Research Article

Multiple Memory Structure Bit Reversal Algorithm Based on Recursive Patterns of Bit Reversal Permutation

K. K. L. B. Adikaram,1,2,3 M. A. Hussein,1 M. Effenberger,2 and T. Becker1

1 Group of Bio-Process Analysis Technology, Technische Universität München, Weihenstephaner Steig 20, 85354 Freising, Germany
2 Institut für Landtechnik und Tierhaltung, Vöttinger Straße 36, 85354 Freising, Germany
3 Computer Unit, Faculty of Agriculture, University of Ruhuna, Mapalana, 81100 Kamburupitiya, Sri Lanka

Correspondence should be addressed to K. K. L. B. Adikaram; lasantha@daad-alumni.de

Received 6 April 2014; Revised 4 June 2014; Accepted 5 June 2014; Published 17 July 2014

Academic Editor: Ker-Wei Yu

Copyright © 2014 K. K. L. B. Adikaram et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

With the increasing demand for online/inline data processing efficient Fourier analysis becomes more and more relevant. Due to the fact that the bit reversal process requires considerable processing time of the Fast Fourier Transform (FFT) algorithm, it is vital to optimize the bit reversal algorithm (BRA). This paper is to introduce an efficient BRA with multiple memory structures. In 2009, Elster showed the relation between the first and the second halves of the bit reversal permutation (BRP) and stated that it may cause serious impact on cache performance of the computer, if implemented. We found exceptions, especially when the said index mapping was implemented with multiple one-dimensional memory structures instead of multidimensional or one-dimensional memory structure. Also we found a new index mapping, even after the recursive splitting of BRP into equal sized slots. The four-array and the four-vector versions of BRA with new index mapping reported 34% and 16% improvement in performance in relation to similar versions of Linear BRA of Elster which uses single one-dimensional memory structure.

1. Introduction

The efficiency of a bit reversal algorithm (BRA) plays a critical role in the Fast Fourier Transform (FFT) process because it contributes 10% to 50% of total FFT process time [1]. Therefore, it is vital to optimize the BRA to achieve an efficient FFT algorithm. In 2009, Elster showed the relation between the first and the second halves of the BRP [2], but did not implement it. Elster stated that implementation of this relation may cause serious impact on cache performance of modern computers. As Elster stated, use of a two-dimensional memory structure to implement this relation reduces the efficiency of the bit reversal permutation (BRP). In contrast, the efficiency of the BRA increased when a one-dimensional memory structure was used for index mapping. When two equal sided one-dimensional memory structures were used, the performance was even much better than with single one-dimensional memory structure. Also, it was found out that bit reversal permutation can be further split into equal size blocks recursively, up to maximum of \( \log_2(n) \) times, where \( n \) is the number of samples and \( n = 2^k \); \( k \in \mathbb{Z}^+ \). These two findings motivate us to introduce a BRA which is capable of using \( 2^k \) \( (k = 2, 3, \ldots, \log_2(n)) \) equal-sized \( (n/2^k) \) one-dimensional memory structures.

In 1965 Cooley and Tukey introduced the FFT algorithm, which is an efficient algorithm to compute the Discrete Fourier Transformation (DFT) and its inverse [3]. FFT is a fast algorithm that has replaced the process of DFT, which had been used frequently in the fields of signal and image processing [4–7]. The structure of the FFT algorithm published by Cooley and Tukey known as radix-2 algorithm [3] is the most popular one [5]. There are several other algorithm structures as radix-4, radix-8, radix-16, mixed-radix, and split-radix [8].

To apply FFT to a certain signal, there are basically two major requirements. The first requirement is \( n = b^N \), where \( n \) is the number of samples of the signal, \( N \in \mathbb{Z}^+ \), and \( b \) is the selected radix structure, for example, \( b = 2, 4, 8, \) and 16 for radix-2, radix-4, radix-8, and radix-16, respectively. The second requirement is that the input (or output) samples
must be arranged according to a certain order to obtain the correct output [3, 5, 8, 9]. The BRA is used to create the order of input or output permutation according to the required order. The BRA, used in most FFT algorithms, including the original Cooley-Tukey algorithm [3], is known as bit reversal method (BRM). The BRM is an operation, for exchanging two elements \( x(k) \) and \( x(\tilde{k}) \) of an array of length \( m \) as shown in (1) and (2), respectively, where \( a_i \) are either 0 or 1 and \( b \) is the relevant base 2, 4, 8, or 16 depending on the selected radix structure:

\[
k = \sum_{j=0}^{m-1} a_j b^j, \quad (1)
\]

\[
\tilde{k} = \sum_{j=0}^{m-1} a_j b^{m-1-j}. \quad (2)
\]

All the later algorithms for creating BRP were named as BRA (bit reversal algorithm), though they used other techniques like patterns of BRP instead of bit reversing techniques.

During the last decades, many publications addressed new BRAs [10] by improving the already existing original BRA (BRM) or using totally different approaches. In 1996, Karp compared the performance of 30 different algorithms [10] against uniprocessor systems (computer system with a single central processing unit) with different memory systems. Karp found that the performance of a BRA depended on the memory architecture of the machine and the way of accessing the memory. Karp stated two hardware facts that influence the BRA, namely, the memory architecture and the cache size of the machine. According to Karp, a machine with hierarchical memory is slower than a machine with vector memory (computers with vector processor), and algorithms do not perform well when array size is larger than the cache size. Also Karp pointed out four features of an algorithm that influence the BRA, namely, memory access technique, data reading sequence, size of the index of memory, and type of arithmetic operations. According to Karp, an algorithm that uses efficient memory access techniques is the fastest among algorithms with exactly the same number of arithmetic operations. Algorithms are faster if (i) they require only a single pass over the data, (ii) they use short indexes, and (iii) they operate with addition instead of multiplication.

Karp especially mentioned that the algorithm published by Elster [11] in 1989 was different from other algorithms, because it used a pattern of BRP rather than improving decimal to binary and binary to decimal conversion. According to the findings of Karp, Elster’s “Linear Bit Reversal Algorithm” (LBRA) performs much better in most of the cases. The publication of Elster (1989) [11] consists of two algorithms to achieve BRP. One algorithm used a pattern of BRP and the other one used bit shifting operations. Both algorithms are interesting because they eliminate the conventional bit reversing mechanism, which need more computing time. The algorithm by Rubio et al. (BRA-Ru) of 2002 [12] is another approach that uses an existing pattern of BRP. However, the pattern described in Rubio’s algorithm is different from the pattern described in Elster’s [11]. In 2009, Elster and Meyer published an improved version of “Linear Register-Level Bit Reversal” which was published in 1989 [11] as “Elster’s Bit Reversal” (EBR) algorithm. Elster mentioned it is possible to generate the second half of the BRP by incrementing the relevant element of the first half by one. Also, Elster mentioned there can be a serious impact on cache performance of the computer if the said pattern (Figure 1) is used.

Programming languages provide different data structures [13] which handle memory in different ways. In addition, the performance of the memory depends on machine architecture and operating system [14]. Therefore, the efficiency of the memory is the resultant output of the performances of hardware, operating system, programming language, and selected data structure.

Based on the physical arrangement of memory elements, there are two common ways of allocating memory for a series of variables: “slot of continuous memory elements” and “collection of noncontinuous memory elements,” commonly known as “stack” and “heap” [15]. In most programming languages, the term array is used to refer to a “slot of continuous memory elements.” Arrays are the simplest and most common type of data structure [16, 17] and, due to continuous physical arrangement of memory elements, provide faster access than “collection of noncontinuous memory elements” memory types. However, with the development of programming languages, different types of data structures were introduced with very similar names to the standard names like array, stack, and heap. The names of new data structures sometimes did not agree with the commonly accepted meaning. “Stack,” “Array,” and “ArrayList” provided by Microsoft Visual C++ (VC++) [18, 19] are good examples. According to the commonly accepted meaning they should be a “slot of continuous memory elements,” but they are in fact a “collection of noncontinuous memory elements.” Therefore, it is not a good practice to determine the performance of a certain data structure just by looking at its name. To overcome this ambiguous situation, we use “slot of continuous memory elements” to refer to “primitive array” (or array) type memory structures.

Due to the very flexible nature, vector is the most common one among the different types of data structures [14]. Vector was introduced with C++, which is one of the most common and powerful programming languages which has been used since 1984 [20]. However, as most of other data structures, the term “vector” is used to refer to memory in computers with processor architecture called “vector processor.” In this paper the term vector is used to refer to the vector data structure that is used in the C++ interface.

Index mapping is a technique that can be used to improve the efficiency of an algorithm by reducing the arithmetic load of the algorithm [8]. If \( n \in [0, N - 1] \) and \( N \) is not prime, \( N \) can be defined as \( N = \prod_{i=0}^{m-1} n_i \), where \( n_i \in [0, N_i - 1] < N \). This allows the usage of small ranges of \( n_i \) instead of large range of \( n \) and maps a function \( f(n) \) into a multidimensional function \( f'(n_1, n_2, \ldots, n_l) \).

There are two common methods of implementing index mapping: one-dimensional or multidimensional memory.
structures. In addition, it is also possible to implement the index mapping using several equal size one-dimensional memory structures. However, this option is not popular as it is uncomfortable for programming. The performance of modern computers is highly dependent on the effectiveness of the cache memory of the CPU [4]. To achieve the best performance of a series of memory elements, the best practice is to maintain sequential access [4]. Otherwise, the effectiveness of the cache memory of the central processing unit (CPU) will be reduced. Index mapping with multidimensional data structures violates sequential access due to switching between columns/rows and thus reduces the effectiveness of the cache memory. Therefore, it is generally accepted that the use of a multidimensional data structure reduces computer performance [4].

In this paper an efficient BRA is introduced to create the BRP based on multiple memory structures and recursive patterns of BRP. The findings of this paper show that the combination of multiple one-dimensional memory structures, index mapping, and the recursive pattern of BRP can be used to improve the efficiency of BRA. These findings are very important to the field of signal processing as well as any field that is involved in index mapping techniques.

2. Material and Methods

2.1. New Algorithm (BRA-Split). Elster stated that it is possible to generate the second half of the BRP by incrementing items in the first half by one [2] (Figure 1), without changing the order and the total number of calculations of the algorithm. Due to the recursive pattern of BRP, it can be further divided into equal size blocks by splitting each block recursively (Maximum log₂N times). After splitting s times, BRP is divided into 2^s equal blocks each containing n/2^s elements. The relation between the elements in blocks is given as follows:

\[ B(2^m + j)[i] = B(j)[i] + 2^{s-m-1}, \]  
(3)

where \(B(2^m + j)[i]\) is the \(i\)th element of block \(B(2^m + j)\) and \(m = 0, 1, \ldots, s-1, j = 1, \ldots, 2^m\).

Table 1 shows the relationship between elements in blocks according to the index mapping shown in (3), after splitting BRP one time and two times for \(n = 16\). Depending on the requirement, the number of splitting can be increased.

2.2. Evaluation Process of New Algorithm. To evaluate algorithms, we used Windows 7 and Visual C++ 2012 on a PC with multicore CPU (4 cores, 8 logical processors) and 12 GB memory. Detailed specifications of the PC and the software are given in Table 2. To eliminate limits of memory and address space related to the selected platform, the compiler option “/LARGEADDRESSAWARE” was set [21] and platform was set to “x64.” All other options of the operating system and the compiler were kept unchanged.

The new algorithm was implemented using single one-dimensional memory structure and the most common multidimensional memory structure. Furthermore, the new BRA was implemented using several equal size one-dimensional memory structures (multiple memory structure).


Table 1: Relation between elements of 16-element BRP \( (n = 16) \) for split = 1 and split = 2.

<table>
<thead>
<tr>
<th>Normal order</th>
<th>Reverse order</th>
<th>Split (( S = 1 )) Calculation method</th>
<th>Blocks ( (1, \ldots, 2^s) )</th>
<th>Index of the block ( i )</th>
<th>Reverse order calculation</th>
<th>Blocks ( (1, \ldots, 2^s) )</th>
<th>Index of the block ( i )</th>
<th>Reverse order calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks ( (1, \ldots, 2^s) )</td>
<td>Index of the block ( i )</td>
<td>Reverse order calculation</td>
<td>( m = 0, \ldots, s - 1 ) ((m = 0))</td>
<td>( j = 1, \ldots, 2^m )</td>
<td>( m = 0; j = 1 )</td>
<td>( i = 1, \ldots, n/2^{s-1} )</td>
<td>( i = 1, \ldots, 7 )</td>
<td>( B(2^m + j)[i] = B(j)[i] + 2^{s-m-1} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( 0 = 0 )</td>
<td>( 0 = 0 )</td>
<td>( 0 = 0 )</td>
<td>( 0 = 0 )</td>
<td>( 0 = 0 )</td>
<td>( B(2^0)[i] = B(0)[i] + 1 )</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
<td>( 8 = 8 + 0 )</td>
<td>( 8 = 8 + 0 )</td>
<td>( 8 = 8 + 0 )</td>
<td>( 8 = 8 + 0 )</td>
<td>( 8 = 8 + 0 )</td>
<td>( B(2^1)[i] = B(1)[i] + 2 )</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>( 4 = 4 + 0 )</td>
<td>( 4 = 4 + 0 )</td>
<td>( 4 = 4 + 0 )</td>
<td>( 4 = 4 + 0 )</td>
<td>( 4 = 4 + 0 )</td>
<td>( B(2^2)[i] = B(2)[i] + 2^{s-0-1} )</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>3</td>
<td>( 12 = 12 + 0 )</td>
<td>( 12 = 12 + 0 )</td>
<td>( 12 = 12 + 0 )</td>
<td>( 12 = 12 + 0 )</td>
<td>( 12 = 12 + 0 )</td>
<td>( B(2^3)[i] = B(3)[i] + 2 )</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>( 2 = 2 + 0 )</td>
<td>( 2 = 2 + 0 )</td>
<td>( 2 = 2 + 0 )</td>
<td>( 2 = 2 + 0 )</td>
<td>( 2 = 2 + 0 )</td>
<td>( B(2^4)[i] = B(4)[i] + 2 )</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>5</td>
<td>( 10 = 10 + 0 )</td>
<td>( 10 = 10 + 0 )</td>
<td>( 10 = 10 + 0 )</td>
<td>( 10 = 10 + 0 )</td>
<td>( 10 = 10 + 0 )</td>
<td>( B(2^5)[i] = B(5)[i] + 2 )</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>( 6 = 6 + 0 )</td>
<td>( 6 = 6 + 0 )</td>
<td>( 6 = 6 + 0 )</td>
<td>( 6 = 6 + 0 )</td>
<td>( 6 = 6 + 0 )</td>
<td>( B(2^6)[i] = B(6)[i] + 2 )</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>7</td>
<td>( 14 = 14 + 0 )</td>
<td>( 14 = 14 + 0 )</td>
<td>( 14 = 14 + 0 )</td>
<td>( 14 = 14 + 0 )</td>
<td>( 14 = 14 + 0 )</td>
<td>( B(2^7)[i] = B(7)[i] + 2 )</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>( 1 = 1 )</td>
<td>( 1 = 1 )</td>
<td>( 1 = 1 )</td>
<td>( 1 = 1 )</td>
<td>( 1 = 1 )</td>
<td>( B(2^8)[i] = B(8)[i] + 2 )</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1</td>
<td>( 9 = 9 + 1 )</td>
<td>( 9 = 9 + 1 )</td>
<td>( 9 = 9 + 1 )</td>
<td>( 9 = 9 + 1 )</td>
<td>( 9 = 9 + 1 )</td>
<td>( B(2^9)[i] = B(9)[i] + 2^{s-0-1} )</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>2</td>
<td>( 5 = 4 + 1 )</td>
<td>( 5 = 4 + 1 )</td>
<td>( 5 = 4 + 1 )</td>
<td>( 5 = 4 + 1 )</td>
<td>( 5 = 4 + 1 )</td>
<td>( B(2^{10})[i] = B(10)[i] + 2^{s-1} )</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>3</td>
<td>( 13 = 12 + 1 )</td>
<td>( 13 = 12 + 1 )</td>
<td>( 13 = 12 + 1 )</td>
<td>( 13 = 12 + 1 )</td>
<td>( 13 = 12 + 1 )</td>
<td>( B(2^{11})[i] = B(11)[i] + 2 )</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>4</td>
<td>( 3 = 2 + 1 )</td>
<td>( 3 = 2 + 1 )</td>
<td>( 3 = 2 + 1 )</td>
<td>( 3 = 2 + 1 )</td>
<td>( 3 = 2 + 1 )</td>
<td>( B(2^{12})[i] = B(12)[i] + 2 )</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>5</td>
<td>( 11 = 10 + 1 )</td>
<td>( 11 = 10 + 1 )</td>
<td>( 11 = 10 + 1 )</td>
<td>( 11 = 10 + 1 )</td>
<td>( 11 = 10 + 1 )</td>
<td>( B(2^{13})[i] = B(13)[i] + 2^{s-0-1} )</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>6</td>
<td>( 7 = 6 + 1 )</td>
<td>( 7 = 6 + 1 )</td>
<td>( 7 = 6 + 1 )</td>
<td>( 7 = 6 + 1 )</td>
<td>( 7 = 6 + 1 )</td>
<td>( B(2^{14})[i] = B(14)[i] + 2 )</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>7</td>
<td>( 15 = 14 + 1 )</td>
<td>( 15 = 14 + 1 )</td>
<td>( 15 = 14 + 1 )</td>
<td>( 15 = 14 + 1 )</td>
<td>( 15 = 14 + 1 )</td>
<td>( B(2^{15})[i] = B(15)[i] + 2 )</td>
</tr>
</tbody>
</table>
Table 2: Hardware and software specifications of the PC.

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i7 CPU 870 @ 2.93 GHz (4 cores, 8 threads)</td>
</tr>
<tr>
<td>RAM</td>
<td>12 GB, DDR3-1333, 2 channels</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>21 GB/s</td>
</tr>
<tr>
<td>L1, L2, and L3 cache</td>
<td>4 × 64 KB, 4 × 256 KB, and 8 MB shared</td>
</tr>
<tr>
<td>L1, L2, and L3 cache line size</td>
<td>64 bit</td>
</tr>
<tr>
<td>Brand and type</td>
<td>Fujitsu, Celsius</td>
</tr>
<tr>
<td>BIOS settings</td>
<td>Default (hyper threading enabled)</td>
</tr>
<tr>
<td>OS and service pack</td>
<td>Windows 7 professional with service pack 1</td>
</tr>
<tr>
<td>System type</td>
<td>64 bit operating system</td>
</tr>
<tr>
<td>OS settings</td>
<td>Default</td>
</tr>
<tr>
<td>Visual Studio 2012</td>
<td>Version 11.0.50727.1 RTMREL</td>
</tr>
<tr>
<td>.NET Framework</td>
<td>Version 4.5.50709</td>
</tr>
</tbody>
</table>

The next task was to identify a suitable data structure from different types of available data structures. We considered several common techniques as summarized in Table 3. Data structure 1 mentioned in Table 3 is not supporting dynamic memory allocation (need to mention the size of the array when array is being declared). For general bit reversal algorithm, it is a must to have dynamic memory allocation to cater different sizes of samples. Even after setting the compiler option “/LARGEADDRESSAWARE” [21], data structures 3 and 4 mentioned in Table 3 were not supported for accessing memory greater than 2 GB. Therefore, structures 1, 3, and 4 were rejected and memory structures 2 (array) and 5 (vector) were used to create all one-dimensional memory structures. The same versions of array and vector were used to create multidimensional memory structures.

The new algorithm mentioned in Section 2.1 was implemented using C++ in 24 types of memory structures as shown in Table 4. The performance of these algorithms was evaluated considering the “clocks per element” (CPE) consumed by each algorithm. To retrieve this value, first, average CPE for each sample size of $2^n$ where $n \in [21, 31]$ (11 sample sizes) were calculated after executing each algorithm 100 times. This gave 11 CPE representing each sample size. Finally, the combined average of CPE was calculated for each algorithm by averaging those 11 values along with “combined standard deviation.” The combined average of CPE was considered as the CPE for each algorithm. The built-in “clock” function of C++ was used to calculate the clocks. Combined standard deviation was calculated using the following:

$$
\sigma_c = \left( \frac{\sum_{i=1}^{k} n_i \sigma_i^2 + n_i \left( \overline{X}_c - \overline{X}_c \right)^2}{\sum_{i=1}^{k} n_i} \right)^{1/2},
$$

where \( \overline{X}_c = \sum_{i=1}^{k} n_i \overline{X}_i / \sum_{i=1}^{k} n_i \) is the number of samples, \( n_i \) is number of samples in each sample, and \( \sigma_i \) is the standard deviation of each sample.

Algorithms 1, 2, and 3 illustrate the implementation of new BRA with single one-dimensional memory structure, multidimensional memory structure, and multiple memory structures, respectively. The algorithm illustrated in Algorithm 1 (BRA_Split_1IA) was implemented using primitive array for split = 1. The algorithm BRM_Split_2_4A (Algorithm 2) was implemented using vectors for split = 2. The algorithm BRM_Split_2_4A (Algorithm 3) was implemented using primitive array for split = 2. A sample permutation filling sequence of algorithms with single one-dimensional memory structures is illustrated in Figure 2. Figure 3 illustrates a sample permutation filling sequence of both multidimensional and multiple memory structures.

Secondly, arithmetic operations per element (OPPE) were calculated for each algorithm. Arithmetic operations within each algorithm were located in three regions of the code: inner FOR loop, outer FOR loop, and outside of the loops. Then, the total number of operations (OP) can be defined as

$$
OP = K_1 \cdot C_1 + K_2 \cdot C_2 + C_3,
$$

where \( C_1 \), \( C_2 \), and \( C_3 \) are the number of operations in inner FOR loop, outer FOR loop, and outside of the loops. \( K_1 \) and \( K_2 \) are the number of iterations of outer loop and inner loop.

Equation (5) can be represented as

$$
OP = \sum_{i=0}^{\log_2(NS) - s - 1} 2^i \cdot C_1 + \sum_{i=0}^{\log_2(NS) - s - 1} C_2 + C_3,
$$

where \( NS \) is the number of samples and \( s \) is the number of splits.

The main contribution to calculations comes from the inner loop. Comparing with the contribution of operations in the inner loop, the contribution of operations in rest of the code is very small. For example, consider the algorithm BRA_Split_1IA shown in Algorithm 1. As sample size is \( 2^{31} \), \( K_1 \cdot C_1 = 2.15 \times 10^9 \cdot C_1 \) and \( K_2 \cdot C_2 + C_3 = 1000 \), where \( C_1 > 1 \). Therefore, only the operations of inner loop were considered for evaluation. Then (6) can be simplified as

$$
OP = \sum_{i=0}^{\log_2(NS) - s - 1} 2^i \cdot C_1.
$$

The “operations per element” (OPPE) can be defined as

$$
OPPE = \frac{\sum_{i=0}^{\log_2(NS) - s - 1} 2^i \cdot C_1}{NS}.
$$

For FFT always \( NS = 2^k; k \in \mathbb{Z}^+ \).

Then from (7)

$$
OPPE = \frac{\sum_{i=0}^{k} 2^i \cdot C_1}{2^k \sum_{i=0}^{k} 2^i + 1}.
$$
### Table 3: Common memory allocating methods that are used in Visual C++.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Syntax</th>
<th>Nature of memory layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Array</td>
<td>int BRP[1000]</td>
<td>Slot of continuous memory elements</td>
</tr>
<tr>
<td>2</td>
<td>Array</td>
<td>int* BRP = new int[n]</td>
<td>Slot of continuous memory elements</td>
</tr>
<tr>
<td>3</td>
<td>Array</td>
<td>array(int)*BRP = gcnew array(int)(n)</td>
<td>Collection of noncontinuous memory elements</td>
</tr>
<tr>
<td>4</td>
<td>ArrayList</td>
<td>ArrayList*BRP = gcnew ArrayList()</td>
<td>Collection of noncontinuous memory elements</td>
</tr>
<tr>
<td>5</td>
<td>Vector</td>
<td>std::vector⟨int⟩ BRP(n)</td>
<td>Collection of noncontinuous memory elements</td>
</tr>
</tbody>
</table>

### Table 4: Different versions of new BRA implemented with different data structures.

<table>
<thead>
<tr>
<th>Split (s)</th>
<th>Data structure type</th>
<th>Single memory structure</th>
<th>Multidimensional memory structure</th>
<th>Multiple memory structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Array (A)</td>
<td>BRA_Split_1_1A</td>
<td>BRA_Split_1_2DA</td>
<td>BRA_Split_1_2A</td>
</tr>
<tr>
<td>1</td>
<td>Vector (V)</td>
<td>BRA_Split_1_1V</td>
<td>BRA_Split_1_2DV</td>
<td>BRA_Split_1_2V</td>
</tr>
<tr>
<td>2</td>
<td>Array (A)</td>
<td>BRA_Split_2_1A</td>
<td>BRA_Split_2_4DA</td>
<td>BRA_Split_2_4A</td>
</tr>
<tr>
<td>2</td>
<td>Vector (V)</td>
<td>BRA_Split_2_1V</td>
<td>BRA_Split_2_4DV</td>
<td>BRA_Split_2_4V</td>
</tr>
<tr>
<td>3</td>
<td>Array (A)</td>
<td>BRA_Split_3_1A</td>
<td>BRA_Split_3_8DA</td>
<td>BRA_Split_3_8A</td>
</tr>
<tr>
<td>3</td>
<td>Vector (V)</td>
<td>BRA_Split_3_1V</td>
<td>BRA_Split_3_8DV</td>
<td>BRA_Split_3_8V</td>
</tr>
<tr>
<td>4</td>
<td>Array (A)</td>
<td>BRA_Split_4_1A</td>
<td>BRA_Split_4_16DA</td>
<td>BRA_Split_4_16A</td>
</tr>
<tr>
<td>4</td>
<td>Vector (V)</td>
<td>BRA_Split_4_1V</td>
<td>BRA_Split_4_16DV</td>
<td>BRA_Split_4_16V</td>
</tr>
</tbody>
</table>

Naming convention for algorithms: "BRA_Split" + <Number of splits> + <nature of memory structure>. xA, xV: x number of arrays and x number of vectors. xDA, xDV: single x-dimensional array and single x-dimensional vector.

```cpp
void mf_BRM_Split_1_1A (unsigned int ui_NS, int ui_log2NS)
{
    unsigned int ui_N = ui_NS;  // Number of samples
    unsigned int ui_t = ui_log2NS - 1;
    unsigned int ui_EB = ui_N/2;
    unsigned int ui_L = 1;
    unsigned int* BRP = new unsigned int[ui_N];
    BRP[0] = 0;
    BRP[ui_EB] = 1;
    for (unsigned int q = 0; q < ui_t; q++)
    {
        unsigned int ui_DL = ui_L + ui_L;
        unsigned int ui_N = ui_N/2;
        for (unsigned int j = ui_L; j < ui_DL; j++)
        {
            BRP[j] = BRP[j - ui_L] + ui_N;
            BRP[ui_EB + j] = BRP[j] + 1;
        }
        ui_L = ui_L + ui_L;
    }
    delete[] BRP;
}
```

**Algorithm 1:** C++ implementation of new BRA with single array for split = 1 (BRA_Split_1_1A).
Void mf_BRM_Split_2_4DV(unsigned int ui_NS, int ui_log2NS)
{
    unsigned int ui_N;
    unsigned int ui_EB;
    unsigned int ui_t;
    unsigned int ui_L;
    unsigned int ui_DL;
    ui_N = ui_NS; // Number of samples
    ui_t = ui_log2NS - 2;
    ui_EB = ui_N/4;
    ui_L = 1;

    std::vector<std::vector<unsigned int>> BRP(4, std::vector<unsigned int>(ui_EB));
    BRP[0][0] = 0;
    BRP[1][0] = 2;
    BRP[2][0] = 1;
    BRP[3][0] = 3;
    for (unsigned int q = 0; q < ui_t; q++)
    {
        ui_DL = ui_L + ui_L;
        ui_N = ui_N/2;
        for (unsigned int j = ui_L; j < ui_DL; j++)
        {
            BRP[0][j] = BRP[0][j - ui_L] + ui_N;
            BRP[1][j] = BRP[0][j] + 2;
            BRP[2][j] = BRP[0][j] + 1;
            BRP[3][j] = BRP[1][j] + 1;
        }
        ui_L = ui_L + ui_L;
    }
}

Algorithm 2: C++ implementation of new BRA with four arrays for split = 2 (BRM_Split_2_4A).

For large \( k \), \( 2^k + 1 \approx 2^k \). Then,
\[
\sum_{i=0}^{k} 2^i = 2^{k+1}. \tag{11}
\]

Because the considered sample size is \( 2^{21} \) to \( 2^{31} \), the value \( k \) can be considered as large. Then, from (9)
\[
\text{OPPE} = \frac{C_1 \cdot (2^{k-1})}{2^k}, \tag{12}
\]
\[
\text{OPPE} = \frac{C_1}{2^r}. \tag{13}
\]

According to (13), OPPE is \( f(C_1, s) \). The value \( C_1 \) (operations in inner loop) and the value \( s \) (number of splits) are a constant for a certain algorithm.

To evaluate the performance of new BRA, we selected three algorithms (LBRA, EBR, and BRA-Ru) which used a pattern instead of conventional bit reversing method. The performance of vector and array versions of the best version of new BRA was compared with the relevant versions of selected algorithms.

### 3. Results and Discussion

Our objective was to introduce BRA using recursive pattern of the BRP that we identified. We used multiple memory structures, which is a feasible yet unpopular technique to implement index mapping. According to Table 5, the numbers of operations in all the array and vector versions of both
Algorithm 3: C++ implementation of new BRA with four one-dimensional arrays for split = 2 (BRM_Split_2_4A).

Figure 3: Permutation filling sequence of 4 individual and single 4-dimensional memory structure for \( n = 16 \).

multidimensional and multiple memory structures are the same. Also, Figure 5 shows continuous decrement of OPPE when the number of splits increases. Then, the algorithm with the highest number of splits and the lowest number of operations is the one which is expected to be most efficient. However, results in relation with CPE (Figure 5) show that the new algorithm with four memory structures of array is the fastest and most consistent in the selected range. Two, four, eight, and sixteen multiple array implementations of new BRA reported 25%, 34%, 33%, and 18% higher efficiency,
respectively, in relation to the array version of LBRA. The algorithm with eight memory structures has nearly the same CPE as the four-array and four-vector versions, but is less consistent. On the other hand, the four-vector implementation of the new algorithm is the fastest and most consistent among all vector versions. Two, four, eight, and sixteen multiple vector implementations of new BRA reported 13%, 16%, and 16% higher and 23% lower efficiency, respectively, in relation to the vector version of LBRA. This result proves that at a certain point, multiple memory structure gives the best performances in the considered domain. Also, usage of multiple memory structures of primitive array is a good option for implementing index mapping of BRP compared to multi-dimensional or single one-dimensional memory structures.

Due to the flexible nature of the vector, it is commonly used for implementing algorithms. According to Figure 4 there is no difference in OPPE between array and vector versions. However, our results in Figure 5 show that the vector versions of BRA always required more CPE (44%–142%) than the array version. The structure of vector gives priority to generality and flexibility rather than to execution speed, memory economy, cache efficiency, and code size [22]. Therefore, vector is not a good option with respect to efficiency.

The results in Table 5 and Figure 4 show that there is no difference between the number of calculations and OPPE for equal versions of algorithms with multidimensional and multiple memory structure. Structure and type of calculations are the same for both types. The only difference is the nature of the memory structure: multidimension or multiple one-dimension. When CPE is considered, it shows 19%–79% performance efficiency from algorithms with multiple one-dimensional memory structures. The reason for that observation is that the memory access of multidimensional memory structure is less efficient than one-dimensional memory structure [22].

We agree with the statement of Elster about index mapping of BRP [2] and the generally accepted fact (the usage of multidimensional memory structures reduces the performance of a computer) [4] only with respect to multi-dimensional memory structures of vector. Our results show that even with multidimensional arrays there are situations where the new BRA performs better than the same type of one-dimensional memory structure. The four, eight, and sixteen dimensional array versions of new BRA perform 8%, 10%, and 2% in relation to one-dimensional array version of new BRA. Some results in relation to single one-dimensional memory structure implementation of new BRA are also not in agreement with the general accepted idea. For example sample size = $2^{31}$, the two-dimensional vector version of new BRA (BRA_Split_2DV) reported $5.42E-05$ CPE which is 389% higher in relation to average CPE of sample size range of $2^{21}$ to $2^{30}$. Also, the inconsistency was very high. Therefore, we excluded values related to sample size = $2^{31}$ for the two-dimensional vector version.

We observed very high memory utilization with the two-dimensional vector version, especially with sample size = $2^{31}$. Windows task manager showed that the memory utilization of all the considered algorithms was nearly the same for all sample sizes except for multidimension versions of vector. The multidimensional version of vector initially

---

**Figure 4:** Operations per element versus reference algorithms and new algorithm with different $s$ (splits), where dashed column corresponds to both array and vector versions of algorithm of Rudio, cross lines column corresponds to both array and vector versions of “Linear Bit Reversal” algorithm, dotted column corresponds to both array and vector versions of “Elster’s Bit Reversal” algorithm, vertical lines column corresponds to both array and vector versions of the new algorithm in single one-dimensional memory structure, inclined lines column corresponds to both array and vector versions of the new algorithm in single multidimensional memory structure, and horizontal lines column corresponds to both array and vector versions of the new algorithm in multiple one-dimensional memory structures.
utilizes higher memory and drops down to normal value. The results in relation to sample size $= 2^{31}$ showed that the extra memory requirement of two dimension vector was higher than that of the four-dimensional vector. Based upon it can be predicted that BRA_Split_1_2DV needs an extra 3 GB (total 13 GB) for normal execution at sample size $= 2^{31}$, but the total memory barrier of 12 GB of the machine slows the process down. The most likely reason for this observation is the influence of memory pooling mechanism. When defining a memory structure it is possible to allocate the required amount of memory. This is known as memory pooling [23]. In all the memory structures used in algorithms discussed in this paper we used memory pooling. Memory pooling allocates the required memory at startup and divides this block into small chunks. If there is no memory pooling, memory will get fragmented. Accessing fragmented memory is inefficient. When the existing memory is not enough for allocating, then it switches to use fragmented memory for allocating memory elements. In the considered situation, the existing memory (12 GB) is not sufficient for allocating the required amount (13 GB) which switches to use fragmented memory.

The total cache size of the machine is 8.25 MB, which is less than the minimum memory utilization of considered algorithms (16 MB to 8 GB) in relation to the sample size range from $2^{22}$ to $2^{31}$. Only the sample size $2^{31}$ occupies 8 MB memory, which is less than the total cache memory. Except BRA_Split_4_IV structure, all algorithms reported constant CPE in relation to the entire sample size range. The best algorithms of each category, especially, reported very steady behaviour. This observation is in disagreement with the statement of Karp “that a machine with hierarchical memory does not perform well when array size is larger than the cache size” [10].

Comparison (Figure 6) of best reported version in the considered domain (four memory structure version) and the selected algorithms shows that the array version of EBR performs the best. The four-array version of new BRA reported 1% lower performance than the array version of EBR. However, the four-array version of new BRA reported 34% and 23% higher performances than array versions of LBRA and BRA-Ru. Also, the four-vector version of new BRA is reported to have the best performance among all the vector versions. It reported 16%, 10%, and 22% performances compared to vector versions of LBRA, EBR, and BRA-Ru, respectively.

4. Conclusion and Outlook

The main finding of this paper is the recursive pattern of BPR and the implementation method of it using multiple memory structures. With multiple memory structures, especially, the newly identified index mapping performs much better than multidimensional or single one-dimensional memory structure. Furthermore, findings of this paper show that the performance of primitive array is higher than vector type. The result is in disagreement with the statement of Karp “that a machine with hierarchical memory does not perform well when array size is larger than the cache size.” Almost all the sample sizes we used were higher than the total cache size.
of the computer. However, multiple memory structure and the multidimensional memory structure versions showed reasonable steady performance with those samples. In general these results show the effects of data structures and memory allocation techniques and open a new window of creating efficient algorithms with multiple memory structures in many other fields where index mapping is involved.

The new bit reversal algorithm with $2^s$ independent memory structures splits the total signal into $s$ independent portions and the total FFT process into $s + 1$ levels. Then, these $s$ signal portions can be processed independently by means of $s$ independent processes on the first level. On the next level the results from the previous level stored in independent memory structures can be processed with $s/2$
processes and so on, until the last level. Therefore, we suggest using the concept of multiple memory structures in total FFT process along with the new algorithm with multiple memory structures and suitable parallel processing technique. We expect that it is possible to achieve higher performance from FFT process with proper combination of parallel processing technique and new algorithm compared to using the new algorithm only to create bit reversal permutation. Figure 7 shows such approach with four (when \( s = 2 \)) independent memory structures for sample size = 16.

**Conflict of Interests**

The authors declare that there is no conflict of interests regarding the publication of this paper.

**Acknowledgment**

This publication is financially supported by the University Grants Commission, Sri Lanka.

**References**


Submit your manuscripts at http://www.hindawi.com