	L3-misses (L3-loads) (ASCII)	L3-misses (L3-loads) (Mask)	L3-misses (L3-loads) (Bitarray)	L3-misses (L3-loads) (Bitarray) (Multi-thread)
0.1	27,422 (316,617)	23,424 (360,180)	21,631 (343,158)	12,049 (151,871)
0.2	31,984 (321,276)	25,425 (234,079)	21,421 (221,399)	22,184 (198,494)
0.3	17,935 (176,758)	29,556 (301,314)	24,008 (203,321)	28,271 (224,900)

Table V, Table VI, and Table VII show the L1, L2, and L3 cache load misses, respectively, while scaling the input number. As can be seen from Table V, the ASCII-based method has the highest numbers of L1 cache misses compared to the other data representations. The lowest numbers belong to the bitarray-based representation. The same situation for L2 cache misses and L3 cache misses in Table VI and Table VII. When the number of inputs increases, the mask-based and the bitarray-based representations are much better than the ASCII-based. Particularly, when the input number is equal to 24, L3 cache misses for the bitarray-based is around 25 times smaller than that for the ASCII-based. Therefore, the execution time for the bitarray-based representation is expected much shorter than that for the ASCII-based.

When the input number increases, the numbers of L2 and L3 cache load misses for the bitarray-based representation on multiple threads are even smaller than the numbers of load misses on a single thread.

Therefore, the execution time of the multi-threading bitarray-based method is predicted shorter than that running on a single thread.

Table V
L1 Cache Miss for Different Input Numbers

Inputs (fill factor)	L1-misses (ASCII)	L1-misses (Mask)	L1-misses (Bitarray)	L1-misses (Bitarray) (Multi-thread)
12 (.1)	3,397,083	3,083,846	2,585,441	2,673,154
16 (.05)	65,956K	54,035K	26,989K	43,314K
20 (.01)	440,162K	387,084K	185,498K	261,792K
24 (.002)	4,554,942K	3,418,002K	1,521,518K	2,608,102K

Table VI
L2 Cache Miss for Different Input Numbers

Inputs (fill factor)	L2-misses (ASCII)	L2-misses (Mask)	L2-misses (Bitarray)	L2-misses (Bitarray) (Multi-thread)
12 (.1)	5,569	11,425	9,329	22,561
16 (.05)	148,174	69,535	122,544	101,358
20 (.01)	662,361	318,347	263,191	175,546
24 (.002)	80,873K	32,984K	3,027K	2,470K

Table VII
L3 Cache Miss for Different Input Numbers

Inputs (fill factor)	L3-misses (ASCII)	L3-misses (Mask)	L3-misses (Bitarray)	L3-misses (Bitarray) (Multi-thread)
12 (.1)	27,888	22,443	21,316	52,716
16 (.05)	315,143	129,136	89,935	78,893
20 (.01)	740,297	433,095	316,474	187,970
24 (.002)	81,396K	33,002K	3,259K	2,643K

The cache miss rates for L1, L2, and L3 cache when keeping the input number at 10 and scaling the fill factor from 0.1 to 0.3 are presented in Figure 2, Figure 3, and Figure 4, respectively. As can be seen from Figure 2, the miss rates are quite small compared to the miss rates for L2 and L3 cache. Although the CPU cores takes a huge number of load references to L1 cache, many of the accesses are to the same memory addresses or memory blocks. This reduces the miss rate significantly. All the three figures show that the miss rates for the ASCII-based are not higher than that for the other two data representations. However, the numbers of misses for the ASCII-based are quite larger than that for the mask-based and bitarray-based representation. As a result, the average access time for the ASCII-based is longer than that for the other representations. It is the reason why the execution time for the original Quine-McCluskey method is so long compared to the mask-based and bitarray-based method.

5.2 Execution Time

Table VIII shows the execution time in seconds for step 1, step 2, and the total process of the Quine-McCluskey method for different fill factors. In this experiment, step 1 was executed in a single thread for each of the three data representations. After that, it was executed in multiple threads for the bitarray-based representation. The results reveal that the mask-based and bitarray-based data representations achieved much higher performance than the ASCII-based did. The

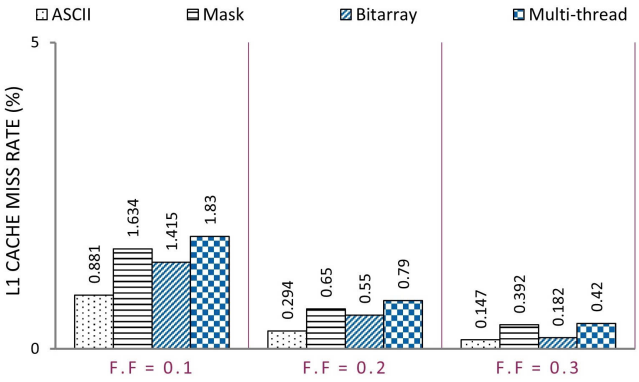

Figure 2. L1 cache miss rate for different fill factors.

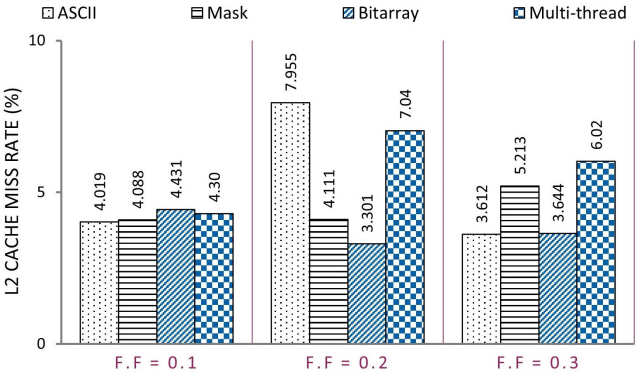

Figure 3. L2 cache miss rate for different fill factors.

Table VIII
Execution Time (s) For Different Fill Factor

Fill-factor	Total (ASCII)	Total (Mask)	Total (Bitarray)	Total (Multi-thread)
	Total	Total	Total	Total
	Step 1	Step 1	Step 1	Step 1
	Step 2	Step 2	Step 2	Step 2
0.1	0.019	0.006	0.007	0.009
	0.016	0.005	0.006	0.008
	0.003	0.001	0.001	0.001
0.2	0.102	0.033	0.046	0.044
	0.096	0.027	0.039	0.038
	0.006	0.006	0.007	0.006
0.3	0.730	0.484	0.809	0.319
	0.706	0.431	0.756	0.258
	0.023	0.053	0.053	0.061

Table IX
Execution Time (s) For Different Number of Input Variables

Inputs (fill-factor)	Total (ASCII)	Total (Mask)	Total (Bitarray)	Total (Bitarray Multi-thread)
	Total	Total	Total	Total
	Step 1	Step 1	Step 1	Step 1
	Step 2	Step 2	Step 2	Step 2
12 (0.1)	0.157	0.055	0.061	0.062
	0.137	0.040	0.047	0.046
	0.020	0.015	0.014	0.015
16 (0.05)	7.550	2.434	2.471	2.527
	6.455	1.681	1.712	1.567
	1.095	0.753	0.759	0.960
20 (0.01)	54.351	17.977	15.470	16.399
	44.079	13.162	10.588	9.029
	10.272	4.815	4.882	7.370
24 (0.002)	616.62	169.15	149.08	144.86
	493.18	118.80	97.14	84.21
	123.44	50.35	51.94	60.65

same situation is presented in Table IX while scaling the number of input variables. This higher performance in the mask-based and bitarray-based methods comes from the bit-level parallel execution and the smaller

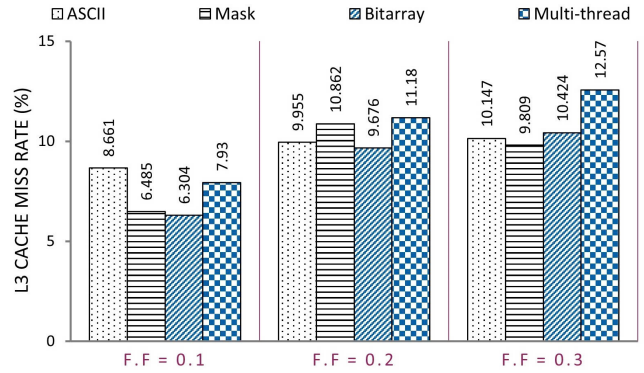

Figure 4. L3 cache miss rate for different fill factors.

numbers of cache load misses revealed in Section 5.1.

As can be seen in Table VIII and Table IX, the bitarray-based method running on multiple threads achieved not much higher performance compared to running on a single thread. 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.

Table VIII also shows that the execution time of step 2 for the ASCII-based data representation are almost the same as that for the two other data representations. However, when scaling the input number, the time for the ASCII-based data representation is much longer than that for the two other data representations as in Table IX. The increase in the input number leads to the growth in the database for the prime implicant list as the input data for step 2. The larger database may cause more cache misses, thus extending the execution time.

5.3 Number of Essential Prime Implicants

Table X and Table XI 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. For the high fill factors, 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.

Table X
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%)

6 Conclusion

In this paper, we address the performance bottleneck of the Quine-McCluskey method in the memory access since the application is memory-intensive. We propose three data representations for prime implicants of Boolean functions and recommend the bitarray-based representation for the Boolean functions with large numbers of inputs. The multithreading execution for the method can achieve a higher performance for each Boolean function, but the improvement is not significant. In this work, we also minimize the number of essential prime implicants. The result show that the percentage of prime implicants that becomes essential is quite small, 0.003% for a 10-input Boolean function with a fill factor of 0.4. This helps to reduce the hardware utilization in the implementation of the Boolean function. In future work, we will take into account the minimization of multi-output Boolean functions as well as a framework for design and verification of Boolean functions. We will also consider logic functions in the form of the product of sums to accelerate the logic minimization.

Table XI
Number of Essential Prime Implicants for Different Number of Input Variables

Inputs (fill factor)	Minterms	Primes	Essential primes
12 (0.1)	409	697	276 (39.6%)
16 (0.05)	3277	4393	2451 (55.8%)
20 (0.01)	10485	10122	9610 (94.9%)
24 (0.002)	33554	32910	32799 (99.7%)

References

[1] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers*, vol. 72, no. 1, pp. 593–598, 1953.

[2] E. J. McCluskey, "Minimization of boolean functions," *Bell System Technical Journal*, vol. 35, no. 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, 1952.

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

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

[6] I. Wegener, *The complexity of boolean functions*. New York, NY, USA: John Wiley & Sons, 1987.

[7] P. W. C. Prasad, A. Beg, and A. K. Singh, "Effect of Quine-McCluskey simplification on boolean space complexity," in *Proceedings of the Innovative Technologies in Intelligent Systems and Industrial Applications, CITISIA 2009*. IEEE, 2009, pp. 99–102.

[8] A. Dua and A. Thien, "Enhancing the minimization of boolean and multivalue output functions with QMC," *The Journal of Mathematical Sociology*, vol. 39, no. 2, pp. 92–108, 2015.

[9] B. Guruswamy and N. Srinivas, "An algorithm for multiple output minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 9, pp. 1007–1013, 1989.

[10] T. K. Jain, D. S. Kushwaha, and A. K. Misra, "Optimization of the Quine-McCluskey method for the minimization of the boolean expressions," in *Proceedings of*

the ACOS'08: International Conference on Autonomous and Autonomous Systems, 2008, pp. 165–168.

[11] A. Majumder, B. Chowdhury, J. J. Mondal, and R. Jain, "Investigation on Quine-McCluskey method: A decimal manipulation based novel approach for the minimization of boolean functions," in *Proceedings of the International Conference on Electronics Computer Networks & Automated Systems*, 2009, pp. 10–12.

[12] V. Siladi and T. Fifo, "Quine-McCluskey algorithm on GPGPU," in *Proceedings of the NSODE-CSC International Conference on Innovation and Computer Science*, 2013, pp. 34–38.

[13] V. Siladi, M. Povinsky, and L. Trajtel, "Adapted parallel Quine-McCluskey algorithm using GPGPU," in *Proceedings of the 14th International Scientific Conference on Informatics*, 2017, pp. 327–331.

[14] H. G. Vu, N. D. .Bui, A. T. Nguyen, and ThanhBangLe, "Performance evaluation of Quine-McCluskey Method on Multi-core CPU," in *Proceedings of the 2014 5th International Conference on Electronics and Automation*, 2014, pp. 60–64.

Hoang-Gia Vu received the B.E. and M.E. degrees in Electrical Engineering from Le Quy Don Technical University, Vietnam in 2007 and 2010, respectively. He received his PhD degree in electrical engineering from Nara Institute of Science and Technology, Japan, in 2018. He now works as a lecturer and researcher in Le Quy Don Technical University. His research interests include re-configurable computing, high-performance computing, and processor architecture.

Thanh Bang Le currently serves as a lecturer and researcher at Le Quy Don Technical University. He earned his Ph.D. in Electrical Engineering from the Brno University of Defence in the Czech Republic in 2019, following his M.E. and B.E. degrees in Electrical Engineering from Le Quy Don Technical University, Vietnam, in 2013 and 2006, respectively. His research focuses on embedded system design, IoT systems, and reconfigurable computing.

Dai-Do Tran is currently a lecturer at the Faculty of Basics Training at Telecommunications University Vietnam. He obtained of his Bachelor degree in electronic engineering from Hanoi University of Science and Technology, Vietnam in 2015. He received his master of Engineering in Electronic engineering from Le Quy Don Technical University in 2023. His research interests are in the area of microcontroller programming, electronics and semiconductor.

Xuan Tien Do received the B.E. and PhD. degrees in Electrical Engineering from St. Petersburg State Electrotechnical University, Russia in 1974 and 1987, respectively. He now works as a lecturer and researcher in Electric Power University. His research interests include parallel computing, high-performance computing, and smart computing.