TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2015) 23: 1405 – 1417
© TÜBİTAK
doi:10.3906/elk-1304-165

Fast bitwise pattern-matching algorithm for DNA sequences on modern hardware

Gıyasettin ÖZCAN[1,*], Osman Sabri ÜNSAL[2]
[1]Department of Computer Engineering, Dumlupınar University, Kütahya, Turkey
[2]BSC-Microsoft Research Center, Barcelona, Spain

Received: 17.04.2013 • **Accepted/Published Online:** 08.09.2013 • **Printed:** 28.08.2015

Abstract: We introduce a fast bitwise exact pattern-matching algorithm, which speeds up short-length pattern searches on large-sized DNA databases. Our contributions are two-fold. First, we introduce a novel exact matching algorithm designed specifically for modern processor architectures. Second, we conduct a detailed comparative performance analysis of bitwise exact matching algorithms by utilizing hardware counters. Our algorithmic technique is based on condensed bitwise operators and multifunction variables, which minimize register spills and instruction counts during searches. In addition, the technique aims to efficiently utilize CPU branch predictors and to ensure smooth instruction flow through the processor pipeline. Analyzing letter occurrence probability estimations for DNA databases, we develop a skip mechanism to reduce memory accesses. For comparison, we exploit the complete *Mus musculus* sequence, a commonly used DNA sequence that is larger than 2 GB. Compared to five state-of-the-art pattern-matching algorithms, experimental results show that our technique outperforms the best algorithm even for the worst-case DNA pattern for our technique.

Key words: Computer architecture, bitwise string match, short DNA pattern, packed variables, 32-bit word

1. Introduction

Recent developments in biology led to an explosion in the data stored in sequence databases. As a consequence, extractions of relevant patterns from databases have become an important research topic. Some of the recent examples of relevant research could be found in [1,2].

The scope of pattern-matching is wide. On one hand, approximate pattern-matching or multiple pattern-matching studies consider complex but crucial problems that require high-performance processors. On other hand, recent exact pattern-matching algorithms aim to enhance the search speed and minimize hardware usage and power consumption.

In terms of approximate pattern searching, recently [3] presented a parallel suffix tree construction algorithm with wildcards. The algorithm divided the text into multiple segments. Afterwards, the segments were indexed by suffix trees in parallel and searched. In general, suffix trees typically have two drawbacks. These are poor memory locality and large memory footprint.

In terms of multiple-string matching, one recent example was presented in [4], which implemented a Bloom filter to search thousands of patterns simultaneously in a text. The technique computes hash functions utilizing Bloom filters for each pattern and text frame and stores them inside a large bit vector. However, Bloom filters may cause false positives. Therefore, all positive results should be checked before drawing the final conclusion. Inevitably, multiple-string matching is computationally demanding while having a large memory footprint.

*Correspondence: giyasettin.ozcan@dpu.edu.tr

It is a fact that approximate-string matching or multiple-string search techniques are critical during genome analysis although they are computationally hard problems. Therefore, novel faster and reliable techniques are required.

Another branch of studies considered exact pattern-matching; this is also our focus in this paper. The exact matching problem is to find all occurrences of a pattern of interest inside a given text. The alphabet of the text may vary. However, some of the most common alphabets are DNA bases and amino acids. In this study, we consider texts that contain DNA sequences. We denote the text, T, as

$$T = t_1, t_1, \ldots, t_n, \tag{1}$$

and the pattern, P, is represented as

$$P = p_1, p_1, \ldots, p_m. \tag{2}$$

The literature denotes various exact pattern-matching algorithms. Two of the recent ones were presented in [5,6]. Both algorithms are composed of preprocess and search phases and determine efficient skip distances when a mismatch occurs. During the preprocess, the pattern is analyzed and, correspondingly, a shift distance table is generated. In order to determine an efficient skip distance, [5] presented an algorithm that determines the comparison order of pattern and text frame indices. The algorithm gives priority to the leftmost and rightmost pattern indices for comparison. The third and the rest of the priorities are held by the leftmost not-compared letters. Nevertheless, the algorithm requires three nested loops during implementation, which could increase the branch misprediction rate. In addition, the algorithm does not present a bitwise algorithm solution where bitwise operators could be executed very fast on modern processors. On the other hand, [6] combined two early pattern-matching algorithms and designed a switch mechanism between them. However, the results denoted that the algorithm does not outperform for DNA patterns where alphabet size is only four. In contrast, the algorithm yields better results for amino acid sequences where the alphabet size is 20.

Since biological data volumes are very large, pattern searches on DNA sequences require efficient computation. If a string-matching algorithm is not optimized for modern processor architectures, its associated memory operations and data comparisons can cause inefficiency.

An architecture-friendly text search technique is the class of bitwise text search algorithms. These algorithms exploit intrinsic parallelism of bitwise operators. Typically, pattern and text are represented in bitwise form while comparisons are implemented through efficient bitwise operation instructions of contemporary processors.

In general, bitwise matching handles generally short-length patterns due to limitations of computer word size. However, short length pattern searches can also be very useful for long patterns, since they reduce the search space considerably.

We group the bitwise matching algorithms into five distinct base classes of algorithms: exploiting intrinsic parallelism of bitwise operators, emulation of automata on bit strings, sliding alignment matrix over text, bitwise bad character heuristics, and bitwise hash. The seminal implementations of these bases are, respectively, Shift OR [7], BNDM [8], BLIM [9], Horspool [10], and Hash [11]. We now briefly discuss each algorithm base.

The Shift OR algorithm makes use of the intrinsic parallelism capabilities of bitwise operators. In each step, the algorithm reads a character from the text and consequently leaves its trace inside the bits of an integer. As a result, bit organization of the integer indicates whether a match occurred or not.

Following studies indicated that lack of a skip mechanism is the main deficiency of Shift OR. Concretely, the original algorithm does not predict unfruitful pattern indices. This may cause significantly larger numbers of memory accesses and therefore the run time of the algorithm could slow down drastically for large sequence sets. In terms of bitwise algorithms, BNDM presents one of the most compact algorithms that features coherence among match and skip procedures. During the match phase, a mask counter ensures a maximum reliable skip distance when a mismatch occurs. However, the code structure of BNDM is more complex than that of Shift OR, since handling the skip mechanism puts in extra conditions. The characteristic of complex multilevel nested loops in BNDM could lead it to perform suboptimally in modern processor architectures: it could suffer from increased branch mispredictions due to complex control flow as well as cache misses due to reduced locality.

A recent algorithm looks at the problem from a different perspective. BLIM targets the pattern length limitations of the bitwise algorithm. During preprocessing, BLIM computes a mask matrix, which helps to determine match indices. In order to incorporate a skip mechanism, BLIM exploits the Sunday algorithm [12].

Variations of Shift OR [13–15], BNDM [13,14], and BLIM [16,17] exist in the literature. However, each modification of base algorithms not only causes improvement, but also causes extra complexities. Finding a balanced trade-off is generally difficult.

Since the DNA alphabet consists of four letters, each DNA letter can be represented by two bits. The technique is defined as two-bit enumeration [18]. Two-bit enumeration of DNA enables us to implement additional bitwise string-matching algorithms, which are derivates of general string-matching. For instance, hash-based techniques and bad character heuristics can be implemented when two-bit DNA enumeration is used.

A hash mechanism is able to map bitwise enumerated patterns and text fragments into fixed bit strings. By two-bit enumeration, each pattern and text letter is replaced by two-bit equivalents. In this respect, pattern and candidate text indices can be compared by bitwise operators that are directly supported by the processor.

The simple form of a bitwise hash technique does not have a skip mechanism. While the Karp–Rabin algorithm embeds a skip mechanism on hash, it is still inferior to other faster pattern search algorithms.

An early study presented bad character heuristics as a general solution to exact match problems [19]. During bad character heuristics, if a mismatch occurs, the pattern will be shifted to the right until the rightmost character is aligned to a match case. Although the original application is not bitwise, it can be implemented for two-bit enumeration of DNA.

Our literature analysis shows that the researchers who work on string-matching generally develop their algorithms without considering processor architecture, or they consider architecture to a limited degree as described above. What is necessary is to design a string-matching algorithm from the ground up with an eye towards its efficient execution on modern processors. In this paper, we develop such an algorithm that we call the bitwise algorithm with packed variables and blind reads (BAPVBR). We introduce BAPVBR in Section 2 and describe how it is designed to leverage modern processor hardware for exact DNA sequence string-matching by minimizing complex control flow (in order to decrease branch mispredictions), by decreasing register spills (in order to decrease costly main memory accesses), and by an intelligent skip mechanism (in order to optimize cache accesses).

2. Methods

In this section, we present our fast bitwise exact matching algorithm, which we call BAPVBR. The BAPVBR is a fast string-matching algorithm for short-length DNA sequences.

The basic philosophy of the BAPVBR is assigning multiple tasks to conditional statements and variables in order to efficiently utilize modern CPU branch predictors and memory structures, and in particular cache units. To achieve this, we pack multiple conditions inside bitwise variables and exploit back-to-back XOR and

OR operators to handle fast string searches. Moreover, we use a heuristic skip mechanism to guarantee a simple and plain search loop during pattern searches.

The algorithm belongs loosely to the bitwise hash family and implements packed variables. It splits the packed variables into two segments. While the first segment of the packed variable counts back-to-back character matches, the second segment of the variable handles the comparison of the most recent text character and pattern. Sections 2.1 and 2.2 explain the structure of the packed variables, Section 2.3 details how the BAPVBR operates on such packed variables, and Section 2.4 describes the BAPVBR bitwise operators. Section 2.5 explains our skip mechanism and finally Section 2.6 explains the pseudocode of the complete BAPVBR algorithm.

2.1. The packed variable data structure

The basic data structure in the algorithm utilizes a 32-bit word integer. The bit structure of the word, W, is denoted in Figure 1, where the words are fragmented into two segments. By packing multiple uses in the 32-bit variable through segments, we aim to decrease register spills to the memory and thus decrease memory accesses. In the structure, the leftmost bit is unused, reserved for keeping the sign of a 32-bit integer.

U	27-comparison bits													Unbroken Match counter			
0														0	0	0	0

Figure 1. Bit structure of BAPVBR words, W.

In the first segment in Figure 1, the rightmost four bits are reserved for counter bits. The counter keeps cumulative unbroken matches between the text and pattern. As a result, four bits can handle a maximum of 15 unbroken matches.

As shown in Figure 1, the remaining 27 comparison bits are reserved for pattern matching. Given that enumeration of each DNA character takes 2 bits inside a 32-bit word, we can align a maximum of 13 length patterns inside 27 comparison bits and that determines the maximum pattern length.

2.2. Bitwise representation of pattern inside the packed variable

We compute the bitwise equivalent of the pattern in the 32-bit integer form as shown in Figure 1. Here, four unbroken matching counter bits of the pattern must be 0. On the other hand, DNA characters of the pattern are transformed into their 2-bit enumerations and aligned into pattern comparison bits.

The alignment procedure starts from the right side of the pattern comparison bits. For instance, P_{m-1} is aligned into [4,5] bit positions of the bit word. In general P_j is aligned into the $[2(m+1-j), 2(m+1-j)+1]$th bit indices of the word.

2.3. Utilizing packed variables in BAPVBR

During a pattern search, two conditions have to be handled in each step. These are:

1. Did the last character match?
2. Did the cumulative matches reach the pattern size?

In our technique, a condition controller, *cc*, handles the two conditions jointly. The *cc* is a 32-bit integer and its bit structure is in the same format as in Figure 1. During pattern searches, value *cc* can denote one of the following three different cases:

Case 1 - *cc* is less than m: All of the pattern comparison bits will be 0. On the other hand, some unbroken match counter bits can be 1. Case 1 denotes that there exists a chain of unbroken letter matches. However, all m characters of the pattern have not been tested yet.

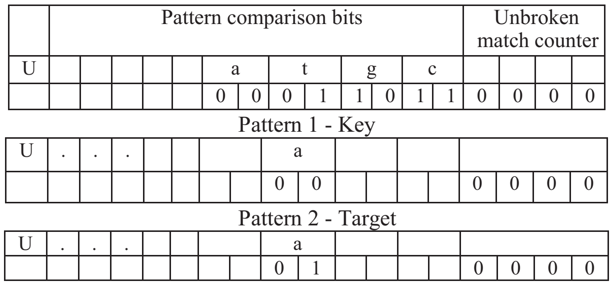
Case 2 - *cc* is equal to m: All of the pattern comparison bits will be 0. On the other hand, unbroken match counter bits denote pattern length m. Case 2 denotes that a chain of m unbroken characters has been tested and matched, indicating that an exact match has been found.

Case 3 - *cc* is larger than m: At least one bit of the comparison bits is not 0. Here we do not need to consider the bit structure of unbroken match counter bits. Case 3 denotes that a mismatch occurred and that the value of the unbroken match counter is useless.

2.4. BAPVBR bitwise operators

In order to find a mismatch efficiently, at least one comparison bit of *cc* should be 1. The bitwise XOR operator ensures this requirement since its output yields 1 when a mismatch occurs. If the text letter does not match the pattern at the relevant index, the integer output of the XOR operator will be at least 16 due to pattern comparison bit alignment at the 32-bit word boundary. In contrast, if the text letter matches, the result of XOR will always be 0.

We incorporate the match/mismatch information into the *cc* variable using the OR operator. The OR operator does not change the unbroken match counter bits of *cc*, since letter comparisons do not take place at unbroken match counter bits. We illustrate the use of the operators through an example shown in Figure 2. Here, a text letter is compared with a pattern. Concretely, the 8th and 9th bits of pattern and text letter words are compared. The bit index implies that P_{m-3} is being compared. The result of XOR operation contains a nonzero bit, which denotes a mismatch. Finally, the mismatch information should be transferred into packed variable *cc* by bitwise OR operator. Recall that the bitwise OR operator between the result and *cc* is able to transfer nonzero bits of the result to *cc*. After the bitwise OR operation, the bit word structure of *cc* maintains packed information, including whether a mismatch occurred or not.



	Pattern comparison bits													Unbroken match counter			
U						a		t		g		c					
						0	0	0	1	1	0	1	1	0	0	0	0

Pattern 1 - Key

U	.	.	.					a									
								0	0					0	0	0	0

Pattern 2 - Target

U	.	.	.					a									
								0	1					0	0	0	0

Comparison Result

Figure 2. Mismatch detection with XOR, where 8th bit of result is nonzero and denotes mismatch. Pattern key is XORed with pattern target.

2.5. The BAPVBR skip mechanism

In this section we present a reliable skip mechanism, which does not skip over any correct match and is optimized for short-length patterns. The goals of presenting a condensed algorithm and pattern occurrence probabilities of DNA are the basic justifications of the skip mechanism introduced here.

During the start of a match phase, the algorithm reads two consequent characters from the text before comparison. Reading multiple characters might seem to be counterintuitive at first. We next explain why we read two characters instead of one (or any other number of characters) during the start.

In terms of probabilistic views of bad character heuristics, there are three cases:

Case 1: If the first compared text letter does not exist in the pattern, general approaches skip m characters from the text. In contrast, our technique reads an extra text letter and then skips m characters. Therefore, the one extra memory access could be an overhead. Such a situation is the worst case in our heuristic technique.

Case 2: If the first read text letter is not a match, but the letter is not a bad character, general approaches present skip distances smaller than m. Our technique is compatible with this situation, since we can determine larger skip distances with two letters. Hence, for this case there is no overhead.

Case 3: If the first letter is a match, our technique outperforms other skip approaches since it requires fewer comparisons.

Lemma: The worst scenario for our approach, Case 1, occurs infrequently.

Proof: By contradiction, we assume that a random text letter take place in the pattern. Since the DNA alphabet has 4 letters, the combinatorial rules [20] denote that such a probability, E(m), for an m-length pattern is:

$$E(m) = 1 - \left(\frac{3}{4}\right)^{n}. \tag{3}$$

Eq. (3) denotes that, if m is 8, occurrence probability of a random text letter inside a pattern is nearly 90%. When m increases, the occurrence probability converges to 100%. Hence, the lemma is true.

In fact, undesired case occurrences are the result of letter diversity depletion, where a pattern does not contain all alphabet letters. Even further, the worst scenario case occurs when three of the four DNA letters do not occur inside the pattern. Therefore, blind reads of BAPVBR cause maximum unnecessary memory access rates.

It is possible to present a blind mechanism that reads more than two letters as a first step. However, extra blind reading has its own risks, as well. If a mismatch occurs at the first letter, extra blind reads are useless and may cause cache misses. On the other hand, redundant blind reads will not produce larger skip distances since the skip distance is limited by the pattern length.

2.6. Pseudocode of BAPVBR

The pseudocode of the full algorithm is presented in Figure 3. The algorithm starts with four input parameters. These are pattern, text, hash equivalent of the pattern *pHash*, and skip array *SkipTable*. The first two parameters are in text format and take place in the memory. The third parameter, *pHash*, is the bitwise equivalent of the pattern where each pattern letter is enumerated by two bits and congruent to W format, as shown in Figure 1. The fourth parameter is an array that keeps the skip distance to shift unfruitful text frames without comparison. Since the skip distance is determined by the two DNA letters, consequently the array size is 16.

During a search on a text frame, the rightmost two letters are compared simultaneously. In line 3, we define a new variable, *hashOfTwoPatternLetters*, that keeps the rightmost two pattern letters in W format. In line 4, *hashOfTwoPatternLetters* is shifted since the rightmost four bits of W are reserved for unbroken match counter bits. In line 5, the variable is incremented by 2 in anticipation of possible matches of the letters before the conditional loop starts. Such preprocess steps are aimed at reducing the overall computations before the search loop starts.

```
1. BAPVBRSearch(P,T,pHash, SkipTable)
2. {
3. hashOfTwoPLetters = (enum[$p_{[m-2]}$]<<2) | enum [$p_{[m-1]}$];
4. hashOfTwoPLetters = hashOfTwoPLetters << 4;
5. hashOfTwoPLetters += 2;
6. curPos=m-1
7. while (curPos < n)
8.     {
9.       hashOfTwoTextLetters = (enum[$t_{[curPos-1]}$] << 2) | enum [$t_{[curPos]}$];
10.      cc = (hashOfTwoPLetters ^( hashOfTwoTextLetters << 4)) ;
11.      while (cc< m)
12.          {
13.          cc |= (((enum [$t_{[curPos-cc]}$] << (4 + 2 * cc)) ^ pHash) & (3<<(4+2*cc)));
14.          cc++;
15.          }
16.       if (cc == m)   output → ( curPos-m+1);
17.       curPos += SkipTable[hashOfTwoTextLetters];
18.       }
19. }
```

Figure 3. BAPVBR algorithm.

Line 6 defines the *curPos* variable, which handles the current text index. Its first value is m − 1, since the initial text frame index is [0, $m-1$]. On the other hand, line 7 denotes that the program should iterate until the last letter of DNA text. In line 9, two consequent characters are read from the text frame and their footprints are aligned into four bits. The bits are moved to pattern comparison bit indices with shift operators. In line 10, the rightmost two characters of the pattern and text frame are compared with the XOR operator. If both two letters match, the value of *cc* becomes 2; otherwise, the value of *cc* becomes large enough to denote a mismatch.

In order to encounter an exact match, the value of *cc* should be m, which denotes an unbroken m letter match in the text frame. Line 11 is a condition that signifies "repeat until m characters are matched or a mismatch occurs". The loop continues, unless the value of the unbroken match counter becomes m or a mismatch occurs. In line 13, the next character is taken from the text and compared with the pattern. The XOR operator determines any mismatch and traces its footprint into *cc*. In line 14, *cc* is incremented by one, since a new character has been compared.

The output of the algorithm is generated only in line 16. The program can arrive at line 16 through two different cases. Either a mismatch occurred or consequent m characters have matched. If *cc* is equal to m, then its pattern comparison should still be 0, and consequently an exact match has occurred. Otherwise, some of the pattern comparison bits have been set to 1, which signifies that a mismatch has occurred.

In line 17, the text index should be skipped to a reliable distance. In order to determine such a distance, the *hashOfTwoTextLetters* variable is utilized again. Recall that the *hashOfTwoTextLetters* variable had another mission in line 9, as well. Therefore, we ensure that each variable of the algorithm has packed information to handle.

3. Results and discussion

3.1. Test data and analysis technique

In order to compare the algorithms, we use a common test bed. The test bed is composed of DNA sequences of *Mus musculus* genes. The original test bed is from PubMed in FASTA format and composed of 23 files and nearly 2.3 billion nucleotides. Before experimentation, we eliminated nonnucleotide characters from FASTA files. The experiments assume that DNA is loaded into the main memory.

We exploit eight different patterns with various lengths. Patterns and their specifications are denoted in Table 1. Patterns, which are identified as *P1*, *P2*, and *P3*, are exploited to evaluate the effect of letter diversity depletion. *P1* is the theoretical worst possible case of BAPVBR since all letters of the pattern are same. The remaining patterns are used to evaluate the effect of pattern length.

In order to minimize jitter such as operating system noise in the results, we repeat each experiment five times; discard the outliers, i.e. the best and worst results; and report the average of the remaining three results as the final.

Table 1. Patterns and their specifications.

Pattern id (Pid)	Pattern	m	Letter diversity
P1	AAAAAAAAAA	10	1
P2	TCTCCCTCTTTTT	13	2
P3	CGCGCGCCA	9	3
P4	AAGCTTGG	8	4
P5	TGCCAGGGAG	10	4
P6	TGTATATGTCA	11	4
P7	GATGCCTGTAGT	12	4
P8	GAGGGTAGCTGAT	13	4

Our experiments are based on elapsed time and hardware counters. In order to determine the elapsed time, we exploit the stopwatch in .NET, which reports final time in milliseconds. In terms of hardware counters, we utilize the performance counters for the Intel X86 architecture. Briefly, we consider the branch and cache counters. In terms of branching, we consider branch instruction executions and mispredicted branches. To analyze the cache, all references to L1 data cache and L2 data requests counters are observed. Finally, the last level cache misses counter hints on the performance of memory.

The tests are implemented with Intel Core i3 CPU 550 @3.20 GHz with 4 GB RAM and the 64-bit Windows 7 operating system. All codebase is written in C# and hardware counters are obtained from the Intel Vtune Performance analyzer.

Our effective performance criterion is the speed of the algorithm, based on elapsed time. The algorithms are compared in each pattern style, where P1 is the theoretical worst case for the BAPVBR.

In order to understand the factors of elapsed time, we analyze CPU counters and implement detailed CPU counter analysis to observe the effect of letter depletion and short and longer patterns.

Implementations of Shift OR [7], BNDM [8], and BLIM [9] algorithms abide by their original presentations. On the other hand, Bitwise Horspool is a modification of [10] where each DNA letter is represented by two bits. In general, algorithms are implemented in optimized forms. For code simplicity, few variables are defined as global in each experiment, where contributions of the global variables to speed are less than 3%.

3.2. Comparison of the algorithms based on elapsed time

Our results indicate that the cost of the preprocess step is negligible when data are GB in size. In our test bed, the preprocess cost is 5×10^{-4} times less than string-matching time.

Based on the stopwatch, performance comparisons of algorithms can be seen in Table 2. For clarity, the results are transformed into milliseconds. Results denote that BAPVBR outperforms in all patterns. The improvement becomes substantial when all alphabet letters exist in the pattern.

Table 2. Execution time of algorithms in milliseconds.

Pid	Bitwise algorithm					
	BLIM	Shift OR	Bitwise Hash	Bitwise Horspool	BNDM	BAPVBR
P1	121,210	68,090	63,337	29,850	21,349	18,397
P2	133,533	66,342	61,201	26,407	22,463	14,277
P3	150,280	68,361	61,455	30,880	28,336	15,597
P4	125,896	66,450	62,122	50,787	39,242	20,483
P5	149,181	66,832	62,943	39,786	29,697	17,175
P6	200,281	67,102	61,457	66,240	29,724	15,118
P7	195,644	66,784	63,548	61,844	26,767	14,805
P8	217,325	66,108	61,154	75,123	25,079	13,954

During experimentation, Bitwise Hash and Shift OR present constant and moderate execution times, regardless of the pattern property. This is not a surprise since both algorithms lack the skip mechanism.

Given that the pattern contains at least three different letters, BAPVBR presents 85% improvement over the previous best-performing algorithm, BNDM. We think that such an improvement rate provides an interesting first step in architecture-sensitive design of DNA search algorithms.

During execution of *P2*, TCTCCCTCTTTTT, BAPVBR outperforms BNDM by 57%. Since *P2* contains two different letters, unnecessary blind read occurrences are less frequent than in *P1*. However, the performance of BAPVBR surpasses that of BNDM with the help of its simple and hardware-level bitwise operations.

In the worst case for BAPVBR, all pattern letters should be same. In terms of *P1*, which is AAAAAAAAAA, BAPVBR outperforms BNDM by only 16%. The reduction from 85% to 16% gain is not a surprise. The basic factor is the heuristic skip mechanism, which runs faster on patterns that have large letter diversity. During the first comparison step, if 'T', 'G', or 'C' occurs, the BAPVBR unnecessarily reads a second text character blindly before launching the skip mechanism. Hence, these unnecessary memory accesses decrease the advantage of BAPVBR over BNDM.

In summary, performance test results strongly suggest that the BAPVBR outperforms for short-length DNA sequences.

3.3. Hardware counter perspective

Results of the previous section showed that simplification of the code (in order to minimize branch mispredictions while maximizing cache hit rate) and keeping condensed information inside variables (in order to minimize register spills) had strong contributions to the speed of searches. In this section, we introduce a detailed performance analysis based on hardware counters. The goal of this section is to present the factors of execution time on modern hardware. For better analysis, we present both positive and negative CPU counter aspects of the algorithms. We analyze *P1* since it is the worst case of BAPVBR and contains only one alphabet letter. On the other hand, *P4* and *P8* evaluate the contribution of minimum/maximum pattern lengths over CPU counters.

We divide the CPU counters into two fragments. At first we consider branch executions rates. In the second phase, we consider cache interactions.

3.3.1. Branch execution and prediction

Current CPU architectures are pipelined and exploit branch prediction to enable high speed computation. The branch predictor, which is a digital circuit composed of a large lookup table, is typically found at the decode pipeline stage of modern processors. The predictor estimates the branch outcome (whether it is taken or not taken) and the branch target address (in the case of indirect branches) of a branch instruction and speculatively pushes it into the processor pipeline. The branch is resolved later in the pipeline during the execute stage. The resolved branch outcome is compared with the predictor result. If both match, then the branch instruction is allowed to commit [21]. Hence, successful prediction of branches enhances the flow in the instruction pipeline of the CPU. In case of a misprediction, the instructions in the pipeline will be flushed and the CPU stalls until instructions from the correct path have been fetched into the pipeline. Thus, efficient branch prediction is crucial for high performance.

In this step, we analyze performance counters of *P1*, *P4*, and *P8*. Recall that *P1* represents letter depletions. *P4* is the shortest and *P8* is the longest pattern.

We present the results about branch instructions executed in Table 3. Results show that BAPVBR and BNDM require minimum branch executions. In contrast, the BLIM algorithm presents the worst branch execution performance. Although Shift OR and Bitwise Hash have the simplest match procedures, they have a large number of branches.

Table 3. Total number of branch instructions.

	P1	*P4*	*P8*
BAPVBR	3,013,600,000	4,067,733,333	2,978,933,333
BNDM	2,870,933,333	5,298,666,666	3,302,933,333
Bitwise Horspool	7,927,733,333	14,742,933,333	22,262,133,333
Bitwise Hash	9,765,066,666	9,837,600,000	9,833,333,333
Shift OR	9,930,400,000	9,842,133,333	9,852,266,666
BLIM	22,913,600,000	23,833,066,666	40,907,200,000

BAPVBR, BNDM, and Bitwise Horspool results show that the highest branch executions occur in *P4*. Recall that *P4* is the shortest pattern. Therefore, it introduces smaller shifts during a mismatch. In contrast, pattern length does not affect Bitwise Hash and Shift OR since they do not have a skip mechanism.

We present the mispredicted branch executions in Table 4. Test results show that mispredicted branch rates of algorithms are very small when compared to the total number of branch instructions executed. For instance BAPVBR's misprediction rate is around 1%. We observe that BNDM and Bitwise Horspool present six-fold more misprediction. The results help to explain the performance differences between BAPVBR and BNDM and Bitwise Horspool.

Table 4. Total number of mispredicted branch instructions.

	P1	*P4*	*P8*
BAPVBR	67,066,666	51,920,000	38,880,000
BNDM	136,533,333	335,440,000	207,786,666
Bitwise Horspool	210,320,000	312,186,666	520,266,666
Bitwise Hash	586	266	240
Shift OR	426	266	373
BLIM	2,053,333	693,653	1,546,666

Test results show that Shift OR and Bitwise Hash do not cause considerable amounts of misprediction. We think that such a result is not a surprise, since the algorithms do not require a condition to skip characters.

The observation denotes that adding complex control-flow instructions to the string-match algorithms could sometimes be detrimental.

In terms of overall branching, BAPVBR presents a well-balanced performance. While Bitwise Hash, Shift OR, and BLIM present approximately four million less mispredictions, they cause an extra 4.8 billion branch executions.

3.3.2. Cache accesses

Another performance criterion of search algorithms is based on cache efficiency. Caches are used to reduce average memory access time during execution and are a critical structure to scale the so-called memory wall, which refers to the limited performance gains for the main memory technology compared to the processor core performance with increased transistor scaling. Modern CPU architectures can typically contain at least three level caches. During execution, the CPU first accesses its cache in order to find the required instruction or data. In the case of a cache miss the instruction or data should be fetched from higher-level memory, which may cause CPU stalls.

In Table 5, we present the "All references to L1 data cache" performance of the algorithms. L1 data cache results validate the results of the stopwatch. The results denote that BAPVBR requires the minimum number of data from the cache.

Table 5. Number of all references to *L1* data cache.

	P1	*P4*	*P8*
BAPVBR	9,784,000,000	12,544,000,000	9,205,333,333
BNDM	10,394,666,666	20,741,333,333	13,189,333,333
Bitwise Horspool	14,456,000,000	26,968,000,000	41,064,000,000
HASH	36,602,666,666	36,864,000,000	36,850,666,666
Shift OR	39,760,000,000	39,317,333,333	39,421,333,333
BLIM	57,688,000,000	58,245,333,333	103,949,333,333

The results are very important since they imply that the BAPVBR minimizes register spills. Register spills occur when the compiler cannot find a free register to assign to a frequently used data item, and this is especially the case for architectures with a limited number of architectural registers such as the X86 [22]. We think that intensely used variables reduce the risk of register spills, especially for architectures that have a limited number of registers.

3.3.3. Memory access

Finally, we present our findings about the last level cache (LLC) miss. When the data are not found in the LLC, they should be fetched from RAM and current high-performance out-of-order processors do not tolerate LLC misses well. The reason for that is their pipeline structures, which extract available instruction-level parallelism such as load/store queues, reorder buffers, issue queues, and physical register files that get filled up in the shadow of a long-latency LLC miss (which can take hundreds of cycles), leading to pipeline stalls.

The results in Table 6 denote that Shift OR and Bitwise Horspool outperform during LLC miss analysis, where both algorithms have the simplest loop structures. The results present LLC misses triggered by loads only, and they do not include misses triggered by either software or hardware prefetches. Such an observation implies that new algorithms must consider prefetching capability of modern hardware.

While the BAPVBR is moderate on LLC misses, other performance counters compensate such a drawback. When all hardware counters are considered, the BAPVBR has the overall best performance. It presents satisfactory branch execution and misprediction rates. In terms of *L1* cache interactions, it denotes the best performance as well. Finally, the hardware counter results are confirmed by elapsed time.

Table 6. Total number of last level cache misses.

	P1	*P4*	*P8*
BAPVBR	1,413,333	1,626,666	1,480,000
BNDM	1,680,000	2,933,333	2,626,666
Bitwise Horspool	1,493,333	,880,320	1,080,000
Bitwise Hash	1,186,666	1,160,000	1,200,000
Shift OR	1,186,666	866,973	1,186,666
BLIM	1,293,333	1,506,666	1,253,333

4. Conclusion

This study presents a new bitwise exact string-matching algorithm, which aims at fast DNA pattern searches on large DNA sequence files. Due to computer hardware advances, large DNA files can be stored in memory and sequence comparisons can be sped up by modern algorithmic techniques. For instance, bus communication could be minimized and computational procedures could be generally executed inside the CPU and cache. Interactions inside the CPU and cache also affect the performance. For instance, bitwise string-matching exploits intrinsic parallelism properties. In general, bitwise algorithms introduce satisfactory performance under the condition that the pattern length is small.

In this study, we numerically establish that CPU pipelining and cache miss rates have substantial contributions on the DNA search performance. Hence, we consider hardware counters of the CPU to enable fast pipeline computation. In order to obtain the best performance from the CPU, we optimize the match code to minimize branch mispredictions and cache misses. In our approach, algorithmic components have multiple missions. In addition, we compress conditions and reduce possible condition lines. Test results show that such actions help successful branch prediction and reduce cache access. During pattern searches, we tested a 32-bit integer as the bitwise condition. The algorithm permits maximum 13-length DNA patterns. In fact, such a length reduces the search space for 2 GB of data drastically. For terabyte-sized text data, the algorithm can be modified to a 64-bit word. As a consequence, maximum pattern length limit can be increased.

Test results show that our algorithm outperforms in short-length DNA patterns, even for the worst possible pattern, where all pattern letters are same. Results imply that searches on shortest pattern sequences take more time. Basically, lengths of patterns limit larger skip distances. On the other hand, there is no limitation on text length. Therefore, the algorithm can be executed on large DNA databases and jointly executed with heuristic algorithms as a tool to find initial pattern seeds.

The results denote that hardware counter analysis should become a requirement for pattern-matching algorithm design and optimization. Therefore, we can help the CPU for better branch prediction and *L1*, *L2* cache data hits. We also observe that modern prefetch technologies help to reduce cache misses. Hence, exact matching algorithms must take into account recent hardware technologies such as prefetching.

Acknowledgment

The authors are grateful to Professor Azmi Yerlikaya for suggesting the *Mus musculus* DNA sequence and for fruitful discussions.

References

[1] Bucak İÖ, Uslan V. Sequence alignment from the perspective of stochastic optimization: a survey. Turk J Electr Eng Co 2011; 19: 157–173.

[2] Pehlivan İ, Orhan Z. Automatic knowledge extraction for filling in biography forms from Turkish texts. Turk J Electr Eng Co 2011; 19: 57–71.

[3] Liu Y,Wu X, Hu X, Gao J, Wang C. Pattern matching with wildcards based on multiple suffix trees. In: IEEE International Conference on Granular Computing; 11–13 August 2012; Hangzhou, China. New York, NY, USA: IEEE. pp. 320–325.

[4] Moraru I, Andersen DG. Exact pattern matching with feed-forward Bloom filters. ACM Journal of Experimental Algorithmics 2011; 16: 2.1:1–2.1:19.

[5] Sheik SS, Aggarwal SK, Poddar A, Balakrishnan N, Sekar K. A fast pattern matching algorithm. J Chem Inf Comput Sci 2004; 44: 1251–1256.

[6] Franek F, Jennings CG, Smyth WF. A simple fast hybrid pattern-matching algorithm. Journal of Discrete Algorithms 2007; 5: 682–695.

[7] Baeza-Yates R, Gonnet GH. A new approach to text searching. Commun ACM 1992; 35: 74–82.

[8] Navarro G, Raffinot M. Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithmics 2000; 5: 4.

[9] Külekci O. A method to overcome computer word size limitation in bit-parallel pattern matching. Lect Notes Comput Sc 2008; 5369: 496–506.

[10] Horspool RN. Practical fast searching in strings. Software Pract Exper 1980; 10: 501–506.

[11] Karp RM, Rabin MO. Efficient randomized pattern-matching algorithms. IBM J Res Dev 1987; 31: 249–260.

[12] Sunday DM. A very fast substring search algorithm. Commun ACM 1990; 33: 132–142.

[13] Cantone D, Faro S, Giaquinta E. Bit-(parallelism): getting to the next level of parallelism. Lect Notes Comput Sc 2010; 6099: 166–177.

[14] Durian B, Holub J, Peltola H, Tarhio J. Tuning BNDM with q-grams. In: Proceedings of the Workshop on Algorithm Engineering and Experiments; 2009; New York, NY, USA. New York, NY, USA: SIAM, pp. 29–37.

[15] Fredriksson K, Grabowski S. Practical and optimal string matching. Lect Notes Comput Sc 2005; 3772: 376–387.

[16] Zhang G, Zhu E, Mao L, Yin M. A bit-parallel exact string matching algorithm for small alphabet. Lect Notes Comput Sc 2009; 5598: 336–345.

[17] Külekci MO, Vitter JS, Xu B. Boosting pattern matching performance via k-bit filtering. Lect Notes Electr En 2010; 62: 27–33.

[18] Külekci O. On enumeration of DNA sequences. In: Proceedings of the ACM Conference on Bioinformatics, Computational Biology, and Biomedicine; 2012; Orlando, FL, USA. New York, NY, USA: ACM. pp. 442–449.

[19] Boyer RS, Moore JS. A fast string searching algorithm. Commun ACM 1977; 20: 762–772.

[20] Rosen KH. Discrete Mathematics and Its Applications. 6th ed. New York, NY, USA: McGraw-Hill, 2007.

[21] Hennessy JH, Patterson DA. Computer Architecture: A Quantitative Approach. 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2006.

[22] Appel AW, George L. Optimal spilling for CISC machines with few registers. ACM SIGPLAN Notices 2000; 36: 243–253.