A Memory-Efficient FM-Index Constructor for Next-Generation Sequencing Applications on FPGAs

Nae-Chyun Chen, Yu-Cheng Li and Yi-Chang Lu
Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan, 10617
Email: r04943093@ntu.edu.tw, d01943008@ntu.edu.tw, yiclu@ntu.edu.tw

***Abstract*—FM-index is an efficient data structure for string search and is widely used in next-generation sequencing (NGS) applications such as sequence alignment and *de novo* assembly. Recently, FM-indexing is even performed down to the read level, raising a demand of an efficient algorithm for FM-index construction. In this work, we propose a hardware-compatible Self-Aided Incremental Indexing (SAII) algorithm and its hardware architecture. This novel algorithm builds FM-index with no memory overhead, and the hardware system for realizing the algorithm can be very compact. Parallel architecture and a special prefetch controller is designed to enhance computational efficiency. An SAII-based FM-index constructor is implemented on an Altera Stratix V FPGA board. The presented constructor can support DNA sequences of sizes up to 131,072-bp, which is enough for small-scale references and reads obtained from current major platforms. Because the proposed constructor needs very few hardware resource, it can be easily integrated into different hardware accelerators designed for FM-index-based applications.**

***Keywords*-next-generation sequencing, FM-index, Burrows-Wheeler transform, self-aided index construction**

I. Introduction

Burrows-Wheeler transform (BWT) [1] is a string rearrangement algorithm first proposed for data compression. Making use of the properties between BWT and its original string, FM-index [2] is designed for efficient string searching. For a certain string, the data structure of its FM-index contains the BWT, the suffix array (SA) and two auxiliary tables of the target string. With its high efficiency in both time and memory, FM-index is widely adopted by many next-generation sequencing (NGS) applications [3], [4], [5], [6].

For FM-index-based sequence aligners such as BWA-backtrack [3] and Bowtie2 [5], the reference sequence is indexed for fast read-locating processes. For this kind of aligners, because only the reference sequence needs to be indexed, the hardware accelerators ([7], [8]) usually build the FM-index externally using CPUs. Since the length of a reference sequence can be as large as three billion base pairs (bps), the memory usage of naive index constructing method is unaffordable. Therefore, how to reduce memory usage in FM-indexing becomes an important issue [9], [10].

Recently, some *de novo* assemblers ([6], for example) apply FM-index for overlap finding of reads. Also, the reference and reads are both indexed in new sequence aligners such as BWA-SW [4]. For these applications, external construction of FM-index has much higher time overhead in comparison with traditional algorithms. Therefore, on-chip indexing becomes necessary and important. Our previous research [11] has demonstrated an FM-index constructor with a lightweight iterative algorithm [10] with an ASIC, but the cost of the indexer is still high when compared to the work proposed here.

In this work, we propose a novel memory efficient FM-index construction algorithm, Self-Aided Incremental Indexing (SAII), which is suitable for hardware realization. This algorithm builds the FM-index incrementally. In each iteration, it utilizes a meta-index to construct the complete index. Since SAII makes use of the FM-index itself for construction, it has no memory overhead for FM-index-based applications. Only few computational logic units are needed. In our hardware system, the processing speed is accelerated by a special prefetch mechanism and a parallel architecture. We choose Altera Stratix V FPGA as the evaluation platform. The SAII FM-index constructor is very compact in terms of logic usage, so it can be integrated with other functional blocks to form a complete hardware pipeline in emerging NGS applications.

II. Background

A. Burrows-Wheeler Transform

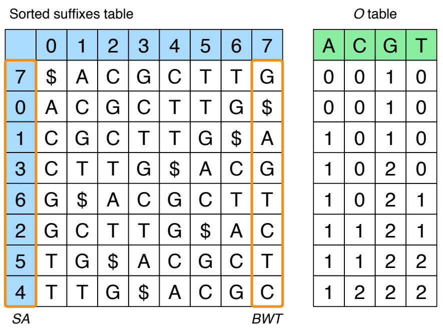
To construct the BWT of target sequence X, a simple method is via the translation of suffix array (SA) with Eq. 1. Also, BWT can be obtained by collecting all characters in the last column of sorted suffixes. Fig. 1 shows an example of the BWT of sequence X=ACGATTG\$, where character \$ is the end-of-string character. The lexical order is \$<A<C<G<T.

$$BWT[i] = \begin{cases} X[SA[i]-1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases} \tag{1}$$

B. FM-index and Backward Search Algorithm

FM-index is extended from BWT and suffix array, with two auxiliary tables—C array and O table. The definitions for C array and O table are shown in Eq. 2 and 3, where n is the length of target sequence X.

$$C(a) \equiv size\{0 \le j \le n-2 : X[j] < a\} \tag{2}$$



Sorted suffixes table

SA	0	1	2	3	4	5	6	7
7	$	A	C	G	C	T	T	G
0	A	C	G	C	T	T	G	$
1	C	G	C	T	T	G	$	A
3	C	T	T	G	$	A	C	G
6	G	$	A	C	G	C	T	T
2	G	C	T	T	G	$	A	C
5	T	G	$	A	C	G	C	T
4	T	T	G	$	A	C	G	C

SA BWT

O table

A	C	G	T
0	0	1	0
0	0	1	0
1	0	1	0
1	0	2	0
1	0	2	1
1	1	2	1
1	1	2	2
1	2	2	2

C array

A	C	G	T
0	1	3	5

Fig. 1. The FM-index of target string ACGCTTG$. This data structure includes SA, BWT, C array and O table. BWT is the last column of the sorted suffixes table.

$$O(a, i) \equiv \begin{cases} size\{0 \leq j \leq i : BWT[j] = a\}, & i \geq 0. \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

With C array and O table, the position of query aW can be efficiently located within a lower bound $\underline{R}(aW)$ and upper bound $\overline{R}(aW)$ as shown in Eq. 4 and 5.

$$\underline{R}(aW) = C(a) + O(a, \underline{R}(W) - 1) + 1 \tag{4}$$

$$\overline{R}(aW) = C(a) + O(a, \overline{R}(W)) \tag{5}$$

In [2], Ferragina and Manzini have proved that $\underline{R}(aW) \leq \overline{R}(aW)$ if and only if aW is a substring of X. This searching algorithm starts from the end of the query sequence and extends iteratively. Therefore it is also called backward search algorithm.

III. Self-Aided Incremental Indexing (SAII) Algorithm

An example of backward search algorithm is shown in Fig. 2. The initial values $\{\underline{R}(\emptyset), \overline{R}(\emptyset)\}$ are set at $\{0, n-1\}$. Here we discuss the mathematical insights of the lower bound in backward search algorithm. $\underline{R}(aW)$ is the sum of $C(a)$, $O(a, \underline{R}(W) - 1)$ and 1. $C(a)$ records the total number of characters lexically smaller than a in target sequence X. The $O(a, \underline{R}(W)-1)$ term is the occurrence of aW in X. With an additional offset, $\underline{R}(aW)$ represents the suffix array index of lexically smallest aW sequence. Similar concept can be used to account for the upper bound. If aW only occurs once in X, $\underline{R}(aW)$ is equal to $\overline{R}(aW)$. It is also of interest that what would happen if aW is not a substring of X. Since $\underline{R}(aW)$ measures the occurrences of lexically smaller substrings in X, the lower bound guarantees the following inequality:

$$suffix_{SA[\underline{R}(aW)-1]} < aW < suffix_{SA[\underline{R}(aW)]} \tag{6}$$

SAII utilizes Eq. 6 to build FM-index incrementally. For a target sequence X, if the FM-index of its substring $L = a_i a_{i+1} ... a_{n-2}\$$ has been constructed, the suffix array index of the query string $a_{i-1}L = a_{i-1}a_i a_{i+1}...a_{n-2}\$$ can be determined by calculating $\underline{R}(a_{i-1}L)$. Since $a_{i-1}L$ could not be a substring of L, the suffix array index obtained is unique

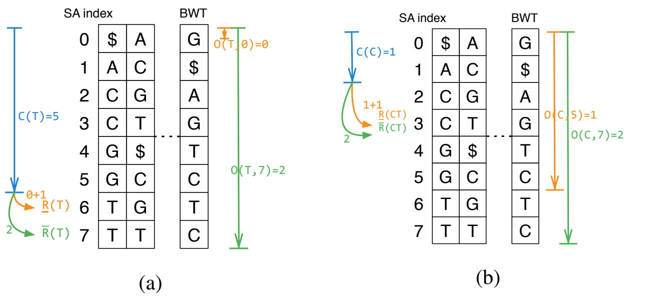


Fig. 2. An example of searching query sequence CT on indexed target string X=ACGCTTG$. (a) and (b) show the first and second iteration respectively.

and follows Eq. 6. Therefore, we can insert a_{i-1} into the FM-index of L, generating the new FM-index of $a_{i-1}L$ without sorting the whole string all over again.

The algorithm of SAII is shown in Alg. 1, and an example of a target sequence ACGCT$ is provided in Fig. 3. In the first iteration, the initial character is $ and the BWT is also $. The corresponding O table and $\underline{R}$ are calculated. In the second iteration, a new character T is added to the target sequence. Then we use O and C obtained in the previous iteration to calculate the new $\underline{R}$, and the updated $ is inserted to this $\underline{R}$ position to form a new BWT. With this updated BWT, we recalculate O and C for this iteration. Then SAII is ready to move on to next iteration. After all the characters in the target sequence are read, the corresponding FM-index of the reference is correctly built.

Because the construction process is entirely based on FM-index itself, nearly no extra computational resources are needed and the memory overhead is zero for FM-index-based applications. The time complexity of SAII algorithm is $O(n^2 k^{-1})$, where k is completeness of the O table. The details of k is given in Sec. IV-B.

Algorithm 1 Self-Aided Incremental Indexing Algorithm

```
Require: target sequence $X$
Ensure: $BWT$, $C$ array and $O$ table
 1: Initialize $C$ array and $O$ table
 2: $n \leftarrow$ length $(X)$
 3: $BWT \leftarrow \$$
 4: $q \leftarrow 0$
 5: for $i$ from $n-2$ to 0 do
 6:     $BWT[q] \leftarrow X[i]$
 7:     $L \leftarrow X[i : n-1]$
 8:     $q \leftarrow C(X[i]) + O(X[i], \underline{R}(L) - 1) + 1$
 9:     $BWT \leftarrow BWT[0 : q-1] + \$ + BWT[q : n-1]$
10:     Update $C$ array with $L$
11:     Update $O$ table with $BWT$
12: end for
```

IV. Hardware Implementation and Discussion

A. Overall Architecture

The hardware system includes a finite-state machine controller and a combinational computing logic. The finite-state

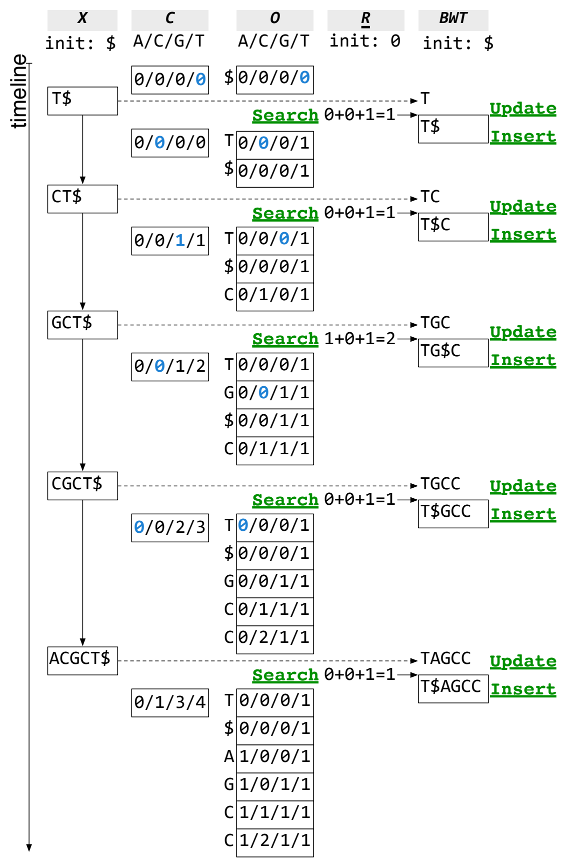


Fig. 3. An example of the FM-indexing of a target sequence ACGCT$ with SAII algorithm.

machine of the proposed hardware is shown in Fig. 4. There are four states in this system: Initial, Search, Update and Finish states.

Initial and Finish states control the initial and finishing conditions of the hardware. Search state computes the lower bound $\underline{R}(aW)$ with Eq. 4. A two-stage parallel pop counter is used in Search state for fast computing. Update State updates the latest incoming symbol to the \$ position in previous iteration. Also, it inserts the \$ symbol to the updated index based on the position calculated in Search state. The whole FM-index data structure including BWT, C array and O table all need to be updated in this state. The finite-state machine repeats between Search and Update states to construct the FM-index and moves to Finish state after the last character of the target sequence is processed.

B. Data Structure

For DNA aligners, the size of alphabets (Σ) is five (including \$). Since symbol \$ occurs only once, in our hardware system only A,C,G, and T are encoded. End-of-string character \$ is encoded the same as the symbol, A, and an additional special pointer is designed to store the position of \$. This design uses only two bits for each character, which is only 67% in comparison to that of naive 3-bit encoding.

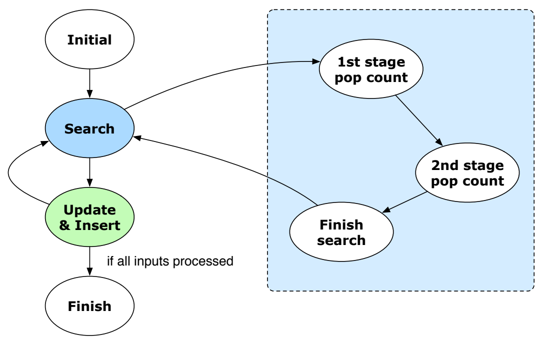


Fig. 4. The finite-state machine controlling the hardware system.

The memory usage of the three main components of FM-index, C array, O table and BWT, are $4 \log n$, $4n \log n$ and $2n$, respectively. However, the length of genome sequence data is sometimes very large. The length of a human chromosome can exceed 200 Mbp and the whole human genome is more than 3 Gbp. With this scale of data, the $4n \log n$ memory usage of a complete O table is very expensive in hardware systems. It should be noted that O table is a hash table obtained from BWT designed for fast computation of $\underline{R}$ and $\overline{R}$. The correct bounds can still be calculated even without O table at the cost of searching efficiency.

In our hardware system, incomplete O table is used to achieve balance between memory usage and computing speed. An incomplete O table stores the occurrence values at every k characters. It is k times smaller than a complete table. With an incomplete O table, the calculation of $O(a, i)$ is split into two parts. First, $O(a, k\left\lfloor \frac{i}{k} \right\rfloor)$ is stored in the incomplete table. Second, the occurrences of a from $k\left\lfloor \frac{i}{k} \right\rfloor$ to i is calculated with a pop counter. In our hardware implementation, k is set at 2,048. The pop counter is designed with a two-stage architecture, in which the first stage has 32 parallel adders and the second stage has 64 parallel adders. The incomplete O table can be adjusted for different applications with simple modification of parameters.

C. Constructing BWT with Prefetch Mechanism

In our hardware system, Search and Update states are in charge of the construction of FM-index. To save computing resource, BWT and O table are both segmented and stored in the BRAM. Though this saves lots of area, it takes longer time to update the BRAM due to the fixed word length. Therefore, how to make use of the limited bandwidth is very important. To address this issue, we design a prefetch mechanism that saves 50% time of BRAM updating. The timing diagram without prefetching is shown in Fig. 5(a). In Search state, $\underline{R}$ is calculated; in Update state, the incoming character is updated to the FM-index as shown in Line 6 in Alg. 1; in Insert state, the \$ is inserted to the FM-index, as shown in Line 9 in Alg. 1. In both Update and Insert states, BWT and O table in the BRAM have to be refreshed, so the processing time is long.

Prefetch mechanism is designed to reduce the runtime to

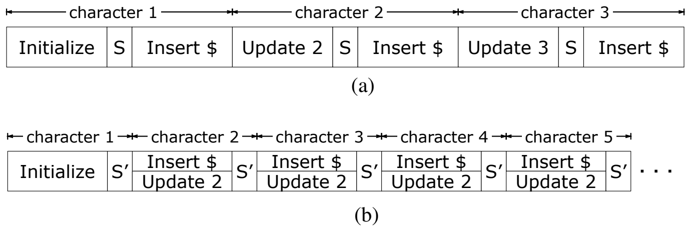


Fig. 5. Timing diagrams for: (a) constructing BWT without prefetch mechanism, and (b) constructing BWT with prefetch mechanism. The blue boxes (S- and S'-boxes) denote Search state. The S'-boxes contains the additional monitor.

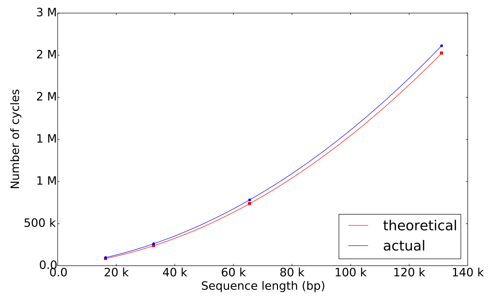


Fig. 6. Processing cycle count of SAII hardware system.

refresh BRAM. As shown in Fig. 3, SAII algorithm first replaces the $ in BWT with the new character, calculates the new insertion position of $ and inserts new $ to the BWT. With prefetch mechanism, the insertion of $ is combined with next Update state and executed after the hardware system sees the next character. This does not generate the completely correct FM-index yet because of its early update design. Therefore, the early updated position and character have to tracked with an additional monitor to make sure the calculation of $\underline{R}$ is correct in the next iteration. In the last iteration, since there is no more character for prefetching, the final FM-index is correct. Fig. 5(b) shows the timing diagram of the hardware system with prefetching.

Assume that one update takes q cycles and one backward search takes m cycles ($q >> m$), prefetch mechanism reduces the computing time from $2q + m$ to $q + m$ for each iteration. The computing time is nearly half of the original design. The expected runtime ($\mathcal{T}$) of the SAII hardware system is shown in Eq. 7, where m stands for search time and is set to 3 cycles in our implementation.

$$\mathcal{T} = k \sum_{i=1}^{\frac{n}{k}} (m + \frac{i}{2}) \tag{7}$$

D. Discussion

We implement our SAII FM-index constructor on an Altera Stratix V FPGA (5SGXEA7N2F45C2N) board. The hardware system is synthesized using Altera Quartus (v.15.0) tool. It only uses 21,944 ALMs (9%) and 266,496 BRAMs (< 1%) on this FPGA, which means it can be integrated into existing sequencing pipelines at very low hardware cost. The operation frequency can reach 120 MHz even with the worst case model (900 mV, 85 °C). Sequences with different lengths, from 16,384-bp to 131,072-bp, are tested and the results are shown in Fig. 6. It takes about 21 ms to finish the indexing of a 131,072-bp sequence. The runtime is very close to our theoretical estimation given in Eq. 7.

V. Conclusions

With many emerging applications based on FM-index, an efficient index construction algorithm is needed. Previous algorithms ([9], [10]) need additional working space to build the index, raising the costs of hardware systems. In this paper, we propose a novel hardware-compatible Self-Aided Incremental Indexing (SAII) algorithm to construct FM-index with no memory overhead. This algorithm is accelerated with a parallel pop counter and a special prefetch mechanism. Its realization on FPGA needs very few hardware resources and can be easily integrated in different FM-index-based applications.

Acknowledgment

This work is supported by the Ministry of Science and Technology, Taiwan, under Grant numbers MOST 105-2221-E-002-090 and 106-2221-E-002-055. Nae-Chyun Chen would like to thank NOVATEK for providing fellowship.

References

[1] M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” 1994.

[2] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

[3] H. Li and R. Durbin, “Fast and accurate short read alignment with burrows–wheeler transform,” *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[4] ——, “Fast and accurate long-read alignment with burrows–wheeler transform,” *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[5] B. Langmead and S. L. Salzberg, “Fast gapped-read alignment with bowtie 2,” *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[6] J. T. Simpson and R. Durbin, “Efficient construction of an assembly string graph using the fm-index,” *Bioinformatics*, vol. 26, no. 12, pp. i367–i373, 2010.

[7] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, “Hardware acceleration of short read mapping,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 161–168.

[8] H. M. Waidyasooriya and M. Hariyama, “Hardware-acceleration of short-read alignment based on the burrows-wheeler transform,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1358–1372, 2016.

[9] D. Okanohara and K. Sadakane, “A linear-time burrows-wheeler transform using induced sorting.” in *SPIRE*, vol. 5721. Springer, 2009, pp. 90–101.

[10] P. Ferragina, T. Gagie, and G. Manzini, “Lightweight data indexing and compression in external memory,” *Algorithmica*, vol. 63, no. 3, pp. 707–730, 2012.

[11] N.-C. Chen, T.-Y. Chiu, Y.-C. Li, Y.-C. Chien, and Y.-C. Lu, “Power efficient special processor design for burrows-wheeler-transform-based short read sequence alignment,” in *Biomedical Circuits and Systems Conference (BioCAS), 2015 IEEE*. IEEE, 2015, pp. 1–4.