Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures

Jing Zhang[1], Heshan Lin[1], Pavan Balaji[2], and Wu-chun Feng[1]
[1]Dept. of Computer Science, Virginia Tech, {zjing14,hlin2,feng}@cs.vt.edu
[2]Argonne National Laboratory, balaji@mcs.anl.gov

***Abstract*—Computational biology sequence alignment tools using the Burrows-Wheeler Transform (BWT) are widely used in next-generation sequencing (NGS) analysis. However, despite extensive optimization efforts, the performance of these tools still cannot keep up with the explosive growth of sequencing data. Through an in-depth performance analysis of BWA, a popular BWT-based aligner on multicore architectures, we demonstrate that such tools are limited by memory bandwidth due to their irregular memory access patterns. We then propose a locality-aware implementation of BWA that aims at optimizing its performance by better exploiting the caching mechanisms of modern multicore processors. Experimental results show that our improved BWA implementation can reduce last-level cache (LLC) misses by 30% and translation lookaside buffer (TLB) misses by 20%, resulting in up to 2.6-fold speedup over the original BWA implementation.**

***Keywords*-short-read mapping, Burrows-Wheeler transform, FM-index, bioinformatics, data locality, multicore**

I. Introduction

Recently, next-generation sequencing (NGS) technologies have dramatically reduced the cost and time of DNA sequencing, making possible a new era of medical breakthroughs based on personal genome information. A fundamental task in human genome sequencing is mapping short DNA sequences, also called reads, that are generated by NGS sequencers, to the reference human genome. Typical human genome data consists of millions of reads and can take up to hundreds of gigabytes. Designing efficient alignment algorithms that can rapidly map large amounts of genome data is an important topic in NGS bioinformatics.

Many short-read alignment tools based on different indexing techniques have been developed during the past couple of years [9]. Among them, alignment tools based on the Burrows-Wheeler Transform (BWT), such as BWA [8], SOAPv2 [10], and Bowtie [7] have become increasingly popular because of their superior memory efficiency and support of flexible seed lengths. The Burrows-Wheeler Transform is a string compression technique that is used in compression tools such as bzip2. Using the FM-index [4], a data structure built atop the BWT, BWT-based alignment tools allow fast mapping of short DNA sequences against reference genomes with a small memory footprint.

State-of-the-art BWT-based alignment tools are well engineered and highly efficient. However, the performance of these tools still cannot keep up with the explosive growth of NGS data. In this paper, we first perform in-depth performance analysis of BWA, one of the most widely BWT-based aligners, on modern multicore processors. As a proof of concept, our study focuses on the exact matching kernel of BWA, because inexact matching is typically transformed into exact matching in BWT-based alignment. Our investigation shows that memory bandwidth is the major performance bottleneck of BWA. Specifically, the search kernel of BWA shows poor locality in its memory access pattern, and thus suffers very high cache and TLB misses. To address these issues, we propose a locality-aware design of the BWA search kernel, which reorders memory accesses to better take advantage of the caching and prefetching mechanism in modern multicore processors. Experimental results show that our improved BWA implementation can effectively reduce cache and TLB misses, and in turn, significantly improve the overall search performance.

Our specific contributions are as follows:

1) We carry out an in-depth performance characterization of BWA on modern multicore processors. Our analysis reveals crucial architecture features that will impact the performance of BWT-based alignment.
2) We propose a novel locality-aware design for exact string matching using BWT-based alignment. Our design refactors the original search kernel by grouping together search computation that access adjacent memory regions. The refactored search kernel can significantly improve memory access efficiency on multicore processors.
3) We evaluate the optimized BWA algorithm on two different Intel Sandy Bridge platforms. Experimental results show that our approach can improve LLC misses by 30% and TLB misses by 20%, resulting in up to 2.6-fold speedup over the original BWA implementation.

The rest of this paper is organized as follows. Section II provides a brief introduction of BWA. Section III presents our performance characterization of BWA on multicore processors. Section IV describes the design of our locality-aware BWA implementation. Performance evaluation is presented in Section V. Section VI surveys related work. Finally, Section VII concludes the paper.

II. Background on BWA

The Burrows-Wheeler Aligner (BWA) is one of the most widely used short-read mapping tools. It is based on the Burrows-Wheeler Transform (BWT), a data compression

technique introduced by Burrows and Wheeler [15] in 1994. The main concept behind BWT is that it sorts all rotations of a given text in lexicographic order and then returns the last column as the result. The last column, i.e., the BWT string, can be easily compressed, because it has many repeated characters. Similar to other BWT-based mapping tools, BWA uses the FM-index [12], a data structure built atop the BWT string that allows for fast string matching on compressed text in order to index the reference genome. In BWA, exact matching of a read (string) is done by a *backward search* [4], which essentially performs a top-down traversal on the prefix tree of the reference genome. The backward search stage accounts for the vast majority of the execution time.

A brief description of backward search in BWA is as follows.[1] Let $a \in \Sigma$ be the letter being considered and $c[a]$ be the number of symbols in $X[0, n-2]$ that are lexicographically smaller than a and $Occ(i, a)$ is the number of occurrences of a in the BWT string of the reference genome based on current position i. $c[a]$, $Occ(i, a)$ and the BWT string form the FM-index. String matching with the FM-Index tests if W is a substring of X, which is done by following a proven rule that $\underline{R}(aW) \leq \overline{R}(aW)$ if and only if aW is a substring of X:

$$\underline{R}(aW) = c[a] + Occ(\underline{R}(W) - 1, a) + 1$$

$$\overline{R}(aW) = c[a] + Occ(\overline{R}(W), a)$$

Iteratively applying the above rule, we get a narrowing search range declared by $\underline{R}(aW)$ and $\overline{R}(aW)$ (k and l in algorithm 1) until $\overline{R}(aW)$ is less or equal to $\underline{R}(aW)$.

The occurrence calculation, i.e., the Occ function, of a is the core function in backward search. A trivial solution of implementing the Occ function is counting the occurrences of a in all previous rows of the BWT string. This solution is inefficient when the BWT string is large. A widely accepted optimization, also used by BWA, is to break the whole BWT string into millions of small buckets and record pre-calculated accumulative counts of A/C/G/T for each bucket. BWA packages these pre-calculated counts along with the BWT string by inserting them before each BWT bucket. We will refer this augmented BWT string as *BWT table* in the rest of the paper (the memory layout of the BWT table is shown in Fig. 1). With the BWT table, the occurrence calculation can be reduced to counting only the occurrences in one bucket, which can be done in constant time. For example, in Fig. 1, the number of occurrences of the second C in bucket 8 equals to the sum 259, fetched from the head of the bucket, and 2, which is the number of occurrences of C in the bucket.

The Occ function in BWA has three steps as shown in Algorithm 2: (1) getting the bucket location based on the input i, (2) fetching the accumulative count at the header of the bucket for letter a, and (3) counting the occurrences of a in the bucket and returning the sum of the local count and the pre-calculate count. Note that the memory-access location in the BWT table is determined by the input i.

[1]We use the same notations as the original BWA paper [8].

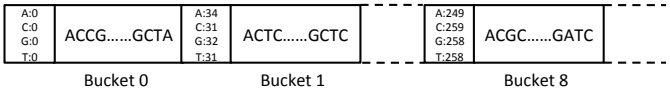

Figure 1. The memory layout of BWT table

Algorithm 1 Original Backward Search

Input: W: sequence reads
Output: k and l pairs

```
for all W_j do
    k = 0, l = |X|
    for i = len − 1 to 0 do
        a ← W_j[i]
        k ← c[a] + Occ(k − 1, a) + 1
        l ← c[a] + Occ(l, a)
        if k > l then
            output k and l
            break
        end if
    end for
end for
```

Algorithm 2 Occ function

Input: i: k or l values; a: letter in reads
Output: n: occurrences of a

```
p ← getBucket(i) {Step 1}
n ← getAcc(p, a) {Step 2}
n ← n + calOcc(p, a) {Step 3}
return n
```

III. Performance Analysis

In order to understand the performance characteristics of BWA, we collect critical performance counter numbers, such as branch misprediction, I-Cache misses, LLC misses, TLB misses, and microcode assists, using Intel VTune [2]. Fig. 2 shows the breakdown of cycles impacted by different performance events. As we can see, the percentage of stalled cycles is overwhelmingly high (more than 85%). Clearly, cache misses and TLB misses are the two major performance bottlenecks of backward search. Together, the two account for over 60% of all cycles. A closer look at the profiling data shows that the main source of these misses is the Occ function, which is the core function in backward search and accounts for over 80% of total execution time. Based on profiling numbers, within the Occ function, the stalled cycles caused by cache misses account for 55% of overall cycles,

and TLB misses caused stalled cycles occupy 41%. Thus, our optimization strategy focuses on how to optimize the memory access of the Occ function.

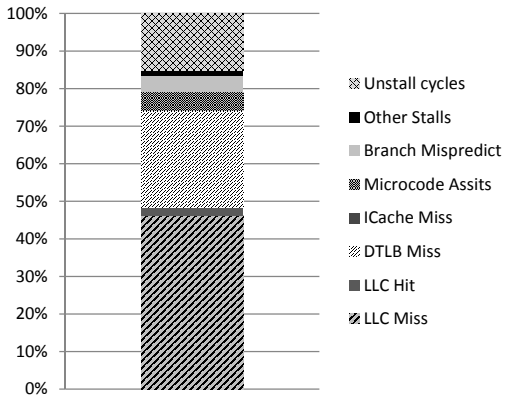

Figure 2. Breakdown of cycles of BWA with Intel Sandy Bridge processors

As we mentioned in Section II, in the Occ function (Algorithm 2), the input i (k or l in backward search) determines the access location in the BWT table. In order to further understand the memory access pattern of the Occ function, we trace the buckets that need to be accessed in calculating ks when searching an input read. As shown in Fig. 3, the access location in the BWT table jumps irregularly with large strides. Also, there seems to be little locality between consecutive access locations. The irregular memory access is the main reason for the high cache-miss rate. Furthermore, as the capacity of TLB is limited, large strides (e.g., larger than the 4K page size) over the BWT table can cause high TLB misses.

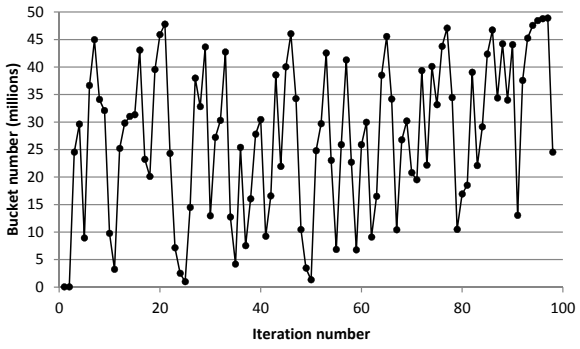

Figure 3. The trace of k in backward search for a read

Clearly, the backward searches of individual reads suffer from poor locality. However, we observe potential locality between processing of different reads. To reduce I/O overhead, BWA load millions of reads into memory as a batch. It is highly probable that multiple bucket accesses from different reads will fall into the same memory region. This observation is the main motivation of our optimizations, which will be presented in Section IV.

IV. Locality-Aware BWT-Based Alignment Design

In order to improve memory-access efficiency in BWT-based alignment, we propose a locality-aware backward search design, which exploits the locality of memory accesses to the BWT table from a batch of reads. Specifically, as discussed in Section III, the computation of occurrences, i.e., the Occ function, is the main source of cache and TLB misses. As shown in Algorithm 3, our design batches the occurrence computation from different reads by swapping the inner and outer loops of the original BWA implementation. To take advantage of CPU caching and prefetching, our design reorders memory accesses by grouping together the occurrence computation that accesses adjacent buckets of the BWT table.

A. Reordering Memory Access with Binning

To reorder memory accesses, our design maintains a list of bins, where each bin corresponds to several consecutive buckets in the BWT table. Designing a highly efficient binning algorithm here is challenging as it involves many competing factors. For instance, while binning can help improve memory access in occurrence computation, it also introduces extra memory accesses that can lead to undesirable cache and TLB misses. The design is also complicated by the complex memory hierarchy and prefetching mechanisms of modern processors.

1) Memory-Efficient Data Structure: The preliminary data structure of a bin entry is depicted in the left picture in Fig. 4. k, l and r_id are the input of the refactored Occ function, where k and l are corresponding top and $bottom$ in the original BWA implementation, and r_id is the id of the read being processed. Beside the three prerequisite variables, we add a small character array for data preloading to help reduce memory access overhead (discussed in Section IV-B). Such a bin entry requires 28 bytes to store. However, because of the data structure alignment, each element should occupy a multiple of the largest alignment of any structure member with padding, i.e., actually requires 32 bytes in memory. For a large batch of reads, there can be millions of entries, which can consume gigabytes of memory. To preserve the memory-efficiency of BTW-based alignment, we optimize the preliminary data structure as follows.

First, we observe an interesting property of k and l that can help shrink the data structure of a bin entry. In the original BWA implementation, the k and l are 64-bit integers, occupying 16 bytes in total. However, for the human genome, the maximum values of k and l are less than 2^{34}. Therefore, storing k or l only requires 33 bits, wasting the remaining 31 bits (in a 64-bit integer). To improve memory utilization, we pack k and l into a single 64-bit

integer such that k takes the first 33 bits, and l is represented as an offset to k in the remaining 31 bits. By doing so, the size of the bin structure can be significantly reduced. However, such a design requires the offset between k and l to be less than 2^{32}. Extensive profiling using data from the 1000 Genome Project [1] shows that the distance between k and l is always less than 2^{31} except the first iteration (example statistics are shown in Fig. 5(a)). This can also be explained in theory because the FM-index mimics a top-down prefix tree traversal, and as such, the distance between k and l decreases quickly as more letters are matched. Based on this observation, we package k and l by just skipping the first iteration. In addition, the trend shown in Fig. 5(a) implies that our method can be easily extended and used for larger genomes by skipping more initial iterations.

Second, cc is a small character array used to temporally store sub-sequences of reads. As the letters in the sequence reads are A/T/C/G and a few other reserved letters, we can use 4 bits to present a instead of 1 byte. By doing so, the 8-byte small char array can be packed into 4 bytes, i.e., a 32-bit integer.

The optimized data structure of a bin entry is shown in Fig. 4. With the aforementioned optimization, the size of an entry reduces by half, thus greatly improving memory efficiency. This can also improve cache performance as more entries can fit into a cache line.

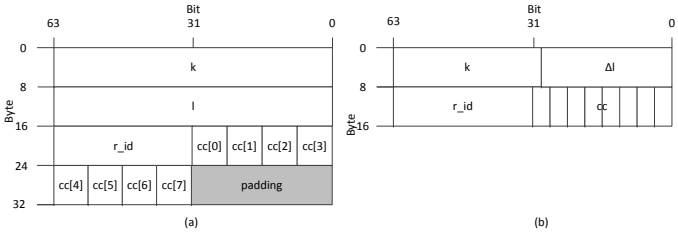

Figure 4. The layout of data structure of one element: preliminary (left), and optimized (right)

2) Bin Buffer Allocation: The memory allocation of the bin buffer is complicated by the fact that the number of entries in each bin varies significantly. Dynamic memory allocation can help workaround this variance but will introduce non-trivial overhead with frequent allocation requests. On the other hand, static allocation can reduce memory allocation overhead, but can lead to memory wastage. To achieve a balance between memory utilization efficiency and runtime overhead, we adopt a hybrid approach—which statically allocates a fixed buffer for each bin and uses a large pool to be stored overflow items.

By carefully analyzing the distribution of bucket accesses to the BWT table, we find that the number of accesses of individual buckets is more evenly distributed after the first few iterations. Since searching a read always begins with the same and k and l values, the access locations of the BWT

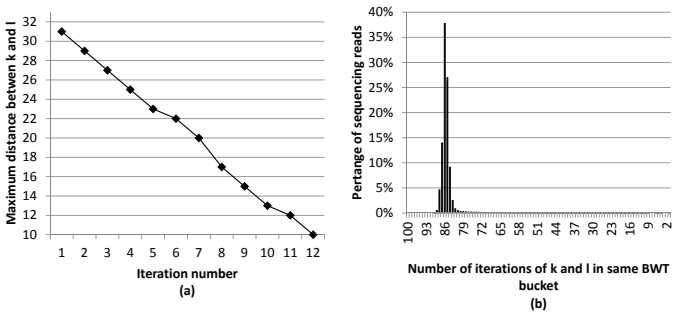

Figure 5. Properties of distribution of k and l (read length 100); (a) the maximum distance between k and l in a given iteration; (b) number of iterations in which k and l in same BWT bucket

table when searching different reads are almost the same for the first iteration. Based on the above observation, our implementation skips the first few iterations before starting the binning process and uses the average number of accesses across all buckets as the size of the preallocated buffer.

3) Cost-Efficient Binning: Compared to the original BWA implementation, our design involves extra computation in the binning process. It is critical to minimize the compute overhead of binning, to avoid offsetting the benefit from memory access reordering. To this end, our implementation simply right-shifts k and l to get the corresponding bin bucket numbers. However, the binning process can still introduce non-trivial overhead because it needs to be performed for every calculation of k and l. To further improve the binning efficiency, we leverage an interesting property from the BWT alignment; that is, the distance between k and l narrows fast as the search progresses. Fig. 5(b) shows statistics of the distance between k and l for representative input reads. As we can see, in matching reads of length 100, for most iterations (more than 80), k and l fall in the same bucket in the BWT table. This is because backward search mimics a top-down traversal over the prefix tree. In fact, this property was also used in the original BWA implementation to improve data reuse; the Occ function is optimized for the case where k and l fall in the same bucket to eliminate duplicated data loading from the BWT table. Based on this observation, our design applies binning only to k, which reduces the binning computation by half.

B. Reducing Binning Overhead with Data Preload

Although the basic binning algorithm can effectively improve locality of memory accesses, it does not come for free. The first two columns in Table I show the comparison between the original BWA and the preliminary binning implementations in cache misses and TLB misses for a representative input file. Surprisingly, while reordering memory access through binning can effective reduce the number of cache misses, it introduces more TLB misses.

With careful profiling, we find that the extra TLB misses are caused by indirect references on the sequence reads;

Algorithm 3 Optimized Burrows Wheeler Aligner

Input: W: sequence reads
Output: k and l pairs

```
1: for i = len' − 1 to 0 do
2:   for all bin_x do
3:     for all e_j in bin_x do
4:       if i mod cc_size = 0 then
5:         preload cc_size letters from reads to e_j.cc
6:       end if
7:       a ← get_a(e_j.cc, i mod cc_size)
8:       ok ← Occ(e_j.k − 1, a)
9:       ol ← Occ(e_j.l, a)
10:      e_j.k ← C[a] + ok + 1
11:      e_j.l ← C[a] + ol
12:      if e_j.k > e_j.l then
13:        output as result
14:      else
15:        y ← get_bin_number(e_j.k)
16:        fill e_j into bin_y
17:      end if
18:    end for
19:  end for
20: end for
```

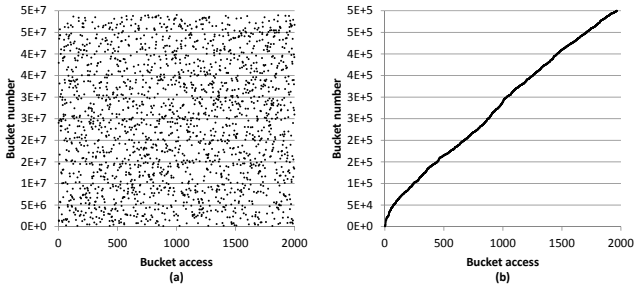

Figure 6. Bucket access pattern of original backward search and binning version: (a) memory access pattern without binning; (b) memory access pattern with binning. Since the amount of bucket access of whole backward search procedure is huge, here we just list first thousands accesses.

when backward search needs to fetch the next character from a read, it uses the read id r_id to locate the corresponding buffer storing the read sequence. As shown in Fig. 7(a), in the original algorithm, the access on a sequencing read is sequential, and thus, fetching read sequence data can benefit from the prefetching mechanism available in modern processors. However, in the preliminary binning design, letters at a location from different reads are processed together as shown in Fig. 7(b). As a consequence, prefetching of sequence data from a read cannot be reused, causing more frequent accesses to read data. As a batch of reads typically occupies several hundreds of megabytes, accessing such a memory space with large strides cause an overflow of the TLB cache.

To mitigate this issue, we add a small character array

Table I
PERFORMANCE NUMBERS OF ORIGINAL BACKWARD SEARCH, PRELIMINARY BINNING ALGORITHM AND OPTIMIZED BINNING ALGORITHM WITH A SINGLE THREAD ON INTEL I5 AND BATCH SIZE 2^{24}. LLC AND TLB MISSES ARE MEASURED IN THE UNIT OF MILLIONS.

	LLC Misses	TLB misses	Execution Time
Original	70	27	3.60
Preliminary binning	56	59	3.28
Binning with preload	53	23	2.60

in each bin entry to periodically store letters loaded from sequence reads. As the character array is embedded in every entry, it will be loaded in the cache when the corresponding k and l are processed, thus greatly reducing TLB misses in fetching the read data. As shown in the third column of Table I, the enhanced binning design can significantly reduce cache misses without incurring extra TLB misses.

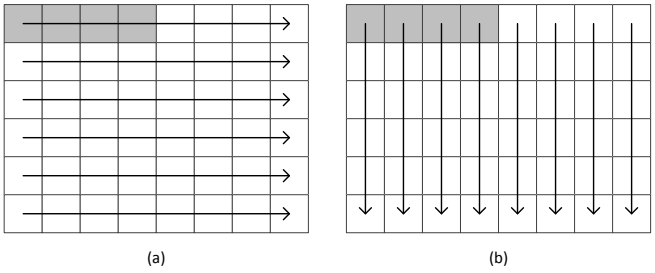

Figure 7. Access pattern for interchanged loop nest in backward search: original BWA (left), and binning BWA (right) - each box presents a block of data; shaded boxes are data loaded into cache lines; arrows show the memory access direction.

Fig. 8 shows the execution profile of the enhanced binning algorithm, collected using Intel VTune. Compared to Fig. 2, the number of non-stall cycles improves from 15% to 30%. Also, the stall cycles caused by TLB misses are greatly reduced. The differences in the execution profiles suggest that our memory-access reordering design is effective in improving memory access efficiency.

C. Multithreading

Multicore architectures add more complexity to our design. False sharing of data between threads can cause thrashing and severely impact performance. A straightforward approach to parallelize our binning design is to have each thread maintain a separate bin and work independently. The disadvantage of such a design is that the memory bandwidth of a multicore processor cannot be efficiently utilized because there is no data sharing between threads. Therefore, in our design, all threads share the same bin structure. A design challenge then lies in how to efficiently synchronize between different threads.

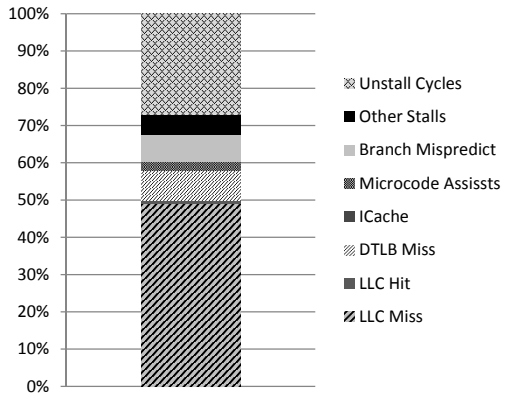

Figure 8. The breakdown of cycles of optimized BWA algorithm

To minimize synchronization overhead, our design maintains two copies of the bin structure. In the beginning, one copy of the bin structure stores all initial values and is marked as read-only. Another copy of the bin structure is marked as write-only. Extensive profiling shows that the processing time of each bin is about the same. Therefore, our design uses a static task-allocation approach, where all bins in the read-only structure are evenly distributed among all of the threads. When processing a bin, each thread computes the k and l of each entry and places them in the corresponding bin of the write-only structure. The read-only and write-only structures are swapped in the next iteration. Such a design reduces the synchronization overhead as there is no need to coordinate accesses to the read-only structure. For the write-only structure, one index is maintained for each bin to mark the last entry in the bin. Thus, a new entry can be safely placed at the end of the bin by executing an atomic add on the index associated with the bin. Our profiling shows that such an design incurs very low overhead, partly because the contention on a particular bin is typically low.

V. Performance Evaluation

We evaluate the performance of our implementation in three aspects: impact of software configuration, impact of micro-architecture, and scalability.

A. Experiment Setup

In order to evaluate the impact of variance of the micro-architecture, particularly cache size, two different Intel Sandy Bridge CPUs are used in our experiments: (1) Intel Core i5 2400 is a high performance quad-core microprocessor with high clock frequency; (2) Intel Xeon E5-2620, a hex-core processor designed for servers, has a lower clock frequency, but is integrated with a large on-chip L3 cache. To eliminate effects of Hyper-threading (HT) on cache performance, we disable HT on the Intel Xeon E5-2620 via BIOS setting.

While our experiments focus on human genome sequencing, none of our analysis is specific to such a genome and easily carry over to other genomic datasets as well. We use sequence datasets from the GenBank database. The read queries used in our paper are from the 1000 Genome Project. To evaluate the impact of read lengths, we choose 4 read queries with different lengths. In the remaining experiments, we use a read query with 100bp as default input.

B. Impact of Software Configuration

In the optimized BWA algorithm, there are three important parameters: (1) preloaded data size - the number of letters in sequence reads preloaded; (2) bin range - the range of bucket access grouped into a bin; (3) batch size - the number of sequence reads loaded into memory to be processed. To achieve optimal configuration of these parameters, we quantify the impacts of the three parameters in this section.

1) Preloading Data Size: The size of preloading data determines the frequency of preloading data. A larger preloaded data size implies less indirect references, but fatter elements and larger memory footprint. In Fig. 9 (a), we can see that when the preloaded data size increases from 4 to 32, the performance improves slightly and peaks at 16.

2) Bin Range: The bin range determine the granularity of memory reordering. A smaller bin range indicates that bucket accesses are more in-order. However, the overhead of binning increases as the bin range reduces. As the cache in modern CPU can be up to several megabytes, which can contain millions of elements, the elements in one bin are unlikely to be evicted before the next bucket is accessed. As we can see in Fig. 9 (b), the overhead of binning increases with decreasing bin range causing the performance to suffer noticeable degradation when the bin range is reduced to 16.

3) Batch size: Batch size is a critical parameter, significantly influencing the overall performance. In Fig. 9 (c), we observe that increasing the batch size dramatically improves cache performance, and consequently the overall application performance. But, increasing batch size barely impacts the performance of original BWA. A large batch size allows more bucket accesses, and more accesses fall into a bin, increasing the possibility that multiple bucket accesses hit the same cache line. Due to memory space limitation, we can maximally get 2.6-fold speedup with 16 GigaBytes memory. If further increasing batch size with larger memory, we can achieve more performance gain.

C. Impacts of Read Length

To clarify the impact of read length, we compare the performance of the original and optimized versions of BWA with different read lengths. We notice that the difference of read lengths has little influence on the speedup of the optimized BWA algorithm as shown in Table.II; that is, the speedup is stable with different read lengths on both single-thread and multi-threaded tests.

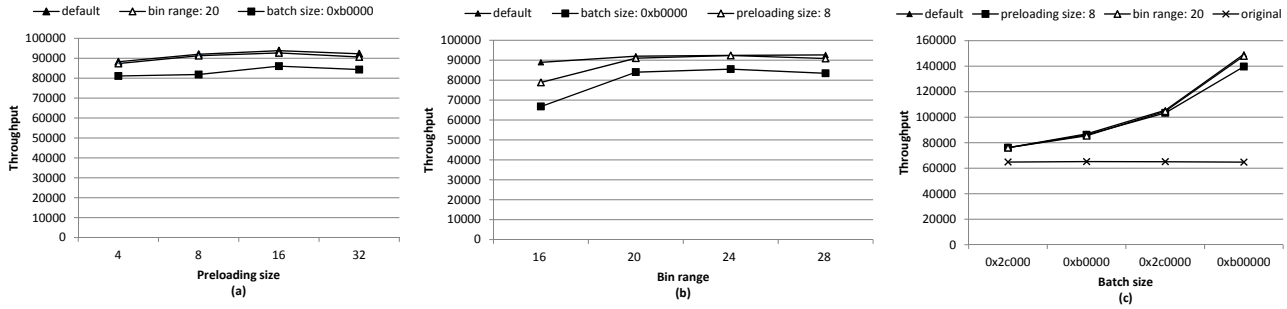

Figure 9. Effects of software configuration: (a) preloaded data size, (b) bin range, (c) batch size, default line in each graphs indicates other two parameters are using default setting - batch size: 0x160000, bin range: 24 and preloading data size: 16. To unify the execution time of different batch sizes, the performance is normalized to throughput - reads per second

Table II
EFFECTS OF READ LENGTH

	SRR003084(36bp)			SRR003092(51bp)			SRR003196 (76bp)			SRR062640(100bp)		
	unopt	opt	speedup	unopt	opt	speedup	unopt	opt	speedup	unopt	opt	speedup
single thread	6.21	4.04	1.43	6.87	4.65	1.44	12.79	8.93	1.54	20.54	14.26	1.48
multithreads	2.4	1.87	1.28	3.79	3.14	1.21	1.21	0.89	1.36	1.29	0.99	1.30

D. Impacts of Micro-Architectures

Micro-architectures can differ in several aspects. In this paper, we mainly focus on cache size. To understand the effect of the variation of cache size, we profile both the original and optimized backward search on the two Intel CPU architecture models described in Section V-A.

As shown in Fig. 10, the speedup on the Intel Xeon CPU is not better than that on Intel i5 CPU, despite the larger cache on the Intel Xeon. This is because our optimization mainly improves spatial locality of the algorithm, which is sensitive to cache line size rather than cache size itself. Furthermore, the higher single-core performance on Intel i5 benefits our optimized algorithm. If we restrict the frequency of Intel i5 to 2GHz, which is the same as the Intel Xeon, the speedup drops to close to that achieved by the Intel Xeon (Fig.10).

E. Scalability

Fig. 11 shows the strong and weak scalability of the optimized BWA algorithm. We notice that the weak scaling of the optimized algorithm is pretty close to ideal, with an approximately 10% loss of scalability going from 1 thread to 6 threads. Strong scaling numbers show a 4.5X speedup with 6 cores (that is, a parallel efficiency of 75%).

We further analyze the loss of scalability for strong scaling in Table III, through a more detailed architectural analysis. We notice that the multithreaded version suffers more cache and DTLB misses due to interference among threads, thus resulting in some loss of performance.

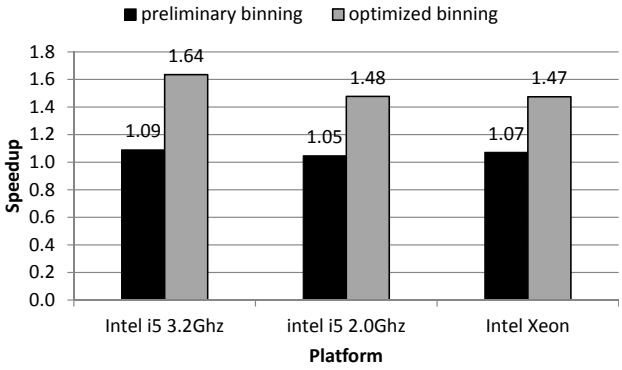

Figure 10. Speedup of optimized backward search algorithm over original algorithm on different platforms with a single thread

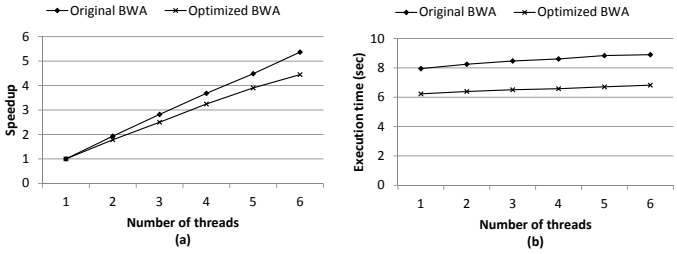

Figure 11. Scalability of optimized binning backward search: (a) and (b) are strong and weak scaling on Intel Xeon platform

VI. RELATED WORK

With the explosive growth of NGS data, many short-read mapping tools have been developed in the past a

Table III
CACHE MISSES AND TLB MISSES (IN MILLIONS) OF OPTIMIZED Occ FUNCTION

	Cache miss	DTLB miss	CPI
multi-threads	127	87	2.077
single thread	109	49	1.589

couple of years. Among them, BWT-based tools [8], [6], [7] are becoming popular because of their small memory footprints. Currently, most optimization studies on BWT-based mapping tools focus on accelerators. For instance, BarraCUDA [6], proposed by Petr Klus and Simon Lam, is a GPU-accelerated mapping tools adapted from BWA. BarraCUDA achieves a 3-fold speedup with 8 NVIDIA GPUs over a 12-core CPU. Another GPU-based short read aligner based on BWT released by Liu and Schmidt, CUSHAW [11], achieves a 4-6x speedup with 2 NVIDIA GPUs over a 4-core CPU. Recently, Torres and Espert propose another GPU-based alignment algorithm [14], which is 3-times faster than Bowtie and 4-times faster than SOAP2. Besides GPU acceleration, there have been multiple studies [13], [3] in optimizing BWT-based alignment with FPGA (Field-Programmable Gate Array). Our paper, on the other hand, focuses on optimizing BWT-based alignment on multicore CPUs by remapping the algorithm to better exploit the caching mechanism of modern processors. In addition, our study performs in-depth performance characterization of BWA, which has not been reported by previous work.

Irregular memory access has also been observed for hash-table based short-read mapping tools. Wang and Tang [16] proposed a memory optimization of hash-index for NGS by reordering memory access and compressing the hash-table. Another cache-oblivious algorithm for short-read mapping is proposed by Hach and Hormozdiari [5]. Our paper is the first to investigate locality-aware implementation of BWT-based alignment, which involves more complicated data structures and more sophisticated memory-access patterns.

VII. CONCLUDING REMARKS

In this paper, we first presented an in-depth performance characterization with respect to the memory access pattern and cache behavior of BWT-based alignment. We then proposed a well-designed optimization approach to improve data locality of backward search via binning. Our optimized BWA algorithm achieves up to a 2.6-fold speedup, and a good weak scaling on multicore architectures. As our optimization approach is generic, it can be easily extended and used in other BWT-based applications. In the future, we will extend our work to support inexact matching in BWA.

ACKNOWLEDGMENT

This work is in part supported by NSF Grant 0916719 "Collaborative Research: Hybrid Opportunistic Computing for Green Clouds" and NSF Grant 1048253 "Commoditizing Data-Intensive Biocomputing in the Cloud."

REFERENCES

[1] A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, Oct. 2010.
[2] Intel VTune Amplifier XE 2013, Product Brief, August 2012.
[3] S. Arming, R. Fenkhuber, and T. Handl. Data compression in hardware - the Burrows-Wheeler approach. In E. Gramatov, Z. Kotsek, A. Steininger, H. T. Vierhaus, and H. Zimmermann, editors, *DDECS*, pages 60–65. IEEE, 2010.
[4] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398.
[5] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature methods*, 7(8):576–577, Aug. 2010.
[6] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC research notes*, 5(1):27, Jan. 2012.
[7] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, Jan. 2009.
[8] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics (Oxford, England)*, 25(14):1754–60, July 2009.
[9] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–83, Sept. 2010.
[10] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics (Oxford, England)*, 25(15):1966–7, Aug. 2009.
[11] Y. Liu, B. Schmidt, and D. L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, 28(14):1830–1837, July 2012.
[12] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
[13] J. Martinez, R. Cumplido, and C. Feregrino. An FPGA parallel sorting architecture for the Burrows-Wheeler transform. In *Reconfigurable Computing and FPGAs, 2005. ReConFig 2005. International Conference on*, RECONFIG '05, pages 17–, Washington, DC, USA, 2005. IEEE Computer Society.
[14] J. Salavert Torres, I. Blanquer Espert, A. Tomas Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. Dopazo. Using GPUs for the exact alignment of short-read genetic sequences by means of the Burrows-Wheeler transform. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 9(4):1245–1256, July 2012.
[15] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the Conference on Data Compression*, DCC '97, pages 469–, Washington, DC, USA, 1997. IEEE Computer Society.
[16] W. Wang, W. Tang, L. Li, G. Tan, P. Zhang, and N. Sun. Investigating memory optimization of hash-index for next gneration sequencing on multi-core architecture. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 665–674, May 2012.