Searching in an Efficiently Stored DNA Text Using a Hardware Solution

T. Berry, S. Keller and S. Ravindran

Department of Computer Science
Liverpool John Moores University
Byrom Street
Liverpool
United Kingdom

e-mail: T.Berry@livjm.ac.uk, S.Keller@livjm.ac.uk, S.Ravindran@livjm.ac.uk

Abstract. In this paper, we describe a storage method that reduces the size of a DNA text file to 25% of its original size. Also outlined is a new algorithm, which can search an input stream of DNA text for multiple DNA sub-strings in a single pass. Although this new algorithm is competitive when compared to the majority of existing string matching algorithms, the intention is to further improve performance by implementing the algorithm as a hardware-only solution.

1 Introduction

String matching and Compression are two widely studied areas in computer Science [10]. String matching is detecting a pattern P of length m in a larger text T of length n. Compression involves transforming a string into a new string which contains the same information but whose length is as small as possible. These two areas naturally lead to Compressed String Matching, i.e. searching for a pattern in a compressed text. This method will save both space and time.

In this paper we describe a hardware solution that searches in the compressed DNA text. We also describe an algorithm coded in the programming language C that will be synthesized into hardware. A DNA text (or molecule) encodes information which by convention is represented as a string over the DNA alphabet A, C, G and T. Compressed String Matching in a DNA text is useful for the following reasons. Although the cost of memory is reducing, the sizes of DNA databases are growing exponentially.

Optimal compression will devote two bits to represent each DNA character, if each character is drawn uniformly at random from the DNA alphabet and that all positions in the text are independent [14]. The compression method described in Section 2 also devotes two bits per character, i.e. the method guarantees to compress the DNA text to 25% of its original size. Section 3, outlines the BK algorithm, as being a string matching algorithm, which as well as being relatively fast as a software solution, could also be implement in a hardware-only solution. Section 4, describes a modification to the basic BK algorithm, which will search a stream of

DNA text for multiple sub-strings in a single pass of the text. Section 5, covers the process of implementing programs as hardware-only solutions. Attention is paid to the inadequacies of modern microprocessors and the advantages which so-called 'hardware compilation techniques' can offer as a means of accelerating the execution of algorithms. Section 6, describes how the BK string matching algorithm may be implemented as a hardware only solution. In section 7 we describe 5 existing string matching algorithms. In section 8 we compare our new algorithm with the 5 existing algorithms by experimentation. The texts and the patterns used for these experiments have been taken from [11] and [1] respectively.

2 Efficient storage of a DNA text

In the DNA alphabet, S, there are four characters, namely A, C, G and T. As there are only 4 possible characters in a DNA text we can represent the character's with the function, f: $\Sigma \Rightarrow [0 .. 3]$, such that:

$$\text{f(A)} = 0, \text{ f (C)} = 1, \text{ f (G)} = 2, \text{ and f (T)} = 3.$$

Let a block be a string of four characters. The code of a block of DNA characters is the value that returned by the function g, g: $\Sigma \times \Sigma \times \Sigma \times \Sigma \Rightarrow [0 .. 255]$, for the block. The function g is defined as follows. $g(\alpha\beta\gamma\delta) = (f(\alpha) \times 4^3) + (f(\beta) \times 4^2) + (f(\gamma) \times 4^1) + (f(\delta) \times 4^0)$

This means that we can represent each of the DNA characters with 2 bits. Namely A = 00, C = 01, G = 10 and T = 11. A DNA text block will be represented by 32-bits, as each character needs 8-bits. Using the function g we can represent a text block with 8-bits. For each text block we print an ASCII character whose ASCII number is the value return by the function g. As the function g is a bijective function, we can compress any text block into 8-bits and it is possible to reconstruct the original DNA text exactly.

For example, CAAGAGCGCAGT $\Rightarrow$ 010000100010011001001011 $\Rightarrow$ 66 38 75 $\Rightarrow$ B&K. So we can store the string CAAGAGCGCAGT using 24 bits. This storage method will guarantee to store the DNA text in a file, which is 25% of its original size.

3 Investigation into a hardware only solution to the string matching problem

The string matching algorithm illustrated in Figure 1 was devised as part of a case study to investigate the feasibility of performing computational algorithms in hardware. String matching was chosen as one of the areas to be tested as such algorithms typically involve many hardware manipulations of words of binary data. These manipulations are invoked by the machine code instructions, which constitute the program and performed by the general-purpose hardware within the microprocessor itself. So called software to hardware synthesis techniques aim to accelerate algorithm execution by first of all removing the need for machine instructions and by also performing computational and logical operations on bespoke hardware.

```
while (match != 0 && word_count != 0) {
      result = current & mask;
      match  = result - target;
      if (match != 0) {
         current = current >> 2;
         temp = buffer << 14;
         current = current | temp;
         if (shifted == 7) {
            word_count--;
            shifted = 0;
            buffer = *ptr;
            ptr++;
         }
         else {
            buffer = buffer >> 2;
            shifted++;
         }
      }
   }
```

Figure 1: The C code for searching for occurrences of a single pattern in a given text

The example code shown works on a word size of 16 bits and can detect a pattern of up to 8 DNA characters in length. However, the algorithm is by no means limited to this word size.

The algorithm works by shifting the input stream through the variable *current*. When the data is shifted, it is shifted two bits at a time to the right. It is shifted two bits at a time because this is more efficient as the algorithm are searching for DNA features which are encoded into two bit patterns. Each time *current* is shifted to the right it is checked for a match with the target pattern. This concept is illustrated in Figure 2.

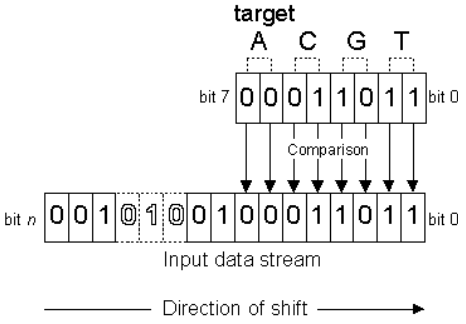

Figure 2: Comparison of input stream against target

When shifted, the two least significant bits (LSBs), which are bits 1 and 0, are lost and the two most significant bits (MSBs), which are bits 15 and 14, become empty. These two null MSBs are filled with the two LSBs of $buffer$. The variable $buffer$ is a pre-fetch word, which will contain word i+1 with *current* containing word i. This is necessary if *current* is to kept full at all times. During initialisation, the first word of data is copied to *current* from the input buffer and $buffer$ is filled with the second word of data.

In order to copy the two LSBs of $buffer$ to the two MSBs of *current*, $buffer$ is first copied to a variable *temp*, which is then shifted 14 bits to the left. This shift operation results in the two least significant bits of buffer (1 and 0) being moved to

the two most significant bits (15 and 14), with the remainder of the word (bits 13 to 0) being filled with 0's. The contents of *temp* is then ORed with *current* resulting in the two most significant bits of *current* being replaced with the two least significant bits of $buffer$.

In order to make sure that $buffer$ always has at least two bits available for *current*, a count is kept of how many times *current* has been shifted to the right. This count is stored in the variable $shifted$, which is initialised to 0 and then incremented each time $shifted$ is shifted to the right and the two MSBs replaced with the two LSBs of $buffer$. If after a comparison $shifted$ is less than 7, then $buffer$ is shifted two bits to the right in order to replace the two LSBs which have been moved to *current* and the variable $shifted$ is incremented. If shifted reaches 7, then the last two bits of data have moved from $buffer$ to *current* and $buffer$ requires re-filling. When this occurs, $shifted$ is set back to 0 and $buffer$ is loaded with a complete new word from the input stream.

The next byte to be fetched from the input stream is pointed to by the pointer variable *ptr*, which is incremented once $buffer$ has been refilled with a word from data buffer named $data\_buffer$.

To ascertain whether *current* contains a match for the bit pattern being searched for, *current* is first ANDed with a variable named *mask*. The purpose of *mask* is to mask out those bits of *current* which are not required for the comparison. To ignore a bit during the comparison between *target* and *current*, then the associated bit of *mask* should be 0. Likewise, to include a bit in the comparison, then that bit of the mask should be set to 1. As illustrated in Figure 3 below, the pattern 'ACGT' is being searched for, which is only an 8 bit pattern. Hence the remaining upper eight bits can be ignored during the comparison and are thus set to 0.

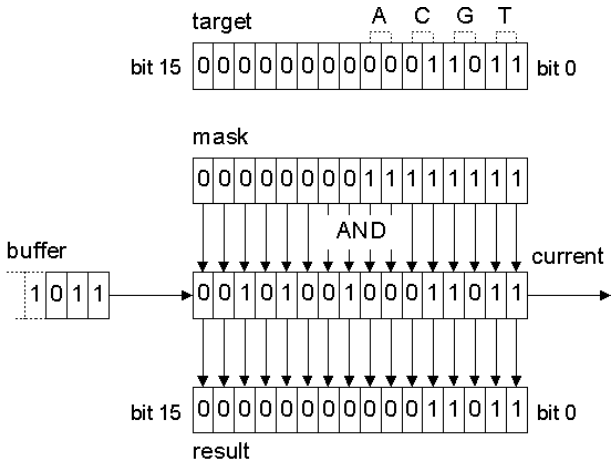

Figure 3: The use of the mask to reduce the number of bits compared

When *current* is ANDed with the mask, the result of the logical AND is stored in *result*. A bit of *result* will only be set to 1 if both the corresponding bits of *mask* and *current* are 1, otherwise the bit will be set to 0. A match with the target can now be determined by subtracting *target* from *result*. If the result of this subtraction is all 0's, then both result and the target must have contained the same values and hence a match has been found. This process is illustrated in Figure 4.

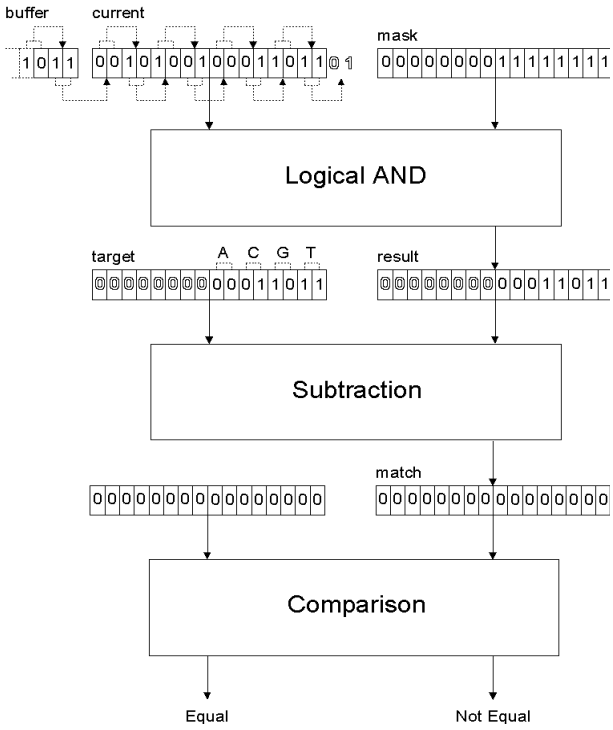

Figure 4: The steps required to determine whether the target matches the current data

The program has been written to locate patterns of DNA up to and including eight two bit codes. Hence, all words are 16bits in length and are declared as being of type *unsignedshort*. However, the program could easily be amended to locate longer patterns by simply changing the variable types and program constants.

4 Searching for multiple strings

The example algorithm illustrated in Figure 1, simply searches an input stream for all occurrences of a single string. The program can be easily modified to search an input stream for all occurrences of many strings by reading in many targets from a file and storing them in an array. This way, each time *current* is shifted, it may be compared with many targets before it is once more shifted. In order to do this, a second array must be created to store the masks for each of the targets. These masks may be automatically generated from the targets as they are read in.

```
while (shifts>0) {
      for (i=0; i<no_of_targets; i++) {
         result = current & mask_array[i];
         match  = result - target_array[i];
         if (match == 0) {
          .. match found
         }
      }

      current = current >> 2;
      shifts--;
      temp = buffer << 14;
      current = current | temp;

      if (shifted == 7) {
         shifted = 0;
         buffer = *ptr;
         ptr++;
      }
      else {
         buff = buff >> 2;
         shifted++;
      }
   }
```

Figure 5: An algorithm to search for multiple patterns in a single text

Apart from this simple modification, the program remains relatively unchanged. This is the version of the program, which will be the subject of the investigation into hardware acceleration of string matching.

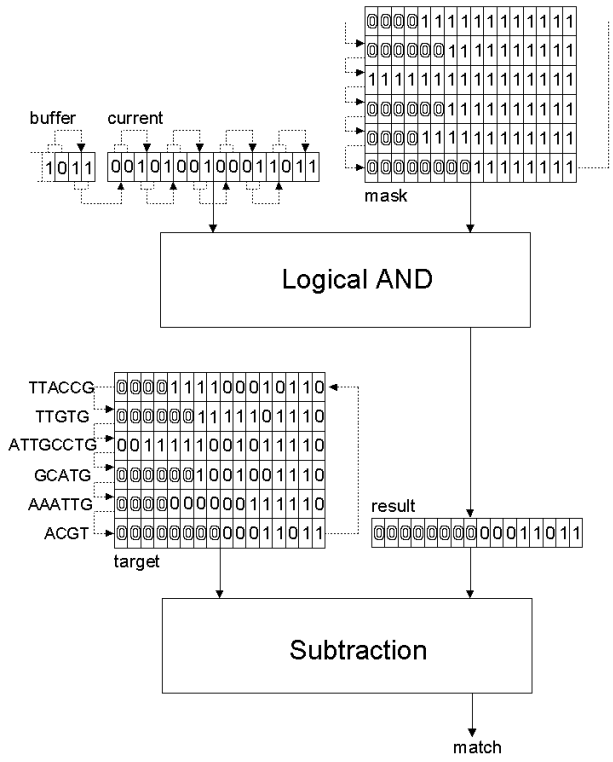

Figure 6: Illustration of Figure 5

5 Hardware acceleration

Over the past decade hardware synthesis has been explored as a method of accelerating computing tasks at which conventional general-purpose microprocessors are inefficient. The problem is that current microprocessors, although being suitable for many tasks, are not particularly efficient at performing any one task. This is because they are designed to be applicable to as many problem areas or tasks as possible. Therefore, through necessity they possess many features which although utilised by one application may never be used by another application. Another problem with conventional machines is the stored program concept whereby and algorithm is executed by the microprocessor obeying a series of commands, which are stored in memory. These commands are the machine code instructions, which the microprocessor fetches, decodes and then executes one at time. This fetching and decoding takes comparatively vast amount of time due to the slow speed of memory and the numerous instructions within the instruction set of the processor. Even the execution phase is by no means efficient. The execution circuits of a processor are finite and although some resources are replicated, many must be shared. This resource contention slows execution times. Additionally, the execution circuits of microprocessors are designed to perform many tasks, making them less efficient.

Hardware and software co-design or hardware to software synthesis is a process whereby computing algorithms expressed in high-level languages, are compiled to produce either an executable program and a hardware circuit design or just a hardware circuit. In the case of hardware and software co-design [16, 17], the majority of the program is turned into an executable binary for execution on a microprocessor, whilst the remainder of the algorithm is synthesised to hardware. The portion synthesised to hardware would be the section of the algorithm at which the microprocessor would be least efficient. The hardware portion is usually programmed into a Field Programmable Gate Array (FPGA) [18], which then acts as a co-processor to the host microprocessor. Producing programs for such architectures is usually performed using a hybrid programming language and appropriate compilers and synthesis tools [15]. Such programming languages tend to be based on C, with extensions being added to express the hardware-only components for the FPGA.

With pure software to hardware synthesis [2, 3], an attempt is made to map the entire algorithm into an FPGA, resulting in a digital circuit, which is functionally identical to and directly derived from an algorithm, which was originally expressed in a programming language. Such approaches tend to used hardware description languages such as VHDL [13], which are exclusively used for expressing the function of hardware circuits.

Synthesis to a hardware only solution offers the greatest potential increase in speed, removing the need for instructions and a conventional fetch-decode-execute cycle. However, it is also the most difficult to achieve. The difficulty arises from the design features of current FPGAs, which were originally intended for implementing digital circuits. Although suitable for the prototyping and implementation of general circuits comprising of digital logic, they are not well suited for implementing algorithms. This is because algorithms require data storage for variables, buses for register to register and register to execution unit transfers. Data storage and buses are not available within an FPGA and must be created using the FPGAs resources,

such as macro-cells and signal lines. What makes the situation worse is that both registers and buses are expensive in terms of FPGA resources, which ultimately limits the size of the algorithm to may be implemented in hardware.

As part of the research into implementing string matching algorithms in hardware-solutions, recommendations will be made regarding the development of a new FPGA architecture, which will be more suited to purpose of implementing software in hardware.

6 Hardware Implementation of string Matching

The research currently being undertaken aims to overcome the limitations of current FPGAs, with regards to configurable computing. First of all, it aims to do this by recommending a new configurable device architecture, which lends itself more to the mapping of software to hardware. The device will feature the busing systems, areas of storage and synchronization circuits required to facilitate both effective and efficient hardware generation. Secondly, software tools are being developed which will process standard C programs and as their output, will produce configuration files for the programmable device.

Because of the low-level nature of the task of string matching, it is an ideal candidate for such acceleration techniques. At the hardware level, the most efficient method of searching a string for a sub-string is as illustrated in Figure 2. The stream to be searched is passed through a register, shifting one bit at a time. Each time the register is shifted, the register is compared with the sub-string being searched for. This is the same method as employed in the C algorithm discussed previously. The number of register bits to be compared need only be equal in length to the number of bits in the sub-string, with any additional bits simply be masked out or ignored in the same way as the C algorithm. Additionally, the register being searched need not only be shifted one bit at a time. In the case of searching for occurrences of bit patterns consisting of two bit sub-patterns, it is more efficient to shift the register two bits before a comparison with the target is made.

Figure 7., illustrates a simplified diagram of the components to be implemented in hardware. Missing are the hardware components responsible for shifting both *current* and $buffer$ to the right. Also missing are the circuits required for synchronizing the activities of the components in order to perform the operations of the algorithm in the correct order.

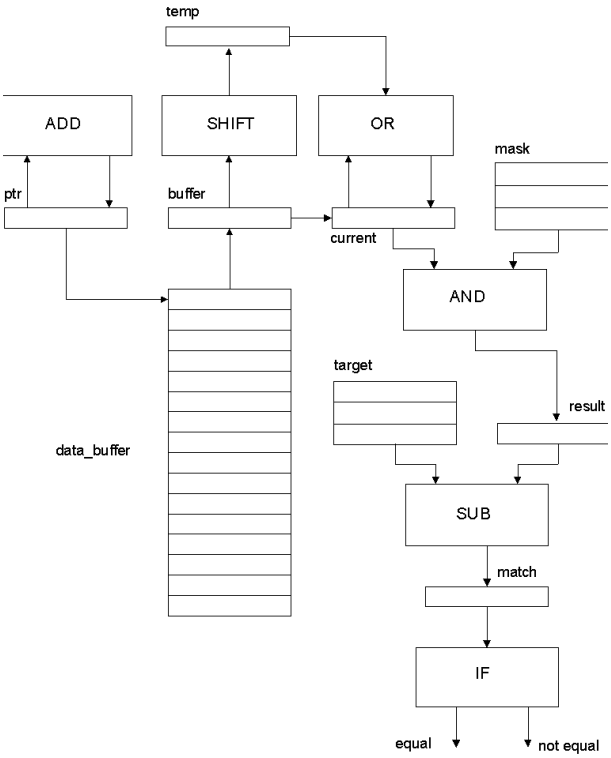

Figure 7: Simplified version of the components to be implemented in hardware

The memory labelled $data\_buffer$ holds the data to be searched for a sub-string. The width of the words contained in $data\_buffer$ is immaterial and may be of any width.

The registers labelled ptr and $buffer$ are associated with the fetching of the words from memory. The register $buffer$ is the same width as the words contained in $data\_buffer$. This register is used to contain a pre-fetch word. The register ptr is used as a pointer to reference the words contained in $data\_buffer$. As such, its width need only be sufficient to reference all of the words in $data\_buffer$.

The register $current$ contains the current bit pattern to be matched against a sub-string bit pattern. It is a shift register, with the data contained in the register being shifted right two bits at a time, with the two least significant bits being lost and the two most significant bits being replaced with the two least significant bits of buffer. This is the purpose of buffer, to keep current full of bits. Only once all the bits contained in buffer have been shifted into current, will new data be loaded into $buffer$ from $data\_buffer$.

As with the C algorithm, the mask register is used to contain a bit pattern to mask off the bits of $current$, which are not to be compared. When ANDed with the contents of $current$, then the resulting word is stored in the register $result$. It is the contents of $result$, which will be compared with the target to determine whether or not a matching bit pattern has been located. To ascertain whether the contents of result and target do match, result is subtracted from target. Again, if the result of the subtraction is zero, then a match has been located.

The synchronisation techniques to be implemented to synchronise the functioning of the component parts is beyond the scope of this paper. However, the techniques employed and the architecture of the programmable logic device, will be reported

upon in subsequent papers.

7 Existing string matching algorithms

The string matching algorithms described below work as follows. First the pattern of length m, P[1..m], is aligned with the extreme left of the text of length n, T[1..n]. Then the pattern characters are compared with the text characters. The algorithms vary in the order in which the comparisons are made. After a mismatch is found the pattern is shifted to the right and the distance the pattern can be shifted is determined by the algorithm that is being used. It is this shifting procedure, which is the main difference between the string matching algorithms.

There are a number of string matching algorithms available in the literature. We have chosen six of them, which were found to be fast in [5] and described them briefly below. All of the algorithms have worst-case search time O(nm). Animations of these algorithms can be found at [9] and more information about the algorithms can be found in [8].

In the Boyer-Moore (BM) algorithm [7] the characters are compared from right to left starting with the rightmost character of the pattern. In a case of mismatch it uses two functions, last occurrence function and good suffix function and shifts the pattern by the maximum number of positions computed by these functions. The good suffix function returns the number of positions for moving the pattern to the right by the least amount, so as to align the already matched characters with the rightmost substring in the pattern that are identical. The number of positions returned by the last occurrence function determines the rightmost occurrence of the mismatched text character in the pattern. If the text character does not appear in the pattern then the last occurrence function returns m.

The Horspool (HOR) algorithm [12] is a simplification of the BM algorithm. It does not use the good suffix function, but uses a modified version of the last occurrence function. The modified last occurrence function determines the right most occurrence of the $(k+m)^{th}$ text character, T[$k+m$] in the pattern, if a mismatch occurs when a pattern is aligned with T[k .. $k+m$]. This algorithm changes the order in which characters of the pattern are compared with the text. It compares the rightmost character in the pattern first then compares the leftmost character, then all the other characters in ascending order from the second position to the $m-1^{th}$ position.

The Berry-Ravindran (BR) algorithm [5] uses the next two characters outside the pattern text alignment T[$k+m+1$] T[$k+m+2$] to calculate the shift. If the pair is in the pattern then we shift the pattern so as to align it with the rightmost occurrence of the pair in the pattern. If the pair is not in the pattern then we shift by m+2 places to the right.

The DS algorithm [6] is an algorithm designed to search directly in the efficiently stored DNA text. It was found to be the fastest algorithm for the task of string matching in DNA files. The speed of the algorithm was mainly due to the cut down in the time required to scan in the text due to it being 25% of the size of the original text. The DS algorithm has a worst case run time of O(nm) but an average case run time of O($n+m$). The algorithm compares text blocks with pattern blocks directly to see if they match. Upon a mismatch the algorithm moves to the next text block to be considered.

The Shift-OR (SO) algorithm [4] has a worst case run time of O(n) independent of the size of the alphabet being used or the pattern being searched for. The SO algorithm constructs a bit array R of length m. The array has the initial state R_i and $R_i[j]$ is equal to 0 if P[0,j] = T[$i-j$,i] for all $0 \leq j \leq m$. Otherwise $R_i[j]$ is equal to 1. R_i is recalculated to form $R_i + 1$ by using two operations a logical shift of 1 and a logical OR hence the name of the algorithm Shift-OR.

8 Comparison with existing algorithms

We measure the user time for the six algorithms. We timed the search of each of the 5 texts randomly chosen from the Entrez database [11] for all occurrences of the 62 enzyme cutting boundaries in [1]. There are 9 patterns of length 4, 50 of length 6 and 3 of length 8. The BK and DS algorithms searched in the efficiently stored DNA text file and the BM, HOR, BR and SO algorithms searched in the original DNA text file. We used a 486-DX66 with 32 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. The user time includes the time taken for any pre-processing and the reading of the text into memory. Each algorithm was evaluated ten times and the average user time taken is given in Table 1. The timings were accurate to 1/100 of a second. The difference between the slowest and fastest time for each test for an algorithm was less than 0.1 of a second.

Text	Text size	BM	BR	HOR	SO	DS	BK
1	100,000	47.57	31.49	41.09	54.92	15.31	45.77
2	100,000	47.63	31.46	40.99	54.91	15.36	45.76
3	253,505	119.97	79.29	102.97	138.96	33.18	115.92
4	319,000	151.13	99.63	129.70	174.71	40.84	145.83
5	217,000	102.72	67.98	88.26	118.85	29.05	99.22

Table 1: The user time taken (given in seconds) to search for all 62 patterns in each of the texts

¿From Table 1 we can see that the DS algorithm is the fastest algorithm for the task. This is due to the savings made by the algorithm searching in the compressed DNA file, which is a quarter of the size of the original DNA text file. The BR algorithm is the best algorithm for searching in the original DNA text file. This is due to the larger shift of m+2 given by this algorithm. Using two characters to perform the search means that the probability of a large shift is increased. We would expect the average shift for the algorithm to be greater than m for all the patterns searched for. The BK algorithm is faster than the BM and the SO algorithms. The BK algorithm is a C implementation of our proposed hardware solution. We expect our hardware solution to be faster than our C implementation, which will also be faster than the DS algorithm.

9 Conclusion

Using the storage method described in Section 2 we can store DNA text files in 25% of space required for the original DNA text file. Using algorithms such as the DS and BK algorithm we can keep DNA texts efficiently stored and perform searches on them. Thus saving both time and space.

Although the BK algorithm, which is presented in this paper, is not the fastest algorithm for the task of string matching in an efficiently stored DNA text file, it is never-the-less still competitive when compared to existing string matching algorithms. Although it is by no means the fastest algorithm for sub-string searches, the hardware synthesis of the BK algorithm into a hardware only implementation is expected to produce a solution that we estimate to be significantly faster than even the DS algorithm.

References

References

[1] Amersham life science products catalogue, pp 378-379, 1998.

[2] James B. Peterson, R. Brendan O'Connor, Peter M. Athanas, *"Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures"*, The Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[3] James B. Peterson, Peter M. Athanas, *"High-Speed 2-D Convolution with a Custom Computing Machine"*, The Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[4] Baeza-Yates R. A., Gonnet G. H., *"A New Approach to Text Searching"*, Communications of the ACM, 35(10), pp. 74-82, 1992

[5] Berry T., Ravindran S., *"A fast string matching algorithm and experimental results"*, Prague Stringology Club Workshop '99, 1999.

[6] Berry T., Ravindran S., *"String matching in a compressed DNA text"*, Proceedings of the Australian Workshop on Combinatorial Algorithms (AWOCA '99), pp. 53-62, 1999.

[7] Boyer R. S., Moore J. S., *"A fast string searching algorithm"*, Communications of the ACM, 23(5), pp 1075-1091, 1977.

[8] Charras C., Lecroq T., 1997, Exact string matching, available at: `http://www-igm.univ-mlv.fr/~lecroq/string.ps`

[9] Charras C., Lecroq T., 1998, Exact string matching animation in JAVA available at: `http://www-igm.univ-mlv.fr/~lecroq/string/`

[10] Crochemore M., Rytter W., *"Text algorithms"*, Oxford University Press, 1994.

[11] Entrez database available at: http://www.ncbi.nlm.nih.gov/Entrez/

[12] Horspool R. N., *"Practical fast searching in strings"*, Software Practice and Experience, 10(6), pp 501-506, 1980.

[13] *"IEEE Standard VHDL Language Reference Manual"*, The Institute of Electrical and Electronics Engineers, Inc. (1987)

[14] Loewenstern D., Yianilos P., *"Significantly lower entropy estimates for natural DNA sequences"*, Journal of Computational Biology, 6(1), 1999.

[15] Page I., Luk W., *"Compiling occam into FPGAs"*, in FPGA, eds., Will Moore and Wayne Luk, 271-283, Abingdon EE&CS books, (1991).

[16] Page I., *"Constructing Hardware-Software Systems from a Single Description"*, Oxford University Computing Laboratory.

[17] Page I., Aubury M., Randall G., Saul J., Watts R., *"hcc: A Handel-C Compiler"*, Oxford University Computing Laboratory.

[18] Xilinx Inc., *"Spartan and SpartanXL Families of Field Programmable Gate Arrays"*, Preliminary Product Specification". San Jose, CA, (1999)