WHAM: A High-throughput Sequence Alignment Method

Yinan Li Allison Terrell Jignesh M. Patel

University of Wisconsin–Madison
{yinan, aterrell, jignesh}@cs.wisc.edu

ABSTRACT

Over the last decade the cost of producing genomic sequences has dropped dramatically due to the current so called "next-gen" sequencing methods. However, these next-gen sequencing methods are critically dependent on fast and sophisticated data processing methods for aligning a set of query sequences to a reference genome using rich string matching models. The focus of this work is on the design, development and evaluation of a data processing system for this crucial "short read alignment" problem. Our system, called WHAM, employs novel hash-based indexing methods and bitwise operations for sequence alignments. It allows richer match models than existing methods and it is significantly faster than the existing state-of-the-art method. In addition, its relative speedup over the existing method is poised to increase in the future in which read sequence lengths will increase.

The WHAM code is available at http://www.cs.wisc.edu/wham/.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information search and retrieval—*search process*; H.2.4 [**Database Management**]: Systems—*textual databases*

General Terms

Algorithms, performance

Keywords

Sequence alignment, approximate string matching, bit-parallelism

1. INTRODUCTION

Last summer marked the 10^{th} anniversary of the sequencing of the first human genome [1, 21]. This key scientific discovery has been a turning point for modern life sciences and has dramatically changed the way in which researchers approach nearly every aspect of biomedical sciences, ranging from deciphering basic cellular mechanisms to drug discovery and drug design for personalized medicine. A crucial part of this first human genome assembly was using advanced data processing methods to assemble the entire genome from vast sets of data items, each of which described only a small portion (called a "read") of the genome. The human genome project delivered its first draft ahead of schedule, primarily because of the use of advanced data processing methods.

Existing sequencing technology has come a long way from the technology of 10 years ago, both in terms of speed (with which they can read parts of the genome) and the cost of reading each of the 3 billion "bases" that make up the whole human genome. As a consequence, while the first human genome took a few billion dollars to assemble, today with the help of next-generation sequencing machines an entire genome can be read for a few thousand dollars.

Interestingly, data processing techniques are an even more crucial aspect of assembling genomes today. The pressing problem now with the next-generation of sequencing is the so called "next-generation gap" – namely, the data processing cost associated with genomic analysis is now the dominating cost of producing a genome sequence [15]. A key component of this data processing task is the alignment of short sequence reads [15]. The focus of this paper is on the design and evaluation of a novel data processing system for speeding up this alignment task by an order of magnitude and more, while accommodating flexible match models. Our system is called WHAM – an acronym for **W**isconsin's **H**igh-throughput **A**lignment **M**ethod. WHAM employs novel database-style indexing, optimization and query processing techniques.

At its heart, the short sequence read alignment problem is similar to the common substring matching problem in data processing systems. At a high-level, the next-gen sequencing works as follows: First the (DNA) sample to be sequenced is broken (by treating with restriction enzymes or using mechanical force) into a number of short pieces. These pieces are then cut into equal-length fragments called "reads". The bases/characters in each fragment are read by the sequencing machine. From the computational perspective the set of reads/fragments can be modeled as a set of strings/sequences. The subsequent data processing task involves aligning each read sequence against some scaffolding/reference genome sequence.

For example, when assembling the genome for a specific individual to look for variations that cause specific diseases, the reference genome is often the publicly available human genome. Thus, from the string matching perspective, one can view the computation task as matching a set of equal-length short strings (reads) against a large reference string (the entire genome), using some string matching model, as described below.

When a read is compared to the reference genome, it is either deemed as a valid or an invalid alignment. The validity of an alignment is measured in terms of mismatches and gaps. A mismatch occurs when a base on the reference genome and a base on the read are aligned, but aren't the same base. A gap occurs when a base is aligned with an empty space. Either or both of these alignments may be the "right answer" when aligning a read if it is advanta-

SIGMOD'11, June12–16, 2011, Athens, Greece.

geous to do so, such as when it allows many other base pairs in the set of reads to be aligned with the reference genome sequence. Often, aligners use two mismatches and no gaps as the definition of a valid alignment, but these models are likely to get more sophisticated in the future [15].

Previous methods for sequencing reads include Maq [12], SOAP [13], and Bowtie [6]. These techniques employ a compact index that works well for aligning so called "short reads", namely reads that are typically around 30-50 characters long. In particular, Bowtie – the leading state-of-the-art method – has focused on utilizing the properties of Burrows-Wheeler Transforms (BWT) [7] to index the reference sequence. A BWT-based index has a small memory footprint, making Bowtie feasible on computers with only 2GB of memory. However, memory is quickly becoming cheaper (while the genome size is constant), making it unnecessary to place such tight memory restrictions on the alignment software. In addition, BWT-based methods use a prefix matching technique that requires only a few iterations to produce a valid alignment. While this technique works well with short reads, its performance degrades rapidly as the read length increases. Unfortunately, impending technological advances with the next-generation sequencing machines are expected to push up the read lengths to produce what are called "long reads", and loosely refer to read strings that are many factors longer than the "short reads". This trend implies that the existing BWT-based aligners will likely become computationally infeasible in the near future. In addition, sequence aligners of the future will also need to accomodate more errors in the match models (for long reads) [9, 15], and the performance of existing methods rapidly deteoriates when using these richer match models.

Our WHAM method addresses the short-comings of the existing methods. It uses a novel hash-based index built on the reference genome that accommodates complex string matching models that are natural for read alignments. This hash index is optimized for space and speed, and to work efficiently with long reads. The hash index quickly finds potential hits for each read. This list of potential hits may contain some false positives that have to be checked using a precise test. This test for an actual hit is essentially a complex string matching function, which is computationally expensive. Another novel aspect of WHAM is the use of bitwise operations to perform this string matching test efficiently. Finally, WHAM uses a novel architecture that incorporates an optimizer (to optimize the indices that are built) and a compressor to reduce the size of the input data, providing a complete data processing and management module that fits into existing computational pipelines for high-throughput genomic sequencing.

We have compared WHAM using a number of real datasets against the leading and commonly used read alignment method – Bowtie. Our results show that WHAM is often orders of magnitude faster than Bowtie, and allows for richer match models. The relative speedup of WHAM over Bowtie increases as the read length as well as the number of errors increases, which bodes well for the future in which we expect to see longer reads that need to be matched with more relaxed/richer match models.

The remainder of this paper is organized as follows: The WHAM method is described in Section 2. Results from an extensive empirical evaluation are presented in Section 3. Section 4 discusses related work, and Section 5 contains our concluding remarks.

2. METHOD

2.1 Problem Statement

A sequence is a series of characters. For genomes, each character is either a nucleic acid represented by the symbols A, G, C, or T, or an unknown character, named N. The unknown character symbol, N, indicates that there is either uncertainty about which nucleotide belongs there, or that there is a repetitive region in the genome and all nucleotides in that region are turned into Ns (since for the genome sequencing task, it does not make biological sense to match reads to these known repetitive "junk" regions).

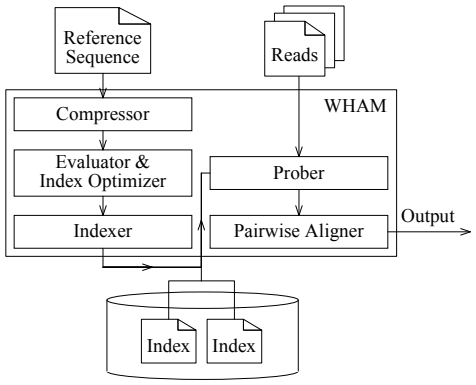


Figure 1: Architecture of WHAM

WHAM takes as input a set of query sequences of equal length, a database of reference sequence(s), and an error model. The error model specifies *alignment constraints* on three types of errors that are commonly used for read alignment. These three types of errors are: *substitutions*, which change characters in a sequence, *insertions* and *deletions*, which add or remove characters respectively. WHAM finds **all** (it is not a heuristic) valid alignments that satisfy the constraint on the number of errors in the set of query sequences. A valid alignment is formally defined as follows:

DEFINITION 1. *Given an error model specifying s substitutions, i insertions, and d deletions, a* valid alignment *for a given query sequence Q is all the subsequences in the reference sequence R that can be aligned by applying up to s substitutions, i insertions, and d deletions between Q and each matched subsequence in R.*

The following example demonstrates alignments between two query sequences and a reference sequence. Suppose we allow 1 substitution, 1 insertion, and 1 deletion. The first query sequence can be aligned with 1 substitution (shown as the underlined character) and 1 insertion (shown as a dash in the query sequence), while the second query sequence can be aligned with 1 substitution and 1 deletion (shown as a dash in the reference sequence). Both alignments satisfy the alignment constraint.

Reference Sequence	ATGGCCACAGAAGTT-GCGA
Query Sequence 1	GACCACA-AAGTT
Query Sequence 2	ACAGTAGTTAGC

The error model is sometimes aggregated to a generic k-error model which implies that the total number of substitutions, insertions and deletions is no more than k in the alignment.

2.2 System Architecture

Figure 1 shows the architecture of the WHAM system. WHAM provides an interface to ingest a set of reads as they come off the alignment machine and manages the computation associated with matching the reads to a specified reference genome. WHAM typically sits as a module in a larger computational pipeline. The WHAM system builds various indices (as described below) and also keeps track of various data management needs such as storing the indices on disk, tracking the usage, reporting to the overall workflow management system, etc. In this paper, we only focus on the components related to the sequence matching parts.

The *indexer* is responsible for building indices on the reference genome. At any given time, WHAM may have stored indices for multiple genomes (e.g. human, mouse, rat, etc.). These indices persist on disk, and are built only once when the reference genome sequence is loaded into WHAM. When the WHAM module is invoked, a single reference genome is specified (e.g. human) and the entire index for that reference genome is loaded into main memory. Note that in current next-gen sequence assembly, when analyzing multiple read sets, often the reference genome remains the same, so this index can stay loaded in main memory across multiple runs.

While the indices persist on disk, the indices are optimized for access in main memory. Since main memory capacities are increasing quickly and since WHAM builds indices on the reference genome (which is of a constant length), this design choice leverages the technological trend that makes it practical and economical to fit the entire genome index in main memory. The WHAM indexing method is described in Section 2.3.

The indexer relies on an indexing configuration setting that specifies the parameters that affect the index performance. This index configuration is generated by the *index optimizer* by comparing various indexing schemes whose costs are computed by the *evaluator* based on an analytical model (described in Section 2.7). The *prober* takes as input a set of query sequences (reads), and uses the same indexing configuration to break up each query sequence, and probes the indices (on the reference genome) to find potentially matching reference subsequences. The *pairwise aligner* is then performed to verify that the reference subsequence actually aligns to the query sequence under the alignment constraint (Section 2.5).

WHAM also employs a *compressor* to remove all unnecessary characters in the reference sequence before building an index.

2.3 Indexing Method

The WHAM indexing method is motivated by the observation that the inexact alignment between two sequences can be solved by the exact alignments on the fragments of sequences [19].

2.3.1 Indexing Schemes

We first introduce a basic indexing scheme based on the following observation: If two sequences, R and Q, match within k errors and $k+1$ non-overlapping fragments are taken from R, then the matching sequence Q contains at least one fragment that will match exactly with one fragment in R. This property is evident, since k errors cannot be placed into the $k+1$ non-overlapping fragments.

Using this observation, given the length l of the query sequences, all l-character subsequences in the reference sequence R are split into $k+1$ uniform-sized fragments of length $\lfloor \frac{l}{k+1} \rfloor$. A hash index is built on all the fragments. To align a query sequence, we break down the query sequence Q into fragments in an analogous way, and search against the fragments on the hash index to find the potential matching subsequences in R.

A problem with this indexing scheme is that the index keys (fragments) may be short and may have relatively low diversity, resulting in long chains in the hash index, which in turn degrades performance. For example, suppose a 36bps[1] query sequence is to be aligned within 2 errors. Three fragments are taken from the sequence, each of length 12bps. Now, for the moment assume that each character has only four possible values – `A`, `G`, `C`, and `T` – (so ignore `N`s for this example), then the 12bps fragment has a maximum of $4^{12} = 16,777,216$ possible values. However, the reference human genome sequence contains around 3 billion characters. When loading these 3 billion 12bps fragments into a hash index, each hash bucket in the index contains on average around 170 entries. When a probe is performed on the hash index, each of these entries needs to be scanned and verified one by one, which degrades the search performance significantly.

To address this problem, we extend the basic indexing scheme to a more general one based on Lemma 1 by relaxing the limit on the number of fragments.

LEMMA 1. *If two sequences, R and Q, match within k errors and $f (f > k)$ non-overlapping fragments are taken from R, then the matching sequence Q contains at least $f - k$ fragments which match exactly with fragments in R.*

We omit a detailed proof of Lemma 1 (in the interest of space), but it is easy to see that if more than k fragments contain errors, then the complete match must have more than k errors.

Based on Lemma 1, indices are built on $f - k$ fragments of all l-character subsequences of R, rather than on one fragment, where l is the length of the query sequences. In particular, each subsequence in the reference genome is split into f non-overlapping fragments of length $\lfloor \frac{l}{f} \rfloor$, from which we select $f - k$ fragments. A selected (unselected) fragment is called an *indexed (unindexed) fragment*. An unindexed segment between two indexed fragments is called an *interval*. For each selection, we concatenate the $f - k$ selected fragments into a *concatenation*. All concatenations are of the same length $(f - k) \cdot \lfloor \frac{l}{f} \rfloor$, and are inserted into hash indices. To search using these indices, we break the query sequence Q, into fragments in an analogous way (see Section 2.3.3 below for details). Note that the extended indexing scheme is identical to the simple indexing scheme, when $f = k + 1$.

Taking the 36bps example again, if four fragments are taken from the sequence, the concatenations contain two fragments, and are 18bps long. There are $4^{18} = 68,719,476,736$ possible values for 18bps concatenations, many more than the characters in the whole human genome sequence. Consequently, the hash collision issue is dramatically mitigated.

2.3.2 Building Indices

The *indexer* component in WHAM is responsible for building indices on all concatenations of fragments for all subsequences of length l in the reference genome R. The goal of the indexer is a) to separate the concatenations into several hash indices to reduce the collisions in the bucket for each hash index, and b) to remove all duplicated concatenations to reduce the space cost.

We observe that the concatenations can be categorized into a few groups based on the intervals between the indexed fragments. The concatenations in a group are matched with others by sliding the concatenations. As an example, Figure 2 illustrates the categorization on six concatenations of choosing two indexed fragments from four fragments. The indexed fragments are represented by rectangles, whereas the unindexed fragments are represented by lines. Among these concatenations, concatenation 1, 3, and 6 contain two contiguous indexed fragments, and are categorized into group I. In concatenation 2 and 5, there is an unindexed fragment between the two indexed fragments. These concatenations belong to group II. Concatenation 4 contains a two fragment gap between the two indexed fragments, and is categorized into group III.

To achieve the first goal (i.e. to separate concatenations into several hash indices), the indexer separately loads the concatenations of all subsequences in R into various hash indices on the basis of groups. The number of hash indices (groups) that is required follows from Lemma 2.

LEMMA 2. *If f-fragment query sequences are aligned within k errors, then C_k^{f-1} indices are required for the alignments.*

[1]The term "bps" is commonly used in genomics and refers to base pairs. The bps is essentially the sequence length.

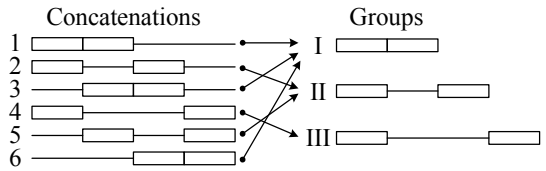


Figure 2: Categorization of six concatenations ($f = 4, k = 2$)

PROOF. The number of indices is equal to the number of groups. One way to count the number of groups is to count the number of ways to select k unindexed fragments from f fragments so that the first fragment is an indexed fragment, because other selections in the group can be matched to this kind of selection by sliding the sequence. The number of these selections corresponds to the number of ways to select k unindexed fragments from $f-1$ fragments (the first fragment is fixed to be an indexed fragment). This is given by the formula C_k^{f-1}. □

To achieve the second goal (i.e. to reduce the indexing space), only one representative concatenation in each group is loaded into the hash index, for each subsequence in R. Other concatenations in the group will be loaded (as a representative concatenation) when sliding the subsequence, because a concatenation is identical to other concatenations in the same group when sliding the subsequence by one or a few fragments.

To build the indices, we slide a l-character window along the reference sequence R (an analysis of various sliding window schemes may be found in [8]). The C_k^{f-1} concatenations associated with the C_k^{f-1} groups are extracted from the subsequence in the window and separately loaded into C_k^{f-1} hash indices. The number of entries in each hash table roughly equals the number of characters in the reference sequence.

EXAMPLE 2.1. *Figure 3 illustrates the process of building indices on a sample sequence for alignments with two errors. Suppose that the query sequences contain 12 characters and are split into 4 fragments, each of length 3 characters. Thus, the indices are built on concatenations of length 6. For each subsequence in the reference sequence, we generate three concatenations that consist of two indexed fragments with zero, one, and two unindexed fragments in the interval, respectively. The three concatenations are then inserted into the three indices as the index keys.*

The space complexity of each hash index is $O(n)$, where n is the number of characters in the reference sequence R. The number of hash indices that are built for a reference sequence is dictated by Lemma 2.

2.3.3 Searching Indices

The *prober* (see Figure 1) is responsible for searching the query sequences (reads) against the indices. First, we describe the search procedure of the prober for alignments with only substitutions, and then (in the next paragraph) extend it to support indels (insertions or deletions). To align a query sequence Q, it is split and concatenated in an analogous way as the subsequences in R. All C_k^f concatenations are assembled with $f-k$ fragments in various combinations. Next, we search each concatenation on its associated indices. Then, the positions of the matched concatenations in R are retrieved. With these positions, we extract l-character subsequences from R as "candidate" occurrences, which are then checked using the technique described in Section 2.5.

Next, we extend our technique to support indels. First, all the C_k^f concatenations used for alignments with only substitutions are generated. Then, for each concatenation, we extend it to a set of

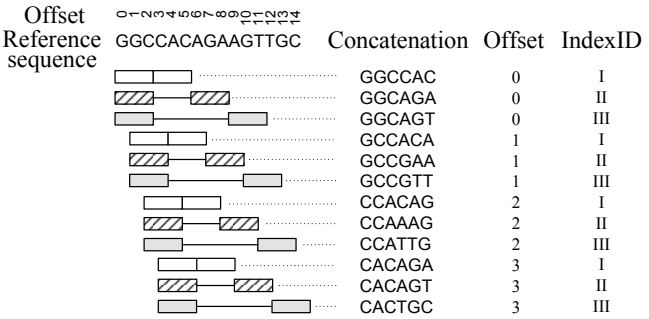


Figure 3: Building three hash indices ($f = 4, k = 2$)

new concatenations that are used to align the query sequence with indels. The basic idea is to slide some indexed fragments by a few characters. More specifically, if the error model specifies i insertions and d deletions, we place up to i insertions and up to d deletions in all intervals in the query sequence. The concatenations are then assembled with indexed fragments whose offsets are recalculated based on the number of indels in the intervals. The probes are performed on all possible placements of indels in all intervals.

The numbers of probes is formally given by Lemma 3.

LEMMA 3. *If f-fragment query sequences are aligned within k errors including i insertions and d deletions, then the number of probes for each query sequence is given by:*

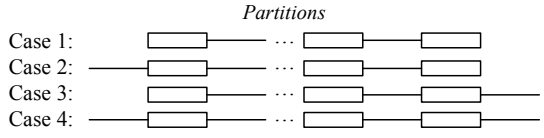
$$N_{probe} = \sum_{h=0}^{k} (C_{h-1}^{k-1} + 2C_h^{k-1} + C_{h+1}^{k-1}) C_h^{f-k-1} C_i^{i+h} C_d^{d+h}$$

In particular, when $i = d = 0$, the number of probes is C_k^f.

PROOF. We have two main tasks to accomplish. The first task is to count how many ways can one select k unindexed fragments from f fragments to compose h intervals (the unindexed segments between indexed segments). Let $F(f, k, h)$ be the number of selections. The second task is to count the number of probes on each selection with h intervals, denoted as $G(h, i, d)$. Thus, we sum the products of $F(f, k, h)$ and $G(h, i, d)$ to compute the total number of probes, obtaining

$$N_{probe} = \sum_{h=0}^{k} F(f, k, h) \cdot G(h, i, d). \quad (1)$$

First, we count the number of selections with h intervals. A selection can be viewed as interleaved segments between interval and indexed segments. Given a selection with h intervals, there are four possible arrangements of h intervals, $h+1$ indexed segments, and (0, 1 or 2) unindexed segments at the ends, as shown in the following figure:



In case 1, these kind of selections have h intervals and $h+1$ indexed segments. The number of selections is equal to the number of ways to partition k unindexed fragments into h intervals, and partition $f-k$ indexed fragments into $h+1$ indexed segments. The formula for the number of partitioning n elements into s non-empty sets is given by C_{s-1}^{n-1}. Thus, there are C_{h-1}^{k-1} and C_h^{f-k-1} ways to partition unindexed fragments and indexed fragments, respectively. By the multiplication principle, there are $C_{h-1}^{k-1} C_h^{f-k-1}$ partitions

in this case. Similarly, in case 2 and case 3, we need to partition k unindexed fragments into $h+1$ sets (h intervals and one unindexed segment at one end), and partition $f-k$ indexed fragment into $h+1$ indexed segments. There are $C_h^{k-1}C_h^{f-k-1}$ partitions in case 2 or case 3. In case 4, we need to partition k unindexed fragments into $h+2$ sets (h intervals and two unindexed segments at the both ends). The number of partitions is given by $C_{h+1}^{k-1}C_h^{f-k-1}$. By applying the addition principle to cases 1-4, we have

$$F(f,k,h) = (C_{h-1}^{k-1} + 2C_h^{k-1} + C_{h+1}^{k-1})C_h^{f-k-1}. \quad (2)$$

Next, we count the number of probes on a selection with h intervals. Consider the following correspondence. We put up to i insertions and up to d deletions into h intervals. The number of placements is equal to the number of probes, because we need to perform one lookup for each placement. One way to solve the problem is to add a "virtual" interval where all the unselected indels are placed. Thus the task is to decide how to partition i insertions into $h+1$ intervals, and partition d deletions into $h+1$ intervals. Given the formula C_n^{n+s-1} on the number of ways to partition n elements into s sets, there are C_i^{i+h} and C_d^{d+h} ways to partition i insertions and d deletions, respectively. By applying the multiplication principle, we have $C_i^{i+h}C_d^{d+h}$ different kinds of partitions on the combinations of insertions and deletions, obtaining

$$G(h,i,d) = C_i^{i+h}C_d^{d+h}. \quad (3)$$

Finally, we complete the derivation by substituting Equation 2 and 3 into Equation 1. □

EXAMPLE 2.2. *We align a query sequence* GACCACAAAGTT *with one substitution and one insertion to the sample sequence shown in Figure 3. The query sequence is split into 4 fragments as* GAC|CAC|AAA|GTT*. Then, we concatenate two of the four fragments into concatenations: (1)* GAC|CAC*, (2)* GAC|AAA*, (3)* CAC| AAA*, (4)* GAC|GTT*, (5)* CAC|GTT*, (6)* AAA|GTT*. In addition, we generate concatenations that are used to find alignments with one insertion. Since the concatenations 2, 4, and 5 have an interval between the two indexed fragments, we move the window of the second indexed fragment one character left to apply the insertion, and then get three more concatenations (7)* GAC|CAA*, (8)* GAC|AGT*, and (9)* CAC|AGT*. Following the relationship between the concatenations and groups as shown in Figure 2, we probe concatenation 1, 3, and 6 on index I, concatenation 2, 5, 7, and 9 on index II, 4 and 8 on index III. The concatenation (9)* CAC|AGT *in index II is found and the subsequence in R with 13 characters* GGCCACAGAAGTT *is returned.*

2.4 Hash Index Structure

WHAM uses tailored compact hash tables to index all concatenations generated by the *indexer* (see Figure 1). Hash indices are designed as static cache-efficient structures. The hash buckets are 32-bit integer arrays. For each slot in a bucket, the most significant bit (MSB) is used as a collision bit. If only one concatenation is hashed into a slot, then the collision bit of the slot is set to 0. The remaining 31 bits represent the positions of the concatenations in the reference sequence R[2]. Otherwise, if several concatenations are hashed into the same slot, then the collision bit is set to 1. The remaining 31 bits point to an array that contains the positions of all concatenations hashed into that slot. The MSB of the last entry in the overflow list is set to 1 to indicate the end of the slot's list.

[2]The offsets in the compressed reference sequence can be represented by 31-bit integers in most cases. See Section 2.6 for details. We also support 32-bit offsets by storing the MSBs of buckets and overflow entries in separate bit vectors.

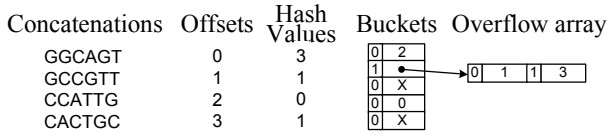


Figure 4: Hash table of index III shown in Example 2.1

EXAMPLE 2.3. *Figure 4 illustrates the hash table for Index III shown in Example 2.1 (Figure 3). The offsets of the four concatenations in the reference sequence are numbered from 0 to 3. In the bucket slots 0 and 3, only one concatenation is hashed into the slot. As a result, the concatenation positions are directly stored in the slots with* `collision bit = 0`*. In the bucket slot 1, we maintain a pointer to the starting position of the slot's overflow list in the overflow array, with* `collision bit = 1`*. Slots 2 and 4 are empty.*

Both space and time cost are taken into consideration in the design of the hash index. The design of the overflow array makes effective use of CPU caching, since the overflow entries are stored in sequential memory positions. In addition, to save space the design doesn't use next pointers that are required by linked lists. The collision bits save both time and space spent on the overflow list when a bucket contains only one concatenation. Moreover, the hash table only stores the positions of concatenations in the reference sequence R, rather than the actual concatenations, to save space, because the size of concatenations is typically a few times larger than that of the positions.

2.5 Pairwise Alignments

WHAM relies on the *pairwise aligner* (see Figure 1) to examine if the query sequence Q can be aligned to a potential matched subsequence in the reference sequence R within s substitutions, i insertions, and d deletions. Conventional pairwise alignment techniques are based on the Needleman-Wunsch dynamic programming method and take $O(l^2)$ time to compute the optimal alignment [16]. This alignment step can quickly become the bottleneck. WHAM uses an alternative technique that stores the sequences in a compact binary representation and leverages the ability of modern processors to compute bitwise operations fast. These two aspects are described next.

WHAM uses three bits to represent each base/symbol. The symbols A, C, G, T, and N are encoded as 000, 001, 010, 011, and 100 respectively. For example, the sequence GACT is encoded as the binary string $= (010000001011)_2$.

A sequence in WHAM is packed into a binary representation that can be fit into one or a few computer words. The proposed bitwise-based techniques (described below) manipulate a sequence as a whole, rather than manipulating the characters one by one. As a result, the alignment cost is not related to the length of sequences if they can fit into a word, but scales with the number of errors.

For ease of presentation, below we first introduce the alignment algorithms using a binary domain (each bit in a binary string is viewed as a base – so first consider a domain with only two symbols), and then extend our technique for the general case with larger number of symbols in the domain (approximately 5 in our case).

In Section 2.5.1, we first introduce three basic bitwise manipulations, that leverage the ability of modern processors to compute bitwise operations fast. Based on these manipulations, the bitwise-based alignment techniques to handle substitutions, insertions, and deletions on binary string are presented in Section 2.5.2, 2.5.3, and 2.5.4, respectively. Then, in Section 2.5.5 we describe the method for combining all three kinds of errors. In Section 2.5.6, the meth-

ods are extended to the general case when the domains have more than two symbols.

2.5.1 Basic Bitwise Manipulations

Given the basic bitwise operations, e.g. AND (&), OR (|), XOR ($\oplus$), and SHIFT ($\ll$, $\gg$), some of the combinations of these basic operations produce important bitwise manipulation techniques that we use in this paper. In particular, we have three bitwise manipulations that provide a fast way to find and manipulate the rightmost 1 bit [10]. The notations $\mathbf{R}$, $\mathbf{S}$, and $\mathbf{RS}$ are used to denote the three manipulations, respectively.

$$x\&(x-1): \quad \text{remove the rightmost 1 in } x. \qquad [\mathbf{R}]$$
$$x|-x: \quad \text{smear the rightmost 1 to the left in } x. \qquad [\mathbf{S}]$$
$$x\oplus -x: \quad \text{remove and smear the rightmost 1 in } x. \qquad [\mathbf{RS}]$$

EXAMPLE 2.4. *The example demonstrates how the three bitwise operations apply on a binary sequence x.*

$$x = (1000111101011100 10000)_2$$
$$x\&(x-1) = (1000111101011100 00000)_2 \qquad [\mathbf{R}]$$
$$x|-x = (1111111111111111 10000)_2 \qquad [\mathbf{S}]$$
$$x\oplus -x = (1111111111111111 00000)_2 \qquad [\mathbf{RS}]$$

2.5.2 Alignments with Substitutions

A bitwise alignment with substitutions is used to check whether two binary sequences can be aligned with up to s substitutions. The idea is that we first apply a bitwise XOR operation to identify the mismatched bits in the two given binary sequences, and then count the number of 1s in the resulting binary sequence, using the bitwise technique $\mathbf{R}$ (described above). After performing $u = x \oplus y$, we continue removing the rightmost 1 of u for s iterations. If the resulting binary sequence only contains 0-bits, the original u contains up to s ones. This implementation executes s bitwise AND operations and s subtraction instructions. The complexity of this step is $O(s)$, because s steps are taken to remove s possible substitutions before checking equality of the two strings.

EXAMPLE 2.5. *An example of an alignment with at most 2 substitutions between x and y is shown below.*

$$x = (10\underline{0}01111010\underline{1}110010110)_2$$
$$y = (10101111010011001 0110)_2$$
$$u = x \oplus y = (00100000000100000 0000)_2$$
$$u = u\&(u-1) = (00100000000000000 0000)_2 \qquad [\mathbf{R}]$$
$$u = u\&(u-1) = (00000000000000000 0000)_2 \qquad [\mathbf{R}]$$
$$(u = 0) = \texttt{true}$$

2.5.3 Alignments with Insertions

A bitwise alignment with i insertions is used to determine whether a binary sequence y can be matched to x by adding at most i gaps. This algorithm is based on the following greedy algorithm: we treat the rightmost i mismatches as insertions, and then examine whether the remaining prefix of x is the same as that of y.

Table 1 demonstrates the first steps of the bitwise alignment algorithm with up to i insertions. First, a bitwise XOR operation is applied between right-aligned x and y, e.g. $u_0 = x \oplus y$. Suppose that the rightmost 1 in u_0 is at position a_0, then the binary sequence u_0 can be represented by $(*^{n-a_0-1}10^{a_0})_2$[3], where $*$ is an arbitrary value in $\{0, 1\}$. Since the a_0-th bits in x and y are mismatched, a_0 can be treated as an insertion position, if x and y can be aligned with only insertions.

[3]We use exponentiation to denote bit repetition, e.g. $(1^4 0^2)_2 = (111100)_2$.

Table 1: Identifying the first two insertion positions ($*$ is an arbitrary binary value)

Index	(2nd insert) $\ldots$ a_1 $+1\ -1$	(1st insert) $\ldots$ a_0 $+1\ -1$	$\ldots 0$
$u_0 = x \oplus y$	$\ldots$ $***$	$\ldots$ $*10$	$\ldots 0$
$u_1 = (x \ll 1) \oplus y$	$\ldots$ $*10$	$\ldots$ $0**$	$\ldots *$
$m_0 = u_0 \oplus -u_0$	$\ldots$ 111	$\ldots$ 100	$\ldots 0$
$u_1 = u_1 \& m_0$	$\ldots$ $*10$	$\ldots$ 000	$\ldots 0$

In the next step, we need to find the second insertion position. Since the subsequence of x on the left of the first insertion position must be shifted one character left to be aligned with y, we compute $u_1 = (x \ll 1) \oplus y$. In u_1, all bits between the first and second insertion position are 0s, whereas the bits at other positions are arbitrary binary values. If the second insertion is at position a_1, then u_1 can be represented by $(*^{n-a_1-1}10^{a_1-a_0-1}*^{a_0+1})_2$. Next, we generate a mask $m_0 = u_0 \oplus -u_0$ by removing and smearing the rightmost 1 to the left of u_0 (apply $\mathbf{RS}$), which is of the form of $(1^{n-a_0-1}0^{a_0+1})_2$. The mask m_0 is applied on u_1 to clear all bits to the right of the first insertion position. Then, u_1 can be represented by $(*^{n-a_1-1}10^{a_1})_2$. Thus, the position of the rightmost 1 in u_1 is the second insertion position. Continuing this for i iterations, we will find all the i insertion positions.

Finally, we get a bit-difference sequence u_i between the two sequences for all bits on the left of all i insertion positions. If and only if all bits in u_i are 0s, then x can be aligned to y with i insertions.

The algorithm can also examine alignments with less than i insertions. To verify this claim, suppose that x can be aligned to y with $i'(i' < i)$ insertions. After executing the first i' iterations, we get $u_{i'} = (...000)_2$. In the next iteration, we have

$$m_{i'} = u_{i'} \oplus -u_{i'} = (\ldots 000)_2$$
$$u_{i'+1} = (x \ll k) \oplus y = (\ldots * **)_2$$
$$u_{i'+1} = u_{i'+1} \& m_{i'} = (\ldots 000)_2$$

Continuing this computation, by $i - i'$ iterations, we have $u_{i'+1} = u_{i'+2} = \cdots = u_i = (\ldots 000)_2$. According to $u_i = (\ldots 000)_2$, the algorithm identifies an alignment with at most i insertions.

The pseudocode for this technique is shown in Algorithm 1. The complexity of this algorithm is $O(i)$, because we take i steps to remove the effects of all possible insertions before checking equality of the two strings.

EXAMPLE 2.6. *This example demonstrates how we can use 12 bitwise operations to align two binary strings x and y with at most two insertions.*

$$x = \quad (1000111101011100 10110)_2$$
$$y = (1000111101\underline{1}10111\underline{0}0010110)_2$$
$$u_0 = x \oplus y = (1010110010111001 0000000)_2$$
$$u_1 = (x \ll 1) \oplus y = (1100100011000000 0111010)_2$$
$$m_0 = u_0 \oplus -u_0 = (1111111111111110 0000000)_2 \qquad [\mathbf{RS}]$$
$$u_1 = u_1 \& m_0 = (1100100011000000 0000000)_2$$
$$u_2 = (x \ll 2) \oplus y = (0000000000110010 1001110)_2$$
$$m_1 = u_1 \oplus -u_1 = (1111111110000000 0000000)_2 \qquad [\mathbf{RS}]$$
$$u_2 = u_2 \& m_1 = (0000000000000000 0000000)_2$$
$$(u_2 = 0) = \texttt{true}$$

Algorithm 1 Sequence alignment within i insertions

1: $u_0 \leftarrow x \oplus y$;
2: **for** $k \leftarrow 1...i$ **do**
3: $\quad m_{k-1} \leftarrow u_{k-1} \oplus -u_{k-1}$; // remove and smear the rightmost 1
4: $\quad u_k \leftarrow (x \ll k) \oplus y$;
5: $\quad u_k \leftarrow u_k \& m_{k-1}$;
6: **return** $u_i = 0$;

2.5.4 Alignments with Deletions

The algorithm for a bitwise alignment with d deletions is similar to that of alignments with d insertions, except for two differences. First, we replace all left shifts by right shifts. Given the property $x \ll (-k) = x \gg k$, we use $x \ll -k$ to represent shifting x right by k bits. Second, the mask m_{k-1} is generated by the bitwise technique $\mathbf{S}$ instead of the bitwise technique $\mathbf{RS}$. We smear the rightmost 1 to the left, but do not remove the rightmost 1 because this position is not skipped by an insertion.

2.5.5 Alignments with all the Three Types of Errors

The final algorithm to align within s substitutions, i insertions and d deletions simply enumerates all possible orders of these three types of errors, and then verifies each permutation. Algorithm 2 shows the pseudocode. In the outer loop, all permutations with s substitutions, i insertions and d deletions are enumerated. We use -1, 0, +1 to denote a deletion, a substitution, and an insertion, respectively. A permutation represents the order of the error types that occurred in an alignment. For example, $o_3o_2o_1o_0 = -1, 0, +1, 0$ implies an alignment with four errors, which are a substitution, an insertion, another substitution, and a deletion, in right-to-left order.

For each permutation $o_{s+i+d-1}o_{s+i+d-2}\ldots o_1o_0$, we examine whether the two given sequences can be aligned with the constraints on the number and the order of error types. In the beginning, the current binary sequence u is initialized as the bit-difference sequence between the right-aligned x and y (Line 4). In the inner loop, we clear all the errors in the permutation from right to left. In iteration k, we manipulate the sequence u according to the error type indicated by o_k. If the error is supposed to be a substitution, then we remove the rightmost 1 in u (Line 6). Thus, the substitution is cleared so that we can continue identifying the next error by finding the next rightmost 1. Otherwise, we first generate a mask m by removing and smearing the rightmost 1 in u if the error is an insertion (Line 9), or by smearing the rightmost 1 in u if the error is a deletion (Line 11). Next, we update $u \leftarrow (x \ll (o_0 + o_1 + ... + o_k)) \oplus y$, and use the mask m on u to clear the rightmost bits that have arbitrary values (Line 13). As a result, the rightmost 1 in the updated u implies the next error position. Continuing this for $s + i + d$ iterations, if the resulting sequence u contains only 0s, we produce a successful alignment.

The algorithm also examines alignments with fewer than the constrained number of errors. Suppose that x can be aligned to y with $s'(s' \leq s)$ substitutions, $i'(i' \leq i)$ insertions, and $d'(d' \leq d)$ deletions. Then, there exists a permutation R such that $P = o_{s+i+d}o_{s+i+d-1}...o_{s'+i'+d'+2}o_{s'+i'+d'+1}P'$, where P' is a permutation of s' substitutions, i' insertions, and d' deletions that can produce an alignment between x and y. After $(s'+i'+d')$ iterations of permutation P', we get $u = (\ldots 000)_2$. Regardless of whether the next error is a substitution ($u = u\&(u-1) = (\ldots 000)_2$), an insertion ($u = (\ldots **)_2\&(u\& - u) = (\ldots 000)_2$), or a deletion ($u = (\ldots **)_2\&(u \oplus -u) = (\ldots 000)_2$), we get $u = (\ldots 000)_2$ in the end of the next iteration. Continuing this step for $s+i+d-s'-i'-d'$ iterations, we have $u = (\ldots 000)_2$, and return true for this alignment.

Algorithm 2 Sequence alignment within s substitutions, i insertions, and d deletions

1: **for** each permutation $o_{s+i+d-1}, o_{s+i+d-2}, \ldots, o_1, o_0$ with s substitutions (0), i insertions (+1) and d deletions (-1) **do**
2: $\quad m = (111111...)_2$;
3: $\quad u = x \oplus y$;
4: $\quad$ **for** $k \leftarrow 0 \ldots s+i+d-1$ **do**
5: $\quad\quad$ **if** $o_k = 0$ **then**
6: $\quad\quad\quad u \leftarrow u\&(u-1)$; // remove the rightmost 1
7: $\quad\quad$ **else**
8: $\quad\quad\quad$ **if** $o_k = +1$ **then**
9: $\quad\quad\quad\quad m \leftarrow u \oplus -u$; // remove and smear the rightmost 1
10: $\quad\quad\quad$ **else**
11: $\quad\quad\quad\quad m \leftarrow u| - u$; // smear the rightmost 1 to the left
12: $\quad\quad\quad u \leftarrow (x \ll (o_0 + o_1 + ... + o_k)) \oplus y$;
13: $\quad\quad\quad u \leftarrow u\&m$
14: $\quad$ **if** $u = 0$ **then**
15: $\quad\quad$ **return true**;
16: **return false**;

The time complexity of this algorithm is $O(\frac{(s+i+d)!}{s!i!d!}(s+i+d))$, where $\frac{(s+i+d)!}{s!i!d!}$ is the number of permutations, and $s+i+d$ is the number of steps for processing each permutation. Recall that the complexity of the Needleman-Wunsch algorithm is $O(l^2)$, where l is the length of sequences. Given the fact that sequence aligner typically chooses 2 or 3 errors ($k = s+i+d$) for a sequence with 50∼100 characters ($l = 50 \sim 100$), our method is much faster than the Needleman-Wunsch algorithm.

EXAMPLE 2.7. *This example shows the steps for an alignment with at most one substitution and one deletion between sequences x and y. This alignment requires a total of 16 bitwise instructions.*

$$x = (10001111010101010110)_2$$
$$y = \ (10101111010110010110)_2$$
$$u_0 = x \oplus y = (11011000111110000000)_2$$
$$u_1 = (x \ll -1) \oplus y = \ (00100000000011011101)_2$$
Case $-1, 0$:
$$v_1 = u_0\&(u_0 - 1) = (11011000111110000000)_2 \quad [\mathbf{R}]$$
$$m_1 = v_1| - v_1 = (11111111111110000000)_2 \quad [\mathbf{S}]$$
$$w_0 = u_1\&m_1 = (00100000000000000000)_2$$
Case $0, -1$:
$$m_1 = u_0| - u_0 = (11111111111100000000)_2 \quad [\mathbf{S}]$$
$$v_1 = u_1\&m_1 = (00100000000000000000)_2$$
$$w_1 = v_1\&(v_1 - 1) = (00000000000000000000)_2 \quad [\mathbf{R}]$$
$$(w_0 = 0)|(w_1 = 0) = \texttt{true}$$

2.5.6 Alignments on the Genome Domain

We extend our bitwise techniques presented above to support alignments on sequences with more than two base types in the domain. Suppose that each character in the domain can be represented by D bits (in our case, $D = 3$ to to encode the symbols `A`, `C`, `G`, `T`, and `N`). As we mentioned before, the techniques described in previous sections cannot be directly applied on the encoded binary strings of genome sequences. For example, given two sequences `GACT` $= (010000001011)_2$ and `GAGT` $= (010000010011)_2$, there are two different bits between the two sequences, because: $(010000001011)_2 \oplus (010000010011)_2 = (000000011000)_2$. With this test, it seems that the sequences match with two errors. However, the sequences can actually be aligned with one character error (C→G).

All our bitwise alignment algorithms discussed in the previous sections (see Sections 2.5.2-2.5.5) begin with one or a few bitwise XOR operations (in the form of $v = (x \gg i) \oplus y$) to identity the different bits between the two sequences. To support alignments on the domain, we add one additional step

$$v = (v \gg D-1|...|v \gg 1|v)\&\mu, \text{where } \mu = (...0^{D-1}10^{D-1}1)_2,$$

immediately after each of these operations to compute a character-difference sequence based on the bit-difference sequence. An OR operation is performed on the D bits representing the same character, to reflect that if any pair of bits is different, then the characters are different. We then mask it by μ to generate a binary sequence in the form: $(...0^{D-1}v_20^{D-1}v_10^{D-1}v_0)_2$, where v_i indicates the inequality between the i-th characters in the two sequences. For example, if we apply this formula on the bit-difference sequence $(000000011000)_2$ in the example shown above ($D = 3$), the resulting sequence is $(000000001000)_2$.

EXAMPLE 2.8. *This example demonstrates an alignment with at most one substitution and one deletion between x and y. The procedure is similar to that of example 2.7, except for the two added lines (shown as underlined).*

$$\begin{aligned}
x &= \texttt{GCTCGAC}\\
y &= \texttt{\ GATCAC}\\
x &= (0100010110010100000 01)_2\\
y &= (\quad 010000011001000001)_2\\
u_0 = x \oplus y &= (0100110110100110000 00)_2\\
\underline{u_0 = (u_0 \gg 2|u_0 \gg 1|u_0)\&\mu} &= (0010010010010010000 00)_2\\
u_1 = (x \ll -3) \oplus y &= (\quad 000001000000010001)_2\\
\underline{u_1 = (u_1 \gg 2|u_1 \gg 1|u_1)\&\mu} &= (\quad 000001000000001001)_2\\
&\textit{Case } -1, 0\textit{:}\\
v_1 = u_0\&(u_0 - 1) &= (0010010010010000000 00)_2\\
m_1 = v_1| - v_1 &= (1111111111110000000 00)_2\\
w_0 = u_1\&m_1 &= (0000000010000000000 00)_2\\
&\textit{Case } 0, -1\textit{:}\\
m_1 = u_0| - u_0 &= (1111111111111110000 00)_2\\
v_1 = u_1\&m_1 &= (0000000010000000000 00)_2\\
w_1 = v_1\&(v_1 - 1) &= (0000000000000000000 00)_2\\
(w_0 = 0)|(w_1 = 0) &= \texttt{true}
\end{aligned}$$

2.6 Compressing Genome Sequences

Some areas of any typical genome are filled with Ns (recall from Section 2.1, the symbol N represent unknown or a repetitive region). For example, within the repeat-masked human genome, about 50.5% of the 3 billion nucleotides are Ns, because entire repetitive regions are masked out as Ns. These Ns can represent any of the nucleotides – A, C, G or T, and are always treated as an error in alignments. This property provides an opportunity to compress the genome sequence by removing unnecessary Ns, and is exploited by the compressor component shown in Figure 1.

A naive way of compression is to directly remove all Ns in the reference sequence. However, this method introduces errors when a query sequence can be aligned to the portions to the immediate left and right of a series of Ns in the reference sequence. The following example shows a wrong alignment on the naive compressed sequence. Query sequence 1 cannot be aligned to the reference sequence, but it can be aligned to the naive compressed sequence.

Reference Seq.	GGCCACAGAANNNNNNTACTACG
Naive Compressed Seq.	GGCCACAGAATACTACG
Query Seq. 1	CCACAGAATACT

To guarantee the correctness of alignments, WHAM uses an accurate compression method to remove unnecessary characters. The length of the compressed sequence is comparable to the naive compressed sequence, but the aligner generates the exact same alignments on the compressed sequence as on the original sequence.

Suppose the alignment constraint is k errors. Then, the portion in the original sequence that consists of at least $k + 1$ consecutive Ns is called *N-series*. WHAM cuts all N-series into $k + 1$ Ns.

Taking the above example again, if we allow 2 errors, then the N-series that consists of 6 Ns is cut to a shorter series with 3 Ns. As shown below, query sequence 1, which cannot be aligned to the original sequence within 2 errors, also cannot be aligned to the compressed sequence within 2 errors (the best alignment is with 3 substitutes, shown by the underlined characters).

Reference Seq.	GGCCACAGAANNNNNNTACTACG
WHAM Compressed Seq.	GGCCACAGAANNNTACTACG
Query Seq. 1	CCACAGAA<u>TAC</u>T

The position of a matched portion in the compressed sequence needs to be mapped to the position in the original sequence when WHAM outputs results. This can be done by building an index on all position pairs of N-series in the original and the compressed sequences. To obtain the original position of a matched portion, we search against the index using its position in the compressed sequence as a search key to find the rightmost N-series that is to the left of the matched portion. The original position is then calculated based on the distance to the N-series and the original position of the N-series. In our implementation, we employ a cache-efficient B-tree [18] to index the position mapping.

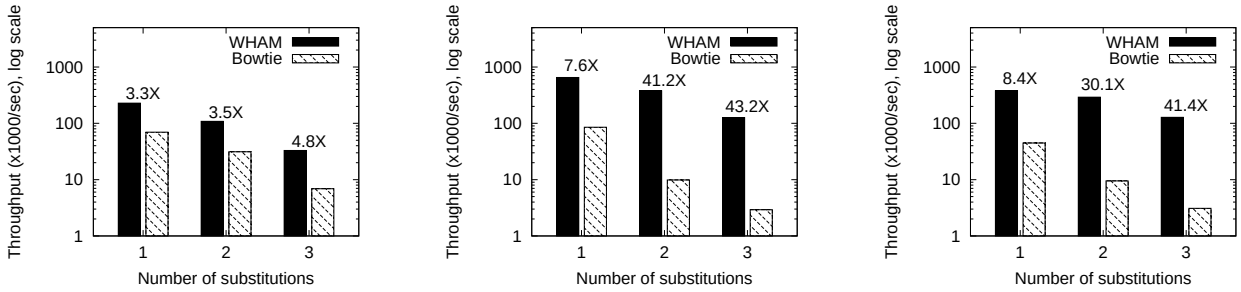
2.7 Analytical Model and Index Optimizer

In this section, we present an analytical model of WHAM's alignment performance. This model is used by the index optimizer shown in Figure 1 to pick an ideal number of fragments (such that performance when probing is maximized).

WHAM's indexing technique is based on a hashing method that is known to have low memory reference locality, which implies that the total execution time is likely to be dominated by the CPU cache stall time. Therefore, we use the number of cache misses as the metric in our model. The model relies on the input parameters including the length of query sequence l, the number of fragments f, the number of errors k, and outputs the estimated number of cache misses for each alignment.

Total alignment cost, M_{align} is simply $N_{probe} \cdot M_{probe}$, where N_{probe} is the number of probes, and has been derived in Lemma 3.

Next, we analyze the cost of a probe on a hash index, M_{probe}. The random variable X is used to represent the number of records that are hashed into a particular bucket slot. When searching records in a hash table, cache misses can be incurred as it accesses the bucket table, the overflow list, as well as the reference sequences in the database. If $X = 0$, the bucket slot is empty, and we only need to access one slot in the bucket table (1 cache miss). When $X = 1$, an extra cache miss is incurred when reading the sequence from the reference sequence following the pointer stored in the bucket slot. When $X = i (i > 1)$, we first access the bucket (1 cache miss), and then scan the overflow list ($\lceil \frac{i}{B} \rceil$ cache misses, where B is the block size of the CPU cache, also known as the cache line size). For each entry in the overflow list, we go to the reference sequence (R) to verify the alignment (i cache misses). By taking the three cases into consideration, the number of cache misses for a probe is



(a) 36bps (b) 60bps (c) 74bps

Figure 5: Throughput comparison between WHAM and Bowtie varying the number of substitutions

given as follows:

$$M_{probe} = P(X=0)+2\cdot P(X=1)+\sum_{i=2}^{n}(1+\lceil\frac{i}{B}\rceil+i)\cdot P(X=i)$$

In a hash table, n records occupy b buckets. We assume that the records have uniform distribution on their range, and the hash function is a perfect hash function. If a record is hashed into a particular hash bucket with a probability of $1/b$, and all n records are hashed, then the number of records in each hash bucket can be modeled as a binomial distribution with parameters n and $\frac{1}{b}$, i.e. $X \sim B(n, \frac{1}{b})$. The probability mass function of X is:

$$P(X=i) = C_i^n(\frac{1}{b})^i(1-\frac{1}{b})^{n-i}$$

The number of occupied buckets, b, depends on the length of concatenations $q = (f-k)\cdot\lfloor\frac{l}{f}\rfloor$. Since the sequence consists of four possible bases for each character, the maximum number of occupied buckets is 4^q. Given the total number of buckets N_{bucket} in the hash table, the number of occupied buckets is:

$$b = \min(4^q, N_{bucket}), \quad \text{where } q = (f-k)\cdot\lfloor\frac{l}{f}\rfloor$$

Using the model presented above, we have implemented an *index optimizer* (see Figure 1) that uses an analytical model *evaluator* to compute the impact of each possible f value on the performance when matching the read set. The *optimizer* (see Figure 1) enumerates the candidate f values, and determines a suitable f value that minimizes the estimated matching cost for a target read length and the error model.

3. EVALUATION

We ran our experiments on a machine with dual 2.67GHz Intel Xeon 6-core CPUs, and 24GB of DDR3 main memory, running Scientific Linux 5 (kernel 2.6.9). Each processor has 12MB of L3 cache shared by all cores on that processor. In addition, each core has a private 32KB L1 instruction and a 32KB L1 data cache, and 256KB of L2 cache. Each processor also employs a two-level hardware TLB. All algorithms were implemented in C++, and compiled using g++ 3.4.6 with optimization flags (O3 and finline-functions).

We chose three sets of *real* query sequences as workloads. These workloads have varying read lengths of 36bps, 60bps, and 74bps, providing for a wide range of read lengths corresponding to what the current next-gen sequencing machines provide today. The 36bps and 74bps workloads come from NCBI, and were chosen because their lengths are similar to the datasets used in the Bowtie paper [6], and the 60bps workload is from our collaborators. Each workload contains about 3 million reads.

Our reference genome (on which we build indices) is the repeat-masked human genome NCBI build 36. We also tested WHAM on the non-masked human genome. After filtering out junk regions that appear more than 100 times in the genome, the performance is similar to that on the masked genome.

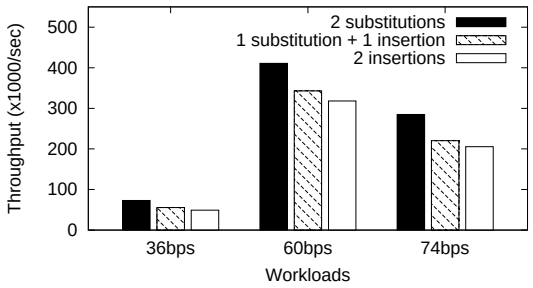


Figure 6: WHAM throughput varying the number of insertions (the number of errors is fixed at two)

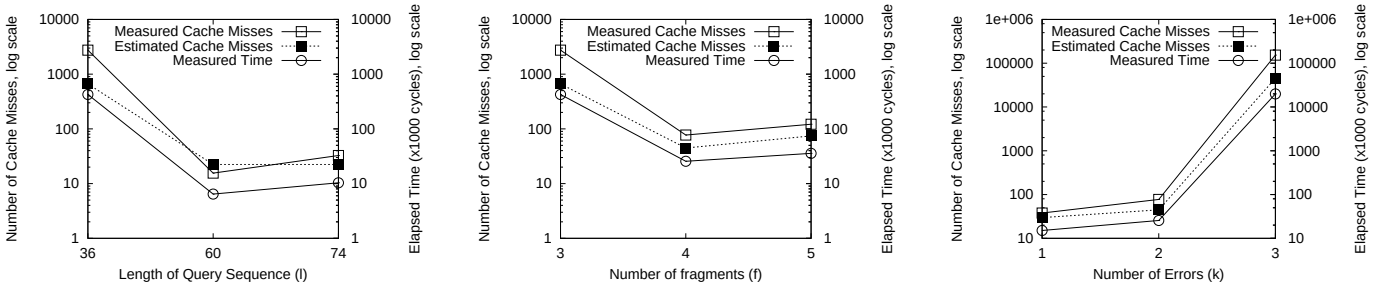
The WHAM index optimizer (see Section 2.7) recommended the number of fragments for each workload, and then computed the number of indices based on Lemma 2 for each workload. Table 2 summarizes the parameters used for the three workloads. The number of fragments plays a key role in the performance of WHAM. For example, if we use 3 fragments to align a 36bps workload with 2 errors, it is one order of magnitude slower than using the recommended value. All values that we used in the experiments are recommended by the *index optimizer*, whose accuracy is evaluated in Section 3.3.

Table 2: Number of fragments and indices for the workloads

	Number of fragments			Number of indices		
	1 err	2 err	3 err	1 err	2 err	3 err
36bps	3	4	6	2	3	10
60bps	2	3	5	1	1	4
74bps	2	3	4	1	1	1

Each WHAM index is about 8.9GB in size and takes about 20 minutes to build. Loading a prebuilt WHAM index from disk takes less than two minutes. The number of indices for each workload is listed in Table 2. If the total size of indices exceeds the memory size, then WHAM loads indices and performs alignments on the loaded indices one after another.

In the evaluation below, we compare WHAM to Bowtie [6]. Bowtie also builds an index on the reference genome, and the Bowtie index is about 1.5GB and takes about 3 hours to build. We have also compared WHAM with RBSA [17], a string matching method that accommodates a broader class of error models compared to WHAM. However, our experiment results show that WHAM is several orders of magnitudes faster than RBSA. In addition, RBSA has a much larger memory footprint than WHAM making it infea-



(a) f =3, k=2 (b) l =36, k=2 (c) l =36, f=4

Figure 7: Analytical Model Validation

sible to use on our machines for the whole human genome. Consequently, we do not consider RBSA further in this paper.

Since index construction and loading is a one-time cost, in the performance comparison, we do not include these costs for either Bowtie or WHAM.

3.1 Comparison between WHAM and Bowtie

We first compare the throughput of WHAM with that of Bowtie (version 0.12.3), the leading state-of-the-art method that is widely deployed in production settings today. We used the 64-bit version of Bowtie running with the `-a` flag, indicating that it should report all valid matches. Since Bowtie does not support indels, we only compare the throughput with varying number of substitutions.

Figure 5 shows the results on the three datasets. The speedup of WHAM over Bowtie is also marked on the top of the bars in the graph. As can be observed from Figure 5, WHAM is uniformly better, with speedup of 3X to 5X for the 36bps workload, and 8X to 43X for the 60bps and 74bps workloads. The speedup difference between the short (36bps) and the long (60bps and 74bps) query sequences is due to the hash collisions. When the query sequence is short, many of the concatenations (keys) are the same, which results in many collisions in the hash tables. Consequently, throughput drops significantly as either WHAM suffers from scanning the long overflow array on a hash probe, or it breaks the query sequences into more fragments, and thus increases the number of probes.

For the short query sequence (36bps), the throughput of both WHAM and Bowtie degrades as the number of substitutions increases (see Figure 5 (a)). The speedup of WHAM over Bowtie is relatively steady, within a small range from 3.3X to 4.8X. This means that the throughput of WHAM decreases as fast as that of Bowtie when allowing for more substitutions. This behavior is due to the combinations of two effects. First, WHAM performs more probes on the hash table as the number of allowed errors increases. Second, and more importantly, the concatenations become shorter and cause significant collisions in the hash tables when allowing for more errors. As a result, the overflow arrays become longer, and the hash probes become more expensive.

For longer query sequences, the speedup of WHAM over Bowtie increases as the number of substitutions increases. For the 60bps workload (Figure 5 (b)), WHAM is 7.6X, 41.2X, and 43.2X faster with 1, 2, and 3 substitutions respectively. For the 74bps workload (Figure 5 (c)), the speedup is 8.4X, 30.1X, and 41.4X with 1, 2, and 3 substitutions respectively. (Since the 60bps and the 74bps datasets are different workloads, the reader should not read too much into variations in performance for the specific number of substitutions across these two datasets.) For the longer query sequences, the indexed fragments in WHAM are long enough to mitigate the hash collision issue. WHAM's throughput degradation is almost always due to the increasing number of probes on the hash table and the number of errors increases. As a result, the throughput of WHAM degrades slower than that of Bowtie as the number of errors in the match model increases.

3.2 Effect of Indels

WHAM also supports indels as well as the combinations of substitutions and indels. In this experiment, we fix the number of errors at two, and vary the number of insertions (the number of substitutions is two minus the number of insertions). Since Bowtie does not support indels, we cannot make a comparison with Bowtie.

Figure 6 shows the throughput for the three combinations of substitutions and insertions. The throughput degrades as the number of insertions increases across the three datasets. This is due to two effects. First, supporting more insertions introduces more probes on the hash indices. Second, the bitwise alignments with insertions are typically more complex than those with only substitutions, and as a result, the cost of the bitwise alignment method increases with increasing indels in the model.

3.3 Model Validation

Finally, we evaluate the accuracy of our analytical model by comparing the estimated and measured performance with various parameters. Figure 7 demonstrates the measured and estimated number of cache misses, as well as the measured elapsed time, varying the parameters l, k, f. The number of cache misses is obtained by enabling the hardware performance-monitoring counters (RDPMC instruction), and only the number for the lowest level data cache is reported.

In each experiment, we fix two parameters and vary the third one. Figure 7(a) shows the measured and estimated performance on each of our three workloads. Figure 7(b) plots the performance when f varies from 3 to 5. Figure 7(c) illustrates the performance when varying the value of k from 1 to 3.

As shown in Figure 7, our estimates show a trend similar to the measured values. The gap between the measured and estimated cache misses is mainly due to the different alignment ratio in the datasets. Note that an invalid query sequence is more likely to access the bucket that has a short overflow list or an empty bucket, whereas a valid query sequence is more likely to access the "hot" bucket. Thus, more cache misses occur in the dataset with a high alignment ratio. For example, the 60bps dataset has a low alignment ratio, whereas the 74bps dataset has a high alignment ratio. As shown in Figure 7(a), our analytical model overestimates on the 60bps dataset, and underestimates on the 74bps dataset.

Furthermore, the estimated best f values match the measured best values. As shown in Figure 7(b), the analytical model accurately estimates that the best performance is achieved when partitioning the query sequence into 4 fragments. We used our ana-

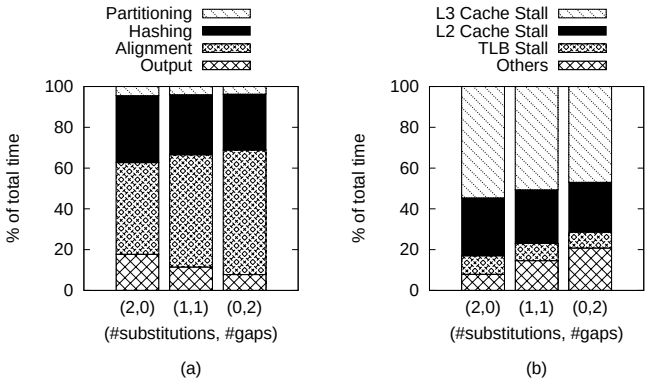


Figure 8: Time breakdown varying the number of substitutions for the 60bps workload

lytical model in our index optimizer to recommend the f value in all our experiments, and the resulting recommended index settings are shown in Table 2. By comparing WHAM's histogram in Figure 5 and the best throughput among all partitioning schemes (the results are omitted here in the interest of space), we observe that the analytical model used in the index optimizer captures the best parameters for all settings in our experiments.

3.4 Time breakdown

To better understand the performance characteristics of WHAM, next we examine the detailed time breakdown for 1) the different phases in WHAM, and 2) the different processor stall times when running WHAM.

The time breakdown for the different phases of WHAM is shown in Figure 8 (a). This time breakdown is for the 60bps workload and for three error models: 2 substitutions, 1 substitution and 1 insertion, and 2 insertions. The time breakdown for the other workloads and error models are similar to the one shown in Figure 8 (a), and we omit these other results in the interest of space.

In Figure 8 (a), the four main phases include: (a) the partitioning phase to partition the short reads into fragments and to combine these fragments into concatenations if necessary, (b) the hashing phase to perform lookups on the hash table, and to scan the corresponding overflow array, (c) the alignment phase to perform bitwise alignment between a pair of sequences, and (d) the output phase to write the results to a file. The partition phase is the fastest phase, and only accounts for about 4% of total time. The hashing time is significant (about 30%) across all three settings, due to cache misses (stalls) when accessing hash entries. Alignment is the most expensive phase. It varies from 45% to 60% of the total time. The alignments with more deletions execute more instructions and are more expensive. The output phase accounts for 8%–18% of the total time, depending on the number of successful alignments.

Figure 8 (b) illustrates the time breakdown for different processor stall times. We used the processor hardware performance counters to measure the number of L3 cache misses, L2 cache misses, and TLB misses. Since these resource stalls can overlap with useful computation in modern processors, there is no real way to measure the overlap between a stall and an instruction executing in the instruction stream, as the overlapped time depends on the degree of instruction-level parallelism. We crudely (over) estimate the stall time as the number of measured misses multiplied by the cycles for each stall. The "other" time in this figure is computed by subtracting the L2, the L3, and the TLB stall times from the total time. All results are with a single threaded execution of WHAM. Multi-threaded results are presented below in Section 3.5.

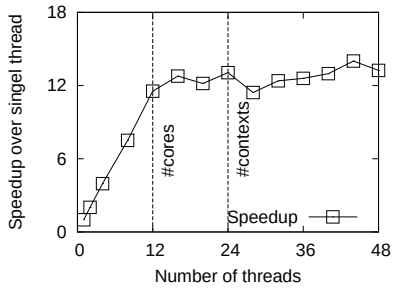


Figure 9: WHAM throughput varying number of threads

As shown in Figure 8 (b), the L3 and the L2 cache stall times account for about 50-60%, and 25-30% of the total time respectively, and are the dominating stall costs in WHAM. This empirically verifies the assumption in Section 2.7 that cache stall time is the performance bottleneck. When we allow more gaps, the "other" time increases, mainly due to more instructions involved in computing pairwise alignments.

3.5 Performance on Multi-Core CPUs

All the previous results are with WHAM running in a single thread. We have also implemented a multi-threaded version of WHAM, and examined its performance on a multi-core processor. In this setting, the batched short reads are evenly distributed across all running threads. Each thread independently lookups the shared hash indices and performs the pairwise alignment in parallel. Figure 9 illustrates the throughput speedup of the multi-threaded execution over the single threaded execution with varying number of threads.

As shown in the figure, the throughput has a linear speedup as long as the number of threads does not exceed twelve, which is the total number of physical cores in our test machine. The linear speedup indicates that, although the cache stall time dominates the execution time of a single thread execution, it does not become the performance bottleneck when using multiple cores, as memory access requests from different threads can be issued and processed in parallel. The architecture of the machine that we are using provides a higher bandwidth between the main memory and the CPU caches than what WHAM demands. However, the speedup has only a slight increase when the number of threads exceeds twelve, which means that WHAM does not benefit from Hyper-Threading. The throughput fluctuates when the number of threads is more than the number of physical cores, due to the cost of context switching when scheduling the threads across the processing cores.

3.6 Summary

The main conclusions from these experiments are that, (a) the analytical model which is at the heart of the index optimizing component of WHAM is accurate, (b) WHAM is often more than an order of magnitude faster than Bowtie on long read workloads, (c) WHAM performs much better on long read sequences than short ones, which bodes extremely well for the future in which read lengths are expected to increase, (d) WHAM supports indels with low overhead, and (e) WHAM exhibits a linear speedup as the number of threads increases to match the number of processing cores.

4. RELATED WORK

A number of commonly used alignment algorithms employ the dynamic programming technique, including the Needleman-Wunsch algorithm [16] (for global alignment) and the Smith-Waterman algorithm [20] (for local alignment).

Several q-gram based methods have been developed for approximate string matching in large sequence databases [2, 11, 14, 23]. A q-gram is a substring of length q. These q-gram based methods are based on counting the number of q-grams that are shared between the query and the reference sequences. Although these q-gram based methods and WHAM find approximately matched sequences by performing exact searches on subsequences, the key differences stem from the fact that q-gram based methods count the number of shared subsequences, whereas WHAM uses subsequences as seeds to find valid matching sequences. Another difference is that the q-grams are overlapping subsequences, whereas in WHAM fragments are non-overlapping subsequences.

Bit manipulation methods such as the Shift-Or algorithm [5] and its extended variant [22] have been used to efficiently search for a query sequence in a reference sequence. Unlike the Shift-Or algorithm and its variants, WHAM's bitwise alignment algorithms are used to globally align the two sequences along their entire lengths, with an arbitrary number of mismatches and gaps.

BLAST [3] is a popular heuristic tool to compute local alignments for long query sequences. However, a major limitation of BLAST is that there is no guarantee that the optimal local alignment will be reported (since it uses a heuristic technique). More recently, a reference-based method, called RBSA [17], was proposed for large queries and accurate results. In RBSA, query sequences are aligned to a group of references with precomputed alignment scores, instead of being aligned to the original reference sequence directly. These methods that support long query sequences and arbitrary error models are more general than WHAM, but are not as efficient for aligning short reads against genomes.

Previous methods for sequencing short reads include Maq [12], SOAP [13], and Bowtie [6]. In particular, Bowtie has focused on utilizing the properties of the Burrows-Wheeler Transforms (BWT) [7] to index the reference sequence. A BWT-based index can be viewed as a suffix tree variant, and has a small memory footprint, making Bowtie feasible on computers with only 2GB of memory. Our technique differs significantly from Bowtie and other suffix-tree based indexes, as we build hash indexes on subsequences of the reference sequence. Although a hash structure does not inherently allow for the same ease of discovering strings with different numbers of mismatches, it does take advantage of current memory sizes to produce an overall faster alignment method.

The WHAM indexing schemes are also related to PartEnum algorithm [4] proposed for set similarity joins, which uses binary vectors to represent sets, and computes the Hamming distance between the binary vectors. WHAM and PartEnum use a similar idea: partitioning, and enumeration on partitions of vectors. However, WHAM differs from PartEnum in a few key ways. First, PartEnum is designed for set similarity joins based on Hamming distance, whereas our method is used for biological sequence alignments based on edit distance. Second, WHAM builds hash-based indexes on the basis of categories of concatenations for large reference sequences. WHAM selects indexing schemes that trade off between the number of lookups on indexes and the performance of each lookup, to achieve optimal overall performance.

5. CONCLUSION AND FUTURE WORK

With the advances in sequencing technology, there is an urgent need for a fast read alignment method that can deal with longer reads and accommodate rich error models. This paper proposes a method called WHAM that addresses this need. WHAM employs novel hash-based indexing and bitwise operations for pairwise alignments. Our extensive experimental studies using real datasets show that, compared to the existing leading read alignment method, WHAM is often faster by an order-of-magnitude (or more) and can accommodate richer error models. In addition, WHAM also leverages multi-core architectures very effectively. For future work, we plan to extend WHAM to work in distributed data processing environments.

Acknowledgments

We would like to thank Ron Steward and Victor Ruotti for various discussion on the topic of next-gen sequence alignment. We would also like to thank Spyros Blanas and the reviewers of this paper for valuable feedback on an earlier draft of this paper. This work was supported in part by a grant from the National Science Foundation under grant DBI-0926269.

6. REFERENCES

[1] White House Press Release. Retrieved 2006-07-22. http://www.ornl.gov/sci/techresources/Human-Genome/project/clinton1.shtml.

[2] M.-S. K. 0002, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[5] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[6] L. Ben, T. Cole, P. Mihai, and S. Steven. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[7] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report*, 1994.

[8] T. Han, S. Ko, and J. Kang. Efficient Subsequence Matching Using the Longest Common Subsequence with a Dual Match Index. *Machine Learning and Data Mining in Pattern Recognition*, pages 585–600, 2007.

[9] E. Karakoc, Z. Ozsoyoglu, S. Sahinalp, M. Tasan, and X. Zhang. Novel approaches to biomolecular sequence indexing. *Data Engineering*, 1001:40, 2004.

[10] D. E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, Jan. 2011.

[11] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[12] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851, 2008.

[13] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713, 2008.

[14] W. Litwin, R. Mokadem, P. Rigaux, and T. J. E. Schwarz. Fast ngram-based string search over data encoded using algebraic signatures. In *VLDB*, pages 207–218, 2007.

[15] J. D. McPherson. Next-generation gap. *Nature Methods*, 6(11s):S2–S5, October 2009.

[16] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[17] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos. Reference-based alignment in large sequence databases. *PVLDB*, 2(1):205–216, 2009.

[18] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[19] R. L. Rivest. Partial-match retrieval algorithms. *SIAM J. Comput.*, 5(1):19–50, 1976.

[20] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[21] Venter, et al. The Sequence of the Human Genome. *Science*, 291(5507):1304–1351, 2001.

[22] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[23] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, pages 353–364, 2008.