Uludağ University Journal of The Faculty of Engineering, Vol. 21, No. 2, 2016 RESEARCH

DOI: 10.17482/uumfd.273970

# DETECTION OF P53 CONSENSUS SEQUENCE: A NOVEL STRING MATCHING WITH CLASSES ALGORITHM

*Gıyasettin ÖZCAN* *

Received: 05.04.2016; revised:14.09.2016; accepted:04.11.2016

**Abstract:** We present a novel fast string matching technique for special DNA pattern forms and compare performance of recent CPU architectures on the matching problem. In particular, we consider consensus P53 DNA-binding consensus sequence, which has an important contribution for cancer treatment. Based on biological findings, consensus P53 pattern may emerge in various sequence forms and its length is not deterministic. Therefore, classic string matching algorithms are not able to solve the problem. For efficient solution, we consider bitwise string matching algorithms with classes and present a novel search technique which is based on 64-bit packed variables. In order to prevent obstacles based on variable length of the pattern, we search right and left side indexes of P53 and reduce search space. For experimental analysis, we make use of mus musculus DNA sequences with approximately 2.3 billion nucleotides. We compare algorithm performance on three processors with distinct CPU architecture. Test results show that our search technique introduces at least 20% efficiency during P53 pattern search in each architecture platform. Due to its structure, the algorithm also introduces an efficient solution to similar string matching with class problems.

**Keywords:** Computer Engineering, Computational Biology, Text Processing, Consensus Sequences, Bitwise Matching, Hardware Counters

**P53 Konsensüs Sekansının Yakalanması: Sınıf Özellikli Yeni Bir Sekans Eşleştirme Algoritması**

**Öz:** Bu çalışmada özel DNA örüntüleri için yeni ve hızlı bir sekans eşleştirme tekniği sunulmakta ve yakın geçmişte üretilen CPU mimarileri üzerinde deneysel karşılaştırmalar yapılmaktadır. Bu çalışmada, bilhassa kanser tedavisinde, önemli bir yere sahip olan P53 DNA-bağlayan konsensüs sekansı göz önüne alınmıştır. Biyolojik kazanımlara göre P53 örüntüsü farklı sekans formlarında karşımıza çıkabilmekte ve sekans uzunluğu değişebilmektedir. Bu nedenle P53 sekansının klasik sekans eşleştirme algoritmaları ile çözümü mümkün olamamaktadır. Bu çalışmada verimli çözüm yöntemi sunmak amacıyla, sınıf özellikli bit-tabanlı sekans eşleştirme algoritması göz önüne alınmıştır. Hedef doğrultusunda, 64-bit paketlenmiş değişken kullanarak yeni bir arama ve eşleştirme algoritması önerilmiştir. Örüntü sekansının değişken uzunluk gösterebilmesi nedeniyle karşılaşılması muhtemel engelleri aşmak için ise veri tabanında P53 sekansının özel kısımlarına dair aramalar yapılmıştır. Deneysel analiz için yaklaşık 2.3 milyar nükleotidden oluşan mus musculus DNA sekansı seçilmiştir. Karşılaştırılan algoritmalar üç farklı bilgisayar mimarisinde test edilmiştir. Deneysel sonuçlar, geliştirdiğimiz algoritmanın P53 sekans arama konusunda tüm mimari platformlarında en iyi verimliliği sağladığını göstermektedir. Yapısı gereği bu algoritma, benzer sekans eşleştirme problemlerinin çözümünde de verimli olanaklar sunmaktadır.

**Anahtar Kelimeler:** Bilgisayar Mühendisliği, Hesaplamalı Biyoloji, Metin İşleme, Konsensüs Dizileri, Bitsel Eşleme, Donanım Sayaçları.

* Faculty of Engineering, Computer Engineering Department, Uludag University, 16059, Bursa, Turkey
Corresponding Author: Gıyasettin Özcan (gozcan@uludag.edu.tr)

## 1. INTRODUCTION

Computational biology is an interdisciplinary research field, which provides comprehensive genome analysis techniques. The field is attractive since recent biological assays have generated vast amount of DNA sequences. In order to extract significant information from large-size biological sequences, it is necessary to develop efficient computational analysis techniques and compatible hardware resources.

In the field of biology, finding similar patterns inside sequence databases is highly important since biologists need to understand the correlation between DNA sequences and protein function. The string matching studies present efficient similarity search techniques to extract such correlation. Concretely, string matching algorithms search query patterns inside large string databases and output the occurrence indexes.

In terms of cancer research, *P53* is a critical protein pattern for multi cellular organisms which regulates cell cycle (Kern and others, 1992), (El-Deiry, 1998). Concretely, it is a tumor suppressor and a vital protein to prevent cancer disease. On the other hand, DNA structure of P53 is complex and computational advancements may help understanding its functions.

DNA-binding consensus sequence of P53 is explained in El-Deiry (1998) and summarized in Table 1. In the table, the consensus sequence is divided into three segments, where first and third segments of P53 have the same sequence characteristics and are composed of 10-length nucleotides. On the other hand, length of second segment is not constant and it may contain minimum 0 and maximum 13 nucleotides. There is no constraint about nucleotide selection in the second segment.

The P53 sequence starts with 5' symbol, which imply that the regulator sequence is starting. First three nucleotides of P53 is a member of Purine, *Pu*, group where both adenine, *A*, and guanine, *G*, are accepted as match. Fourth index must be cytosine, *C*. Fifth and sixth indices can be either *A* or thymine, *T*. While seventh index must be *G*, next three indices must be a member of pyrimidine group, *Py*, where both *C* and *T* are accepted as match. At last, the 3' symbol denotes the end of regulator sequence.

**Table 1. Three Segments of Consensus P53 DNA Sequences**

| Segment | Sequence |
|---|---|
| **1** | 5'-*PuPuPuC(A/T)(A/T)GPyPyPy* |
| **2** | *N*(0-13) |
| **3** | *PuPuPuC(A/T)(A/T)GPyPyPy*-3' |

Since some of the indexes may accept more than one letters, the P53 consensus sequence may occur in various combinations. In this study, we define P53 or other patterns with the same characteristics as *flexible patterns*. Due to its importance, novel string search on flexible patterns including P53 pattern, could contribute to the literature.

## 2. DEFINITONS AND LITERATURE

String matching algorithms may aim to search two types of patterns during search. These are conventional patterns and flexible patterns. In this section we explain both of them.

### 2.1. Conventional Exact String Matching

Exact string matching algorithms are used when speed and memory efficiency are important. A survey on exact string matching can be found in (Faro and Lecroq, 2010). During conventional exact string matching, a pattern, *P*, is searched inside text, *T*. Formally, let text is defined as

$$T = t_0, t_1, ..., t_{n-1} \tag{1}$$

and pattern is represented as:

$$P = p_0, p_1, ..., p_{m-1} \tag{2}$$

An exact match occurs if sequence alignment of text fragment and pattern is completely same. Formally,

$$p_0 = t_j, p_1 = t_{j+1}, ..., p_{m-1} = t_{j+m+1} \tag{3}$$

In terms of exact match, a pattern index accepts one alphabet letter. Formally

$$|p_i| = 1 \tag{4}$$

During exact string matching, algorithms present efficient skip mechanisms that predict mis-match cases before actual comparison. Consequently, they try to minimize overall sequence comparisons, CPU and memory usage. Some of the well-known algorithms that presented the fundamental string matching techniques are (Knuth and others, 1977), (Boyer and Moore, 1977), (Horspool, 1980), (Sunday, 1990). These algorithms presented base characteristics of efficient skipping during search. On the other hand, Karp and Rabin (1987), Kim (1999), Fuyao and Qingwei (2009) introduced hash based solutions to the problem.

Recent exact matching algorithms inherit the advancements of early techniques and establish further improvements. For instance, Durian and others (2009) make use of q-grams. On the other hand, automata based algorithms denotes a different aspect of exact matching and present a automata based solution that utilizes directed acyclic word graph (Fan and others, 2009).

Bitwise string matching is another form of exact matching, where pattern and texts are represented in bits. In this form, string matching problem can be executed by utilizing intrinsic parallelism of bitwise operators. In addition, low level operations speeds up the string matching execution time. In terms of bitwise match, (Baeza-Yates and Gonnet, 1992), (Navarro and Raffinot, 2000), (Kulekci, 2008), (Kulekci, 2012) introduced novel algorithms.

A general survey paper, written by Faro and Lecroq (2010) denotes that performance of the algorithms depend on the dataset. On the other hand bitwise algorithms are convenient during DNA based string matching, since alphabet of DNA consist of only four letters (Ozcan and Unsal, 2015).

Exact string matching algorithms have common pitfall during P53 consensus sequence search on sequence databases. In fact P 53 sequence can be observed in various forms as shown in Table 1. For instance first index of pattern is *Pu*, which implies that first index can be either *A* or *G*. Such pattern contradicts with equation 4, which expects single pattern letter at each index.

### 2.2. Flexible Patterns and String Matching with Classes Problem

In contrast to conventional patterns, we define a flexible pattern in the following formal form,

$$1 \leq |p_0| \leq |\Sigma| \tag{5}$$

Equation 5 is more convenient to P 53 pattern. As an example, first index of P53 accepts both A and G letters and $|p_1|=2$. It can be easily seen from the Equation 5 that flexible patterns are super set of conventional patterns.

In order to present solutions to string matching with class problem, conventional exact string matching need to handle extra conditions. As an example, if the alphabet size is small, it can be modified for flexible patterns by bitwise representation, where occurrence of each alphabet letter can be represented with a single bit.

In literature, some of the bitwise string matching algorithms are adapted for flexible patterns as bitwise string matching with classes problems (Baeza-Yates and Gonnet, 1992), (Navarro and Rafinot, 2000), (Kulekci, 2008). These bitwise algorithms are based on bit masks and require simple modification to handle string matching with classes.

There exist further advancement possibilities for bitwise string matching since power of bitwise string matching is based on computer architecture and hardware systems are still improving.

Recent advancements on computer architectures introduced new hardware and software techniques. For instance, parallel architectures reduce the workload of single CPU. On the other hand, novel branch predictors estimate future instruction streams of CPU correctly and introduce more efficient pipelining architectures (Vintan, 1999). Furthermore, novel memory management mechanisms try to keep frequently used variables inside CPU registers (Appel and George, 2000) in a process called register promotion.

New architectures also enable access to CPU hardware counters. As a result algorithms can be analyzed more comprehensively. For instance users can access total CPU cycles during execution of a program. In addition total branch executions/mis-predictions can be counted. Similarly, we can obtain counter information from memory structures such as caches. Recently, software such as PAPI enable convenient access to CPU counter information.

To the best of our knowledge, bitwise string matching studies do not have comprehensive CPU counter analysis. Henceforth, detailed architectural analyses during program executions has not been conducted before.

In fact, such CPU counter experimentations may help hardware architects. For instance, the analysis introduce the cost of CPU cycles, cache misses during string matching. As a result of the analysis, computer architects can consider the contribution of hardware costs of string matching problems and optimize for string matching problem.

## 3. METHOD

Recall that first and third segments of P53 are completely same and their length is 10, whereas length of second segment can be at most 13. The new algorithm searches the index positions of first segment inside text. If the distance between consecutively founded indexes is less than 23, we assume complete P53 pattern has been found.

The new algorithm aims at DNA sequences, where alphabet has four letters. We represent each pattern index with four bits, where each bit implies the existence/non-existence of a letter/nucleotide in pattern. For instance, we represent *C,G,T,A* as 8,4,2,1 respectively and denote in Table II in bitwise form.

**Table 2. Bit Transformations of DNA Letters**

| DNA letter | Bitwise representation |
|---|---|
| C | 1000 |
| G | 0100 |
| T | 0010 |
| A | 0001 |

As a consequence of the bit representation in Table II, *Pu* becomes 0101 since *A* and *G* are accepted members of *Pu*. On the other hand, *Py* subsumes *C*,*T* and it is represented by 1010. Finally, (*A*/*T*) is denoted as 0011. Depending on these rules, bitwise representation of P53 pattern segment 1 and segment 3 can be denoted as:

0101 0101 0101 1000 0011 0011 0100 1010 1010 1010.

### 3.1. Bit Structure of the Algorithm

Structure of our algorithm is based on 64-bit words and denoted in Figure 1. We define 64-bit word as *W* and segment into three parts. We define rightmost 4-bits as unbroken match counter bits, *UBMC*, where the bits keep unbroken match counter. In the middle, 59 bits are used for comparison of pattern and text. Finally, leftmost bit is not used since it is the signature bit.

The *UBMC* keeps cumulative matches between pattern and candidate text fragment. During comparison of an index, if pattern and text match at that index, the counter is incremented by one. When the value of *UBMC* becomes *m*, algorithm understands that consecutive *m* comparison pattern indexes have matched. In contrast, if a pattern and text mismatches at an index, value of *UBMC* will be reset to 0. Hence we trace the mismatch inside bit word.

| U | 59 comparison bits | | | | | 5 Unbroken Match Counter(*UBMC*) bits | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | ……. | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

***Figure 1:***
*Bit structure of 64-bit word, W*

The comparison bits are used to compare pattern and candidate text frame, *F*. There are 59-bits reserved for comparison. Since each DNA letter representation requires four bits, we can represent 14-letter patterns inside a 64-bit word.

At first sight 14-bit pattern can be seen too short. However, 14-bit words reduce search space decently. In terms of DNA, 14-letter pattern reduces the search space by $4^{14}$ times, which efficiently satisfy gigabyte-size sequences.

During search technique we assume *P* and *F* are bit words in the *W* format. In order to explain our algorithm, we define *caseWord*. During the comparison of pattern and text letter, trace of the match/mismatch is projected to *caseWord*. Its default value is equal to *P* and its UBMC bits are initially 0. After each letter match between *P* and *F*, value of *caseWord* is incremented by one. When a letter mismatches, *caseWord* returns to its default value.

### 3.2. Illustrative Example

In Figure 2, we consider comparison of $F_{m\text{-}5}$, and $P_{m\text{-}5}$. The situation implies that rightmost four letters of the $F$ must have already matched and *caseWord* keeps necessary information about the condition. Recall that P53 segment demands either *A* or *T* letters at the corresponding index whose bitwise equivalent is 0011.

In order to show a mismatch, let assume that letter from text frame is *G*, and its bitwise equivalent is 0100. The mismatch can be detected by bitwise OR operation between *caseWord* and $F_{m\text{-}5}$ since new value of *caseWord* after bitwise OR becomes larger than *P*. The accrual is large enough to detect a mismatch and is denoted in the equation.

### 3.3. String Matching Cases of Algorithm

During string matching, mathematical difference between *caseWord* and *P* determines all cases of the algorithm. Three cases of the results and corresponding decisions are as follows:

$$Case1: If\ (caseWord - P)\ > m \rightarrow Signal\ mismatch$$

$$Case2: If\ (caseWord - P) == m \rightarrow Signal\ pattern\ match$$

$$Case3:\ If\ (caseWord - P)\ < m \rightarrow Tested\ letters\ matched,\ continue\ to\ compare\ text\ frame$$

If Case 1 or Case 2 occurs, string matching procedure should re-start from a new text frame. Consequently, value of the *caseWord* must be defaulted to *P* and its UBMC bits become 0, which denote characters matched for the new frame yet.

As Figure 2 denotes, *caseWord* variable is used for both comparison and counting. Concretely, the variable is not only called during comparison of text letter, but also used to track overall matched indexes of candidate text frame. Because of this fact, we name *caseWord* variable as a packed variable, which keeps multiple information and handles every step of the algorithm. We expect that *caseWord* variable called very frequently; so that it will usually get promoted to a CPU register and thus speeds up the search time.

### 3.4. Skip Mechanism

When a mismatch occurs, the search should be started from an untried text frame. However, it is possible to predict that some respective indices that will eventually mismatch. Hence computation should rather skip such indexes. Here we implemented a special skip technique that is convenient for P53 segment.

For efficiency, skip mechanism must compensate at least the overhead of skip distance computation. The average skip distance depends on the possible subsequence combinations. For instance, DNA alphabet has four elements and yield 4s combinations with s-length subsequence.

For conventional patterns, Ozcan and Unsal (2015) fetch two text letters successively from memory and decide a skip distance when a mismatch occurs. However reading only two letters for p53 does not ensure enough skip. Such decrement is the result of accepting class of characters for an index. However, it is possible to determine skip larger distances by considering three letters at first step.

We can enable larger skip distance by considering four or more letters at first step. However, such action may cause unnecessary memory reads, cause register spills and cache misses.

| U | *CaseWord* | | | | | | | | | | | | | | *UBMC* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | . | . | . | . | . | . | . | 0 | 0 | 1 | 1 | . | . | . | 0 | 1 | 0 | 0 |

OR

| U | $F_{m-5}$ | | | | | | | | | | | | | | *UBMC* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | . | . | . | . | . | . | . | 0 | 1 | 0 | 0 | . | . | . | 0 | 0 | 0 | 0 |

=

| U | | | | | | | | | | | | | | | *UBMC* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | . | . | . | . | . | . | . | 0 | 1 | 1 | 1 | . | . | . | 0 | 1 | 0 | 0 |

***Figure 2:***
*A mismatch example with OR operation*

## 4. RESULTS

We compare the performance of our algorithm, BAP4Bit, against BNDM-Class (Navarro and Raffinot, 2000), ShiftOR-Class (Baeza-Yates and Gonnet, 1992), Hash-Class and BLIM-Class (Kulekci, 2008) algorithms.

In order to evaluate algorithm performance, complete mus musculus DNA sequences are collected from Pubmed database. The mus musculus is known as house mouse or laboratory mouse. Complete DNA sequence of mus musculus is approximately 2.3 GB in size and stored in 23 fasta format files (http://www.ncbi.nlm.nih.gov/pubmedhealth, 2011).

The experiments are performed in three different platforms that are denoted in Table 2. In the table, *M1* is a desktop workstation, while *M2* and *M3* are laptop computers. Each platform have multiple CPU cores (http://www.intel.com/, 2011).

In order to minimize jitter in the results, each experiment at each platform is repeated ten times. While outliers are discarded, average of the remaining eight results is reported as final experimental output.

All codebase is written in C++ and compiled with GCC (GNU C Compiler). We wrote the code with an Object Oriented Programming approach to have same environments with different algorithms. We effectively used Boost Libraries (http://www.boost.org, 2013) to handle external functions.

In terms of evaluation, fundamental performance criterion is the total CPU cycles. Best algorithm is the one which ensures minimum number of CPU cycles during execution. Consequently that algorithm can yield fastest computation time and introduce best energy efficiency.

Various hardware based factors affect the CPU cycle performance. We test the following hardware counters that affect total CPU cycles: Instructions per clock, Level 1 cache access, Conditional branch instructions, and finally, Conditional branch instructions mis-predicted.

While the evaluation of branch counters denotes whether the control flow in the instruction pipeline is efficient, the cache counters collect overall cache misses inside multiple levels of fast on-chip cache memory.

**Table 3. Hardware and Software Profiles of the Test Platforms**

| **Platform** | ***M1*** | ***M2*** | ***M3*** |
|---|---|---|---|
| CPU | 2 X Intel Xenon E5 -2640 @2.5 GHZ<br><br>16 Cores | Intel i7 820Q @1.73 GHZ 4 Cores | Intel i5 2410M @2.30 2 Cores |
| Cache | 15MB | 8MB | 3MB |
| RAM | 64MB | 8MB | 4GB |
| KERNEL | 3.2.0-59-generic | 3.10.25-Gentoo | 3.5.0-52-generic |
| GCC | 4.6.3 | 4.7.3 | 4.6.3 |

The hardware counters are collected from version 5.3.0 of PAPI (Browne and others, 2000). To collect the hardware counters we used low level interface of PAPI which ensure more reliable results. We used low level interface events to calculate cache accesses, cache reads, cache misses and branch instructions.

### 4.1. Execution of the Algorithms

During evaluation, we report total CPU cycles to present objective execution time of the exact match. In contrast, we do not present elapsed time in order to factor out the different clock frequencies. Recall that, elapsed time can be computed using Total CPU cycles and CPU clock speed of each platform.

### 4.2. Total Cycles

Experimental results in Figure 3 denote that BAP4bit algorithm outperforms in each platform since BAP4bit requires minimum number of CPU cycles. In other words, it presents the fastest algorithmic option during string matching and may optimize energy efficiency. The results imply that BAP4bit can be used for DNA search on large databases efficiently. We believe better results are caused by two factors: First, it introduces large skip distance when a mismatch occurs. Second, the algorithm is more compact, and comprises simple loops and few variables.

Another interesting result of Figure 3 is based on the performance comparison of the three architecture platforms. Recall that *M1* has more cache and memory capacities than others. Hence, it is expected that *M1* outperforms other platforms. On the other hand, advantage of *M2* is the linux distribution that enable more flexibility. *M2* runs on a Gentoo linux distribution, which is compiled from source code. Hence, Gentoo can offer dramatically flexible customization opportunities and booting this fully optimized operating system with a customized kernel may reduce CPU instruction requirement. Because of customization oppotunities, hardware resources can be reserved to string matching programs.

While CPU cycles can be an objective verification, it is necessary to analyze the factors that increase/decrease CPU cycles. Due algorithmic and architectural structures, each algorithm requires various amounts of branch instruction executions, branch mis-predictions and cache misses. Such hardware counters determine the overall CPU cycles of a process. In the next section we analyze the most effective hardware counters.

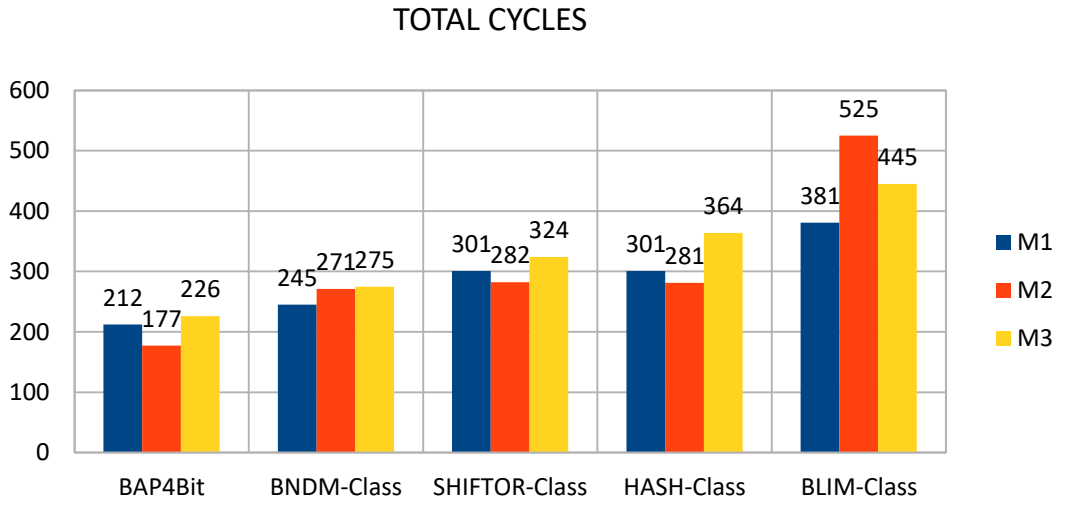



***Figure 3:***
*Cycle cost of the algorithms*

### 4.3. Level 1 Data Cache Access and Register Spills

Register spills are one of the fundamental factors that can increase CPU cycle requirement. In computer architecture, CPU is the fastest component of the computation and its memory is the register. In the best case, information take place inside registers and CPU proceeds without stalls. On the other hand, if the information does not exist inside CPU registers, it should be fetched from the peripheral devices such as cache or RAM. During the fetch, CPU have wait idle and cause CPU stalls. In literature, such event is defined as register spilling and is a important research field. A detailed analysis on register spilling and its effect can be found on (Appel, 2000).

In order to predict overall register spills, we wanted to analyze L1 data cache accesses (Browne and others, 2000). Due to CPU counter access limitations on platforms, we had chance to analyze only *M2* platform for this experiment. In Figure 4, results denote BAP4bit algorithm present at least 50% improvement over L1 data cache access during execution and minimize register spills inside CPU hardware. Results denote that minimized L1 data cache accesses can reduce total CPU cycles during execution. As a consequence, computation can be finished by using less number of CPU cycles. We predict that fundamental advantage of our algorithm is caused by L1 data cache accesses. While *M1* and *M3* does not permit analysis of L1 data cache access, results of *M2* presents valuable hints about L1 data cache accesses.

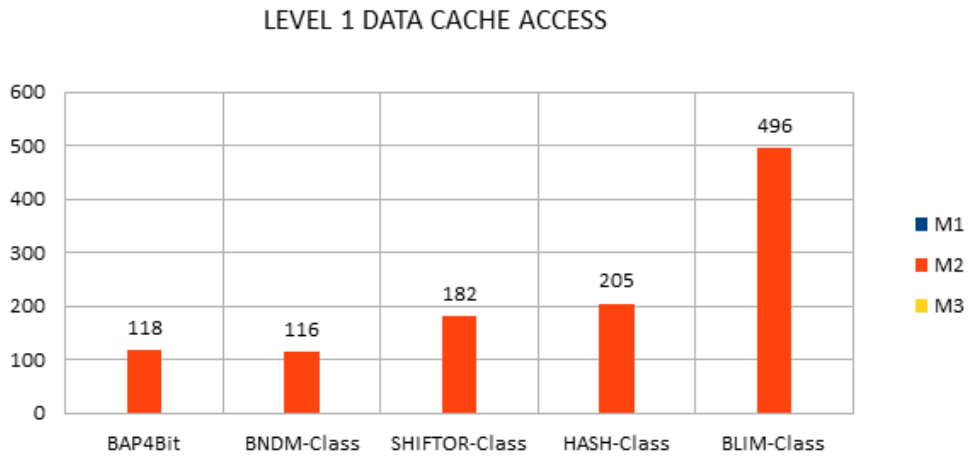



***Figure 4:***
*L1 cache accesses during executions*

### 4.4. Instructions Per Clock

One of the factors of Total Cycles experiment is based on Instruction per Cycle, IPC. The IPC counter denotes whether an algorithm leverages instruction level parallelism well or not. In Table 4, we present the ILP performance of the algorithms. Results denote that BLIM-class algorithm presents the best ILP performance, whereas BNDM-class denotes the worst. This result is very interesting and gives hint about the instruction level parallelism behaviors of CPU's. Concretely, there should be a correlation between ILP and simple algorithms. Hence, trade-off between "being simple" and "estimation of long skip distance" should be determined carefully.

### 4.5. Total Instruction Executions and Branch Mis-predictions

Two other performance evaluation techniques are branch instructions and branch mis-predictions. For efficiency, current CPU architectures are based on pipelining where next instructions are predicted and loaded into pipeline before actual use. In this way, architectures aim to reduce memory latency. However, if the next instruction is mis-predicted, the CPU pipeline will be flushed and CPU stalls until correct instructions are loaded into pipeline. Consequently, branch mis-predictions slow down the execution speed.

**Table 4. Instruction Level Parallelism Tests (Instructions Per Clock)**

| Algorithm | M1 | M2 | M3 |
|---|---|---|---|
| BAP4Bit | 1,33 | 1,42 | 1,25 |
| BNDM-Class | 1,07 | 0,88 | 0,96 |
| SHIFTOR-Class | 1,88 | 1,85 | 1,75 |
| HASH-Class | 2,41 | 2,26 | 1,99 |
| BLIM-Class | 2,63 | 1,81 | 2,25 |

During algorithm design, we should consider estimation behaviors of branch predictors and present efficient solutions that minimize branch mis-predictions. (Patterson and others, 2007) denotes that branch predictors are impotent to nested conditions and loops. Here, our packed variable try to keep multiple information inside bitwords to reduce nested loops and conditions. We present branch executions test in Table 5. Results imply BAP4bit algorithm presents the minimum number of branch instruction executions in all 3 platforms. Therefore such result minimizes the CPU cycle requirement of BAP4bit algorithm.

**Table 5. Conditional Branch Instructions**

| **Algorithm** | **M1** | **M2** | **M3** |
|---|---|---|---|
| BAP4Bit | 227 | 230 | 227 |
| BNDM-Class | 374 | 418 | 374 |
| SHIFTOR-Class | 680 | 680 | 680 |
| HASH-Class | 680 | 680 | 680 |
| BLIM-Class | 859 | 859 | 859 |
| **Note :** Values x $10^7$ | | | |

Another interesting observation implies that *M1* is much more efficient platform than others. During execution of each string processing algorithm, *M1* required minimum number of branch instruction executions. These results denote that *M1* has more powerful pipelining design than other platforms. Since *M1* is a desktop workstation, results imply that workstations are efficient especially during conditional instruction executions since they have powerful pipelining capacity.

The performance of *M2* and *M3* depends on the algorithm. For instance *M3* and its two CPU's show poor performance during execution of Shift OR and BLIM style algorithms where source of algorithm include less conditions. In contrast, source of BNDM-Class handles complex conditions. In such algorithm *M3* beats *M2* which has 4 CPU's. The result is interesting since parallelism does not guarantee better performance, especially during the execution of complex conditions of a program.

We also consider branch mis-prediction rates of the algorithms in Figure 8. Results show that *M2* present poor performance on this test. We need to mention that *M2* is first generation of i7, whereas *M1* and *M3* are at least second generation. Result denote CPU enhancements about Conditional Branch mis-prediction rates.

In terms of CPU counters perpective, Table 6 presents that algorithms without skip mechanism outperforms on this category. In fact, without skip mechanism, string processing could be very plain and recovered from branch mis-predictions. On the other hand, BAP4bit outperforms BNDM-Class strongly. While BAP4bit has a skip mechanism, it tries to minimize conditional procedure. In Table 6, results show that ShiftOR-Class minimize the branch mis-predictions. On the other hand BNDM algorithms include complex conditions and nested loop. Consequently, it is mortal to branch mis-predictions.

In summary, results denote that BAP4bit is an efficient algorithm for bitwise string matching with classes problem. The algorithm minimizes L1 data cache accesses and branch instructions. Consequently it presents minimum number of CPU cycles during pattern search.

**Table 6. Conditional Branch Instructions Mispredicted**

| **Algorithm** | **M1** | **M2** | **M3** |
|---|---|---|---|
| BAP4Bit | 1180 | 1303 | 1182 |
| BNDM-Class | 3971 | 5063 | 3981 |
| SHIFTOR-Class | 54 | 56 | 55 |
| HASH-Class | 55 | 54 | 54 |
| BLIM-Class | 69 | 100 | 70 |
| **Note :** Values x $10^7$ | | | |

## 5. CONCLUSIONS

This study considered string matching with classes problem where each pattern index may accept multiple alphabet letters. During analysis, we aimed at P53 DNA-binding consensus sequence, which is a vital tumor suppressor.

Due to large size of the sequence databases, search speed of the algorithms is important. Moreover, biological sequences can be observed in variable forms. Therefore, new biological search techniques are widely accepted as worthy.

Bitwise algorithms present fast string search due to intrinsic parallelism of bitwise operators. We present a novel bitwise algorithm which minimizes search time for special pattern types such as P53. The algorithm is based on packed variables which enhances modern hardware efficiency. It aims to simplify conditional statements inside code and register spills. Therefore, text search will not be decelerated by the peripheral devices of CPU.

Recently, CPU hardware technology presented new cache, branch prediction, and prefetch techniques. Such improvements could revise the performance outputs of the string matching algorithms. For better performance, algorithm designers must understand the prediction mechanisms of the new hardware components.

During hardware counter experiments, we compared results of three different platforms. In other words, we had chance to compare three different architectures for the same problem. Results denote that more CPU and parallelism may yield drawbacks during execution of codes that contain nested loops and conditions.

Computational analysis of biological data is in the early stages yet. Still, further enhancements are expected at least in the next decades. The hardware and software level advancements are strongly important to understand biological processes and health of human nature and biological processes.

## 6. ACKNOWLEDGMENT

This work is supported by the Dumlupinar University under Grant Number: BAP-2012-34. The authors are grateful to Professor Azmi YERLIKAYA for suggesting the mus musculus DNA sequence, P53 gene, and for the fruitful discussions.

## REFERENCES


**1.** Appel, W. and George, L. (2000) Optimal spilling for CISC machines with few registers, *ACM SIGPLAN Notices*, Vol 36 No 5, 243-253, doi: 10.1145/378795.378854

**2.** Baeza-Yates, R. and Gonnet, G. H., (1992) A new approach to text searching, *Communications of the ACM*, 35(10) , 74–82, doi: 10.1145/135239.135243

**3.** Boyer, R.S. and Moore, J.S. (1977 ) A Fast String Searching Algorithm, *Communications of the ACM*, 20, 10, 762-772, doi: 10.1145/359842.359859

**4.** Browne, S., Dongarra, J., Garner, N., Ho, G. and Mucci, P. (2000) A Portable Programming Interface for Performance Evaluation on Modern Processors, *The International Journal of High Performance Computing Applications*, 14:3, 189-204, doi:10.1177/109434200001400303

**5.** Durian, B., Holub, J., Peltola, H., and Tarhio, J. (2009) Tuning BNDM with q-grams, *Proceedings of the Workshop on Algorithm Engineering and Experiments ALENEX*. 29–37, doi: 10.1137/1.9781611972894.3

**6.** El-Deiry W. (1998) Regulation of p53 downstream genes, *Semin Cancer Biololgy*, 8 :345-357.

**7.** Fan, H., Yao, N., and Ma, H. (2009) Fast variants of the backward-oracle-marching algorithm, *Proceedings of the Fourth International Conference on Internet Computing for Science and Engineering*, IEEE Computer Society, 56–59

**8.** Faro, S. and Lecroq, T. (2010) The Exact String Matching Problem: *A Comprehensive Experimental Evaluation*, doi: 10.1145/2431211.2431212

**9.** Fuyao, Z. and Qingwei, L. (2009) A string matching algorithm based on efficient hash function, *Information Engineering and Computer Science – ICIECS*, doi: 10.1109/ICIECS.2009.5363191

**10.** Horspool, R. N. (1980) Practical fast searching in strings, *Software – Practice & Experience*, New Jersey, Volume 10 Number 6.

**11.** http://www.boost.org, Erişim Tarihi: 01.01.2013, Konu:C++ Boost Libraries :

**12.** http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. Erişim Tarihi: 1.1.2011, Konu: Intel Manual

**13.** http://www.ncbi.nlm.nih.gov/pubmedhealth/. Erişim Tatihi: 1.1.2014, Konu: PubMed Health Bethesda (MD): National Library of Medicine, mus musculus DNA sekansları

**14.** Karp, R. M. and Rabin, M. O. (1987) Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, New Jersey Volume 31 Issue 2, doi: 10.1147/rd.312.0249

**15.** Kern, S. E., Kinzler, K. W., Bruskin, A., Jarosz, D., Friedman, P., Prives, C., and Vogelstein B. (1991) Identification of p53 as a sequence-specific DNA-binding protein. *Science*. 252(5013):1708–11, doi: 10.1126/science.2047879

**16.** Kim, S. (1999) A new string-pattern matching algorithm using partitioning and hashing efficiently, *Journal of Experimental Algorithmics (JEA)*, JEA Homepage archive Volume 4, Article No. 2, doi: 10.1145/347792.347803

**17.** Külekci, O. (2012) On enumeration of DNA sequences, *Proceedings of ACM Conference on Bioinformatics, Computational Biology, and Biomedicine*, Orlando, doi: 10.1145/2382936.2382993

**18.** Külekci, O. (2008) A method to overcome computer word size limitation in bit-parallel pattern matching, *S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, Proceedings of the 19th International Symposium on Algorithms and Computation, ISAAC*, Lecture Notes in Computer Science, Springer-Verlag, Berlin volume 5369, 496–506, doi: 10.1007/978-3-540-92182-0_45

**19.** Knuth, D., Morris, J. H. and Pratt, V. (1977) Fast pattern matching in strings, *SIAM Journal on Computing*, 6 (2), 323–350, 10.1137/0206024

**20.** Navarro, G., and Raffinot, M. (2000) Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, Volume 5, New York, doi: 10.1145/351827.384246

**21.** Özcan, G., and Ünsal, O. S. (2015) Fast bitwise pattern matching algorithm for DNA sequences on modern hardware, *Turk J Elec Eng & Comp Sci*, Vol 23 (2015), pp.1405-1417 doi: 10.3906/elk-1304-165

**22.** Patterson, D., Hennessy, J., Arpaci-Dusseau, A. (2007). Computer architecture: a quantitative approach. Morgan Kaufmann,

**23.** Vintan, L. N. "Towards a High Performance Neural Branch Predictor (1999) *Proceedings of the Int'l J. Conf. on Neural Networks*, doi: 10.1109/IJCNN.1999.831066

**24.** Sunday, D. M. (1990) A Very Fast Substring Search Algorithm. *Communications of the ACM*, New York. , 33, 8, 132-142, doi: 10.1145/79173.79184