Sequence Alignment Through the Looking Glass

Raja Appuswamy
Data Science Department
EURECOM
France
raja@rajaappuswamy.com

Jacques Fellay
School of Life Sciences
EPFL
Switzerland
jacques.fellay@epfl.ch

Nimisha Chaturvedi
School of Life Sciences
EPFL
Switzerland
nimisha.chaturvedi@epfl.ch

***Index Terms*—sequence alignment, computational genomics, microarchitecture**

***Abstract*—Rapid advances in sequencing technologies are producing genomic data on an unprecedented scale. The first, and often one of the most time consuming, step of genomic data analysis is sequence alignment, where sequenced reads must be aligned to a reference genome. Several years of research on alignment algorithms has led to the development of several state-of-the-art sequence aligners that can map tens of thousands of reads per second.**

In this work, we answer the question "How do sequence aligners utilize modern processors?" We examine four state-of-the-art aligners running on an Intel processor and identify that all aligners leave the processor substantially underutilized. We perform an in-depth microarchitectural analysis to explore the interaction between aligner software and processor hardware. We identify bottlenecks that lead to processor underutilization and discuss the implications of our analysis on next-generation sequence aligner design.

I. Introduction

The past few years have witnessed dramatic improvement in both cost and throughput of DNA sequencing technologies. Today, it is possible to sequence a single human genome at 30-fold coverage for as little as $1,000. With sequencing becoming more affordable, the amount of genomic data generated has been increasing at an alarming rate far outpacing Moore's law [18]. This data deluge has the potential to pave way for the emerging field of personalized medicine and assist in detecting when genomic mutations predispose humans to certain diseases like cancer, autism, and aging.

However, to unlock the potential of genomic data, one needs scalable, efficient data analytics platforms and tools. The first and one of the most time-consuming steps in analyzing such data is sequence alignment–the task of determining the location in the reference genome that corresponds to each sequenced read. In the last decade, researchers have designed over 70 read mapping tools [8], each differing from another with respect to accuracy, sensitivity, specificity, and speed. Today, state-of-the-art read aligners can map tens of thousands of reads per second to the reference genome.

While all prior research focuses on improving performance, by reducing mapping time of individual reads, or scalability, by mapping more reads per second, there is no study that analyzes sequence aligners at the microarchitectural level. Such an analysis is important for several reasons.

First, it answers the following question–Do state-of-the-art aligners use modern processors efficiently? Microarchitectural analysis in other applications areas, like relational databases and data analytics platforms, showed that these applications do not utilize modern processors efficiently [1], [6]. Such analyses spurred research efforts to improve performance by improving processor utilization by redesigning data structures, or energy efficiency by using low-power processors that can match application requirements. In this work, we perform a similar analysis for sequence alignment.

Second, modern processors are complex pieces of hardware that use a variety of techniques to execute instructions faster. However, in order for such improvements to translate into tangible performance benefit, software must be optimized to avoid bottlenecks. Microarchitectural analysis reveals these bottlenecks by exposing harmful interaction between application software and processor hardware and helps in answering the following question–Will current software automatically benefit from microarchitectural improvements in the next-generation of hardware?

Third, the past few years have witnessed an rise in adoption of heterogeneous computing, as accelerators like General-Purpose Graphics Processing Units (GPGPU) and Intel Xeon Phi are being increasingly adopted in several data-intensive application domains. Given that sequence alignment is a complex, multi-stage process, several researchers have built aligners that execute some, or all stages of sequence alignment, on these accelerators [11], [16], [17], [19]. However, there has been no systematic analysis that explores the interaction between alignment stages and CPU microarchitecture to clearly identify which stages are more suited to the GPGPU than the CPU.

In this work, we present, to our knowledge, the first microarchitectural analysis of four state-of-the-art sequence aligners. We show that despite a decade of research and optimized implementations, modern aligners substantially underutilize processor resources as the processor is stalled in more than 50% of execution cycles without doing useful work. We provide an in-depth breakdown of stall cycles to identify hardware components in the processor pipeline and algorithmic components in sequence alignment software that contribute to these stalls. We discuss the implications of our analysis on the design of next-generation of sequence aligners and suggest directions for further research.

The rest of this paper is organized as follows. In Section II, we provide a brief overview of alignment techniques and processor microarchitecture. In Section III, we describe the hardware and software setup we use for this analysis. We present a global microarchitectural analysis of aligners in Section IV and a stage-by-stage analysis of one the aligners in Section V. Finally, we present the implications of our analysis in Section VI and conclude in Section VII.

II. Background

In this section, we will provide a brief overview of sequence alignment techniques and modern processor microarchitecture to set the context for this work. We refer the reader to prior work for an in-depth algorithmic survey of sequence alignment algorithms and experimental analysis of aligners [5], [8]–[10].

A. Sequence alignment

Modern Next-Generation Sequencing (NGS) technologies produce millions of short string sequences, referred to as reads, with each sequence corresponding a portion of the DNA. The first step in the analysis of this NGS data, referred to as sequence alignment, is to determine the location in the genome that corresponds to each of these short reads. Thus, sequence alignment is essentially a string matching problem where given a string G (reference genome), and a set of substrings R (reads), the origin of each read r ϵ R must be identified in G. However, due to sequencing errors or due to differences between the reference genome and the sequenced organism, a read might not exactly match its corresponding location in the reference genome. Thus, an aligner has to perform approximate string matching that is tolerant to mismatches, insertions, and deletions.

Since a read could potentially align at each one of the 3 billion locations in the reference, brute force search of each possible alignment is infeasible even for a single read. Thus, all modern aligners build an index over the reference and use the index to quickly narrow down the search space of potential locations. Aligners can be broadly classified into two types based on the indexing technique used, namely, seed-and-extend (SE) aligners or Burrows-Wheeler-Transform (BWT)-based aligners.

SE aligners typically use a hashtable to index the reference genome and store the contents and the occurrence locations of short string sequences, also referred to as seeds or k-mers, in a hash table. Each read is processed in three steps. First, seeds are extracted from the read and the hash table is used to look up potential mapping locations in the reference genome. Second, filtering techniques are used to reduce the number of potential locations where further extension must be performed. Third, the entire read is aligned at each of potential reference location using an approximate string matching algorithm like Needleman-Wunsch or Smith-Waterman.

BWT-based aligners align reads against a suffix array built using the reference genome. As suffix arrays are memory intensive, all of these aligners use a space-optimized data structure called FM-index [7] that uses a compressed string representation called Burrows-Wheeler Transform [4] to index the reference genome and enable fast exact string matching. As the FM-Index itself does not allow approximate string matching, BWT-based aligners use an algorithmic technique called backtracking that tries to insert errors at various positions in the read as it is matched while traversing the FM-Index. As the cost of backtracking increases exponentially, BWT-based aligners also use heuristics to prune the search space.

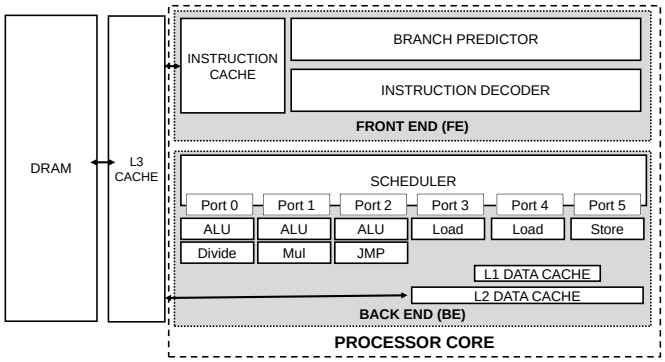

Fig. 1: Simplified microarchitectural depiction of modern processors.

BWT-based alignment, introduced by Bowtie [12] and BWA [14], has been the most popular technique for aligning short reads. The low error rate of NGS technologies and a low divergence between reference and sequenced organisms allowed aligners like trie-based Bowtie and BWA to traverse the search space for very short reads 10 to 100 times faster than hash-based aligners. However, with read lengths increasing to 100-150 base pairs, backtracking emerged as the bottleneck especially if one needs to tolerate a larger number of errors. Thus, newer variants of these aligners, like Bowtie2 [2] and BWA-MEM [13] have also reverted back to using the seed-and-extend technique.

B. Modern processor microarchitecture

In order to understand how state-of-the-art sequence aligners utilize modern processors, we need to examine how well are the microarchitectural resources of a processor used by the aligner software. The micrioarchitectural pipeline of a modern high-performance processor is quite complex. Figure 1 shows a simplified view of the microarchitectural components of a processing core. Modern processors are typically multi-core in nature and thus contain several of these processing cores. For the rest of this paper, we will use the term processor and core interchangeably.

The pipeline of a processor is divided conceptually into two halves, the Front-end (FE) and the Back-end (BE). The FE is responsible for fetching the program code corresponding to the Instruction Set Architecture and decoding them into one or more low-level hardware operations called micro-operations (μOps). Once decoded, these μOps are queued for execution by the BE. Before a μOp can be executed, all necessary data operands must be fetched from memory if necessary. The BE

scheduler is responsible for monitoring when a μOp's data operands are available. Once ready, the scheduler associates the μOp with a port depending on its intended execution purpose. For instance, a μOp corresponding to an arithmetic operation would be associated with a port between 0 and 2, while a μOp associated with loading data from memory would be associated a port between 3 and 5 in Figure 1. When an execution unit is available, the port dispatches the queued μOp and executes it. Execution units, labelled as ALU, Divide, Mul, etcetera, in Figure 1, are the work horses that perform various operations like memory loads and stores, addition, multiplication, division, etcetera. The FE and BE are also equipped with both on-chip and off-chip cache memory, shown as L1/L2/L3 instruction/data caches in Figure 1, to avoid long-latency DRAM accesses by buffering data and instructions.

The completion of a μOp's execution is called *retirement*. When a μOp is retired, its results are committed back to the architectural state by updating CPU registers or writing back to memory. The Front-end of the pipeline on recent Intel microarchitectures can allocate four μOps per clock cycle, while the Back-end can retire four μOps per clock cycle. Thus, in each clock cycle, modern Intel processors can potentially execute four instructions simultaneously. During instruction execution, most μOps pass completely through the pipeline and retire. But sometimes, a μOp that is not be able to complete immediately might delay, or *stall*, the pipeline.

Intel's Top-Down Analysis Methodology [20] classifies stalls into three major types, namely *Front-end* stalls, *Speculation* stalls, and *Back-end* stalls. During a cycle, if the BE is ready to execute a μOp but the FE is unable to queue it for execution, the stall is classified as a FE stall. A typical reason for FE stalls is instruction cache misses caused by large instruction footprint corresponding to a complex code base. Disk-based relational database engines are known to suffer from such stalls [1].

Modern processors use speculative execution to improve instruction throughput. Conditional execution in programs, like if–else blocks get translated into branch instructions that decide control flow depending on predicates in the conditional statement. Before the processor has to execute a branch instruction, the predicate value has to be determined. Instead of waiting until a branch instructions predicate is resolved, the Branch Predictor component in the processor FE implements an algorithm that guesses the predicate and fetches the appropriate instruction stream. If the guess is correct, the execution continues normally, and if it is wrong, the pipeline is flushed, and the correct instruction stream is fetched and executed. Such flushing creates pipeline stalls and these stalls caused by incorrect branch prediction are classified as Speculation stalls.

In the case where the FE has a μOp ready but the BE is not ready to handle it, the processor is stalled on the BE. BE stalls can be further classified into *Memory stalls* and *Core stalls*. As mentioned earlier, a μOp can be executed only if its data operands are available. Memory stalls are caused by the processor having to wait for such data operands to be fetched from the cache or from memory. Core stalls, in contrast, are caused by a less-than-optimal use of the available execution units during each cycle. This can happen due to contention for resources. For instance, if the code contains instructions that result in several μOps being associated with a few ports, then queue for those ports become full. Thus, the scheduler can no longer associated any further μOps with those ports until the queue shrinks. Similarly, if the code contains several divide instructions in a row, they will compete for the few divide execution units resulting in resource conflicts.

III. Experimental Setup

In this section, we describe the hardware and software setup we use in this analysis and outline the experimental methodology.

A. Hardware–software setup

All experiments are conducted on a server running RHEL 7.2, equipped with a 12-core Intel Xeon E5-2650L v3 CPU and 256GB RAM. We analyze four state-of-the-art sequence aligners, namely, *BWA-MEM* [13] and *Bowtie2* [2], *Snap* [21], and *FSVA* [15]. We chose *Bowtie2* and *BWA-MEM* as they are the most popular short-read aligners that use a BWT-based reference index. We chose *Snap* as a representative of a new breed of hashtable-based aligners that exploit the increasing read lengths to improve performance without sacrificing accuracy. We chose *FSVA*, as it is a state-of-the-art aligner built for cohort studies that explicitly trades off accuracy for fast single-threaded performance.

As described in Section II, all these aligners use the seed-and-extend technique to perform fast alignment of reads. However, these aligners differ dramatically with respect to the actual methodology used for seed selection, filtration, and extension. As a result, these aligners occupy different points in the performance–accuracy dimensions. Our goal in this analysis is to not to perform a side-by-side analysis of execution time or accuracy of various aligners. The optimal aligner choice is a complex analysis topic covered by prior research [5], [8]–[10] as there is a delicate balance between accuracy and performance that must be met depending on the expected usage. As our focus in this paper is on microarchitectural analysis of sequence aligners, we run each aligner only in the single threaded mode. Thus, we report the execution time only for completeness.

B. Experimental methodology

We use both synthetic and real data set for evaluating each system. The synthetic dataset is generated by using wgsim. We generate two datasets with one million reads of length 150 bp each, one with the default rate of indels and one without any indels or substitutions. We use the two simulated datasets to perform a sensitivity analysis to the presence of indels. For the real dataset, we use a paired-end read (sample NA12878/09252015) obtained from the public Genome-In-A-Bottle (GIAB) dataset [22].

We use Intel VTune for profiling each system. Before profiling, each aligner is run once to warm up the file system

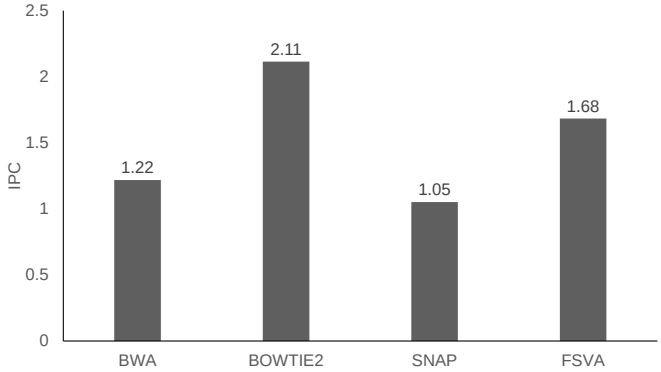

Fig. 2: IPC for indel-free dataset

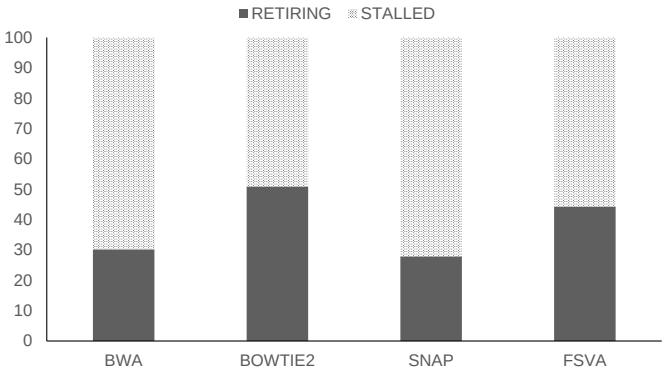

Fig. 3: Execution cycle breakdown for indel-free dataset

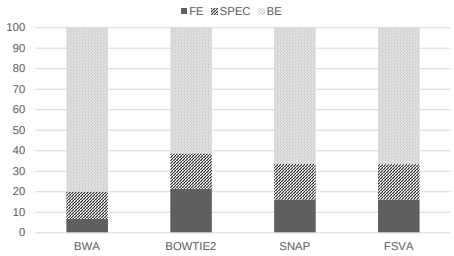

Fig. 4: Stall cycle breakdown for indel-free dataset

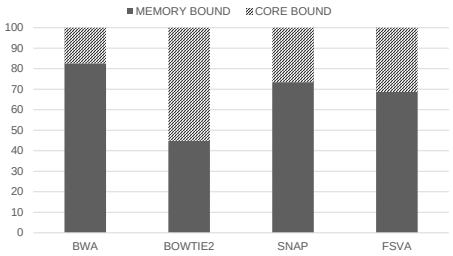

Fig. 5: Back-end stall breakdown for indel-free dataset

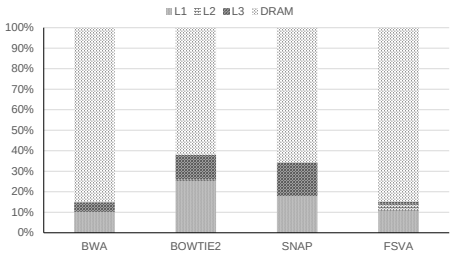

Fig. 6: Memory stall breakdown for indel-free dataset

Aligner	Simulated Indel-Free	Simulated With-Indel	GIAB
bowtie2	420	437	3126
bwa	176	458	2919
snap	47	112	717
fsva	55	80	288

TABLE I: Execution time in seconds of aligners under different data sets

cache and ensure that the necessary indices and input data are memory resident. Then, we profile each analyzer by executing it for 30 seconds to warm up the instruction cache and then use VTune to to profile the target thread for another 30 seconds.

IV. Analysis results

In this section, we present our analysis of the four sequence aligners with respect to processor utilization.

A. Indel-free dataset analysis

Modern processors use an array of techniques like pipelining, out-of-order execution, speculation, and instruction prefetching to improve single-threaded performance. As a result, modern processors can retire multiple instructions per clock cycle as described in Section II. The efficiency of a software is determined by the metric Instructions Per Cycle (IPC), which determines the number of machine instructions executed and retired by the processor in each clock cycle. The Intel processor we use in this analysis can retire four instructions per cycle. Thus, in the ideal case, the IPC value of a sequence aligner should be four.

IPC analysis. Table I shows the single-threaded execution time of aligners and Fig 2 shows the IPC values for the four aligners under the indel-free simulated dataset. There are two important observations to be made from Figure 2. First, note that the maximum IPC value observed across all aligners is only about 50% of the theoretically achievable maximum. Second, although *Snap* has the lowest execution time among the four aligners, we see its IPC is the lowest. Counter intuitively, faster aligners seem to utilize the processor even lesser than slower aligners.

Execution cycle breakdown. In order to further understand why observed IPC values are lower than the theoretical maximum, and explain difference in IPC across aligners and under different datasets, we need to analyze the processor activity on a finer microarchitectural level. Figure 3 shows the breakdown of execution cycles for each aligner into retiring and stalled for the indel-free datasets. We see that even in the best case, the processor is stalled in nearly 50% of cycles. This implies that the processor is substantially underutilized under all aligners and these stalls result in the low IPC values observed earlier.

Stall-cycle breakdown. Given that a substantial fraction of cycles are stall cycles, the next step is to identify the root cause of these stall cycles. Figure 4 shows the stall cycle breakdown for each aligner under the indel-free dataset. Clearly, the dominating source of stalls is the Back-end which accounts for 60% to 80% of all stall cycles. Speculation stalls occupy the second largest share of cycles accounting for 13% to 17%.

Figure 5 breaks down the back-end stalls further into mem-

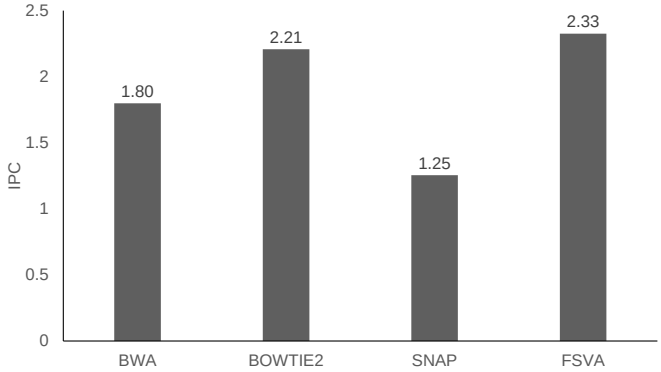

Fig. 7: IPC for with-indel dataset

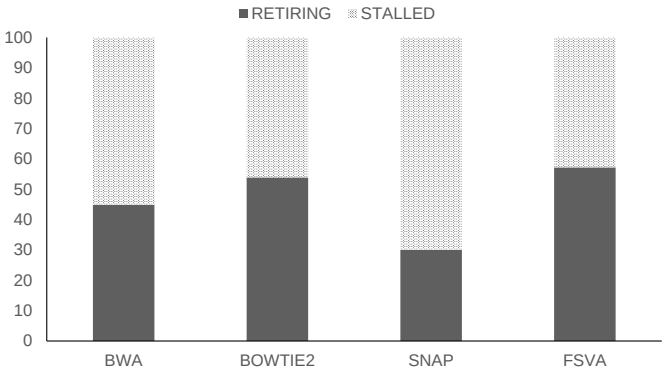

Fig. 8: Execution cycle breakdown for with-indel dataset

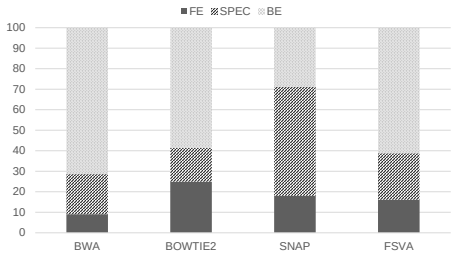

Fig. 9: Stall cycle breakdown for with-indel dataset

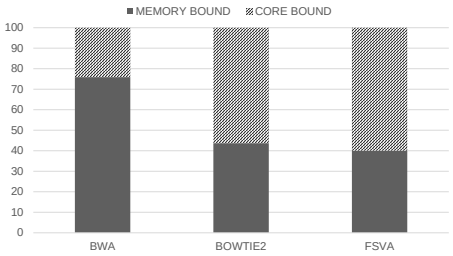

Fig. 10: Back-end stall breakdown for with-indel dataset

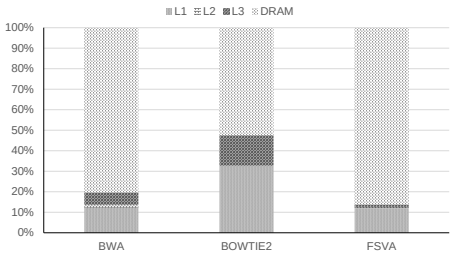

Fig. 11: Memory stall breakdown for with-indel dataset

ory or core bound. Memory stalls account for a 45% to 80% of back-end stalls across all aligners. This indicates that the processor is stalled waiting for data. *Bowtie2* suffers more than other aligners from core stalls indicating that in addition to waiting for data, the *Bowtie2* code results in resource conflicts where instructions compete for similar execution units.

Memory stalls analysis. Given that memory stalls dominate the back-end across all aligners, it is important to know if these stalls are due to cache-resident data or DRAM-resident data. The Intel processor we use in this study has a three-level caching hierarchy, with a 32KB L1 cache, 256KB L2 cache, and 30MB Last-level cache. In general, software optimizations attempt to move data closer to the processor so that critical data structures are L1-cache resident. Thus, a memory stall at a lower cache level typically indicates an optimization opportunity where data structure redesign can improve utilization. However, if memory stalls are due to DRAM-resident data, this is typically due to cache-unfriendly random data access pattern which is harder to optimize.

Figure 6 breaks down memory stalls into load and store stalls. Load stalls are further decomposed based on the location that contributes to the stall (L1, L2, L3 cache, or DRAM). Clearly, between 60–80% of memory stalls are DRAM stalls indicating that the processor is waiting for long-latency memory accesses from DRAM.

Insights. *All aligners substantially underutilize processors as over 50% of execution cycles are spent on stalls. The processor is stalled as the Back-end is blocked on long-latency DRAM accesses waiting for data.*

B. Analysis of dataset with indels

Having analyzed the microarchitectural behavior of aligners under the indel-free dataset, we now present our results using the with-indel dataset. Our goal is to understand if the presence of indels changes the microarchitectural behavior of aligners.

IPC analysis. Table I shows the single-threaded execution time of aligners under the with-indel simulated dataset. Figure 7 shows the corresponding IPC values. Comparing the execution times between indel-free and with-indel datasets, we see that the execution time of all aligners increases in the presence of indels. This is expected given that the indels trigger expensive approximate alignment algorithms. However, comparing Figures 2, 7, we see that the IPC value of aligners under dataset with indels is higher than the IPC value without errors.

Execution cycle breakdown. Figure 8 shows the breakdown of execution cycles for each aligner into retiring and stalled for the with-indel datasets. Similar to the indel-free dataset (Figure 3), we see that even in the best case, the processor is still stalled in nearly 50% of cycles. However, across all aligners, the processor retires more instructions when the dataset has indels compared to a dataset without indels although. This directly translates into corresponding difference in IPC observed earlier between the two datasets.

Stall cycle breakdown. Figure 9 shows the stall cycle breakdown for each aligner under the with-indel dataset. Comparing this with the indel-free case (Figure 4), we can make two observations. First, under *Bowtie2*, *BWA-MEM*, and *FSVA*, the dominating source of stalls is still the Back-end which

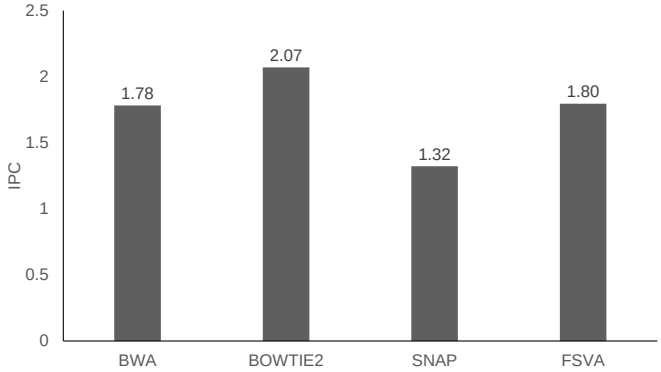

Fig. 12: IPC under GIAB dataset

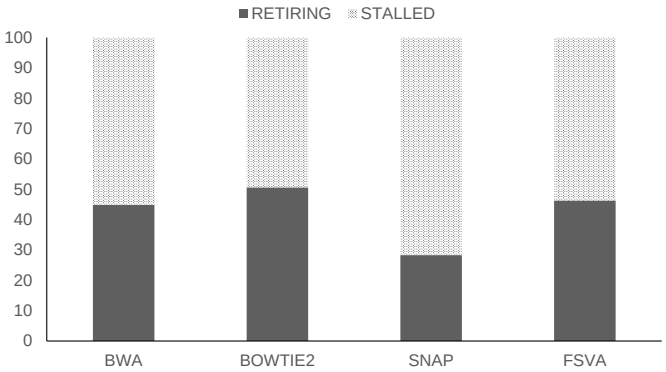

Fig. 13: Execution cycle breakdown under GIAB dataset

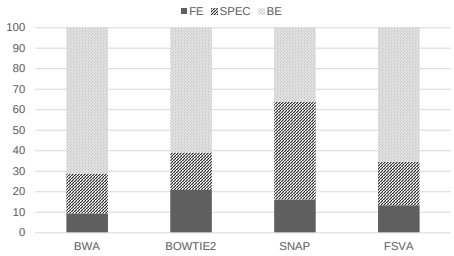

Fig. 14: Stall cycle breakdown for GIAB dataset

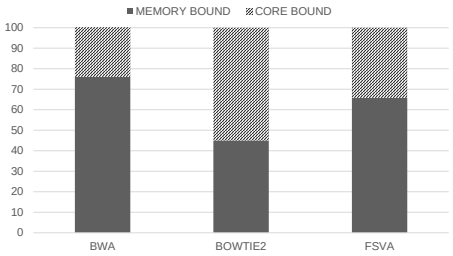

Fig. 15: Back-end stall breakdown for GIAB dataset

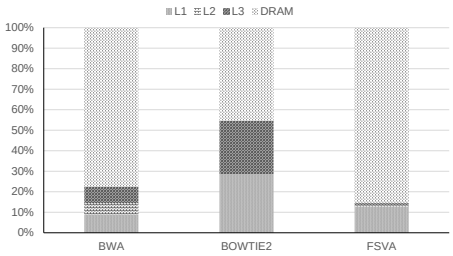

Fig. 16: Memory stall breakdown for GIAB dataset

accounts for 60% to 70% of all stall cycles. Second, while the contribution of speculation stalls increases marginally in the presence of indels under *BWA-MEM*, *Bowtie2*, and *FSVA*, we see that speculation emerges as the dominating source of stalls under *Snap* accounting for 53% of stall cycles. Back-end stalls contribute oto 28% only with *Snap*.

Figure 10 breaks down the back-end stalls further into memory or core bound for *Bowtie2*, *BWA-MEM*, and *FSVA*. Comparing this with the indel-free case (Figure 5), we see that *BWA-MEM* and *Bowtie2* behave similarly at the microarchitectural level irrespective of the presence or absence of indels. However, with *FSVA*, memory stalls are no longer the dominating source in the presence of indels as core stalls account for 60% of stalls. This indicates the presence of different code paths in *FSVA* for dealing with indel-free and indel-based cases, and the existence of complex branching logic in the extension stage of both these aligners that is activated only under datasets with indels.

Memory stalls analysis. Given that memory stalls still account for 40–76% across aligners, Figure 11 breaks down memory stalls across various cache levels and DRAM under the with-indel dataset. Comparing Figures 6, 11, we see that under *BWA-MEM* and *FSVA*, long-latency DRAM stalls continue to dominate and contribute to over 80% of all memory stalls in both indel-free and with-indel datasets.

Insights. *While processor utilization improves in the presence of indels across all aligners, the processor still spends majority of its cycles in the stalled state. Memory stalls are no longer the majority contributor, as the presence of indels increases speculation stalls and core stalls across aligners.*

C. GIAB dataset analysis

So far, we have presented our analysis based on the simulated datasets. The microarchitectural behavior of aligners under the GIAB dataset is very similar to the simulated, with-indel dataset.

Table I shows the single-threaded execution time of aligners under the GIAB dataset. Figures 12, 13 show the IPC and execution cycle breakdown. Comparing this with Figure 8, we see similar trends between the with-indel simulated dataset and the GIAB dataset as the processor is stalled in 50-70% of execution cycles.

Figure 14 breaks down the stall cycles into various components. Similar to the simulated dataset (Figure 8), back-end stalls account for 60–70% of stall cycles under *Bowtie2*, *BWA-MEM*, and *FSVA*. Speculation stalls are the dominating source of stall cycles under *Snap*.

Among aligners bottlenecked on back-end stalls, Figure 15 shows that memory stalls are the dominating factor. Figure 16 shows that long-latency DRAM stalls are the main contributors for memory stalls.

V. Per-stage analysis

The analysis presented so far answers some questions–how do aligners utilize the processor? what causes processors to be stalled? But, it also raises other questions–why do faster aligners have lower IPC than slower ones? why does the presence of indels change the microarchitectural behavior? To

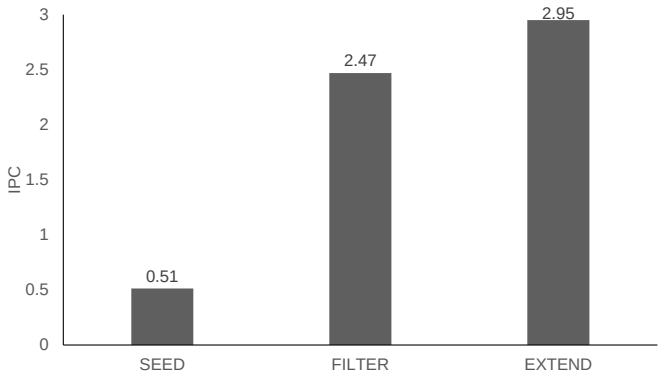

Fig. 17: Per-phase IPC under simulated with-indel dataset

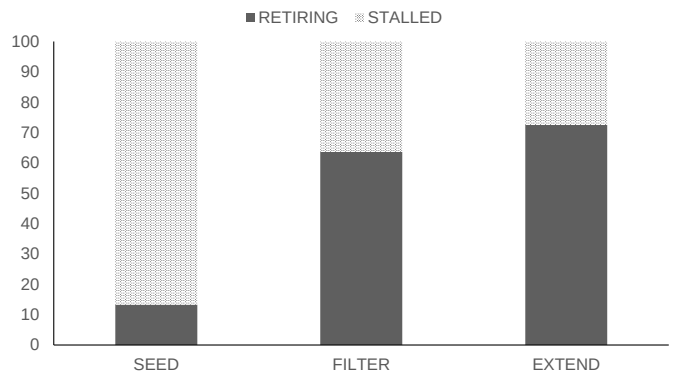

Fig. 18: Per-phase execution cycle breakdown

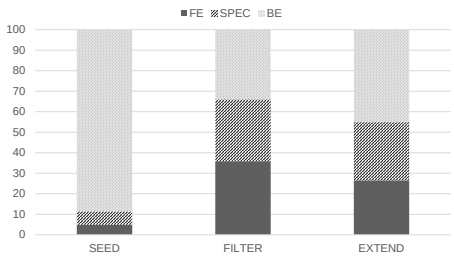

Fig. 19: Per-phase stall cycle breakdown for with-indel dataset

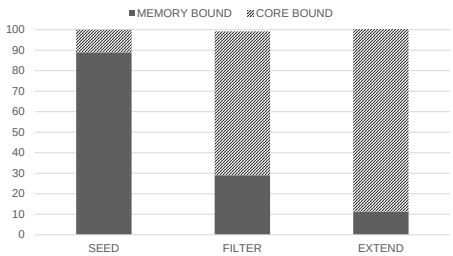

Fig. 20: Per-phase Back-end stall breakdown for with-indel dataset

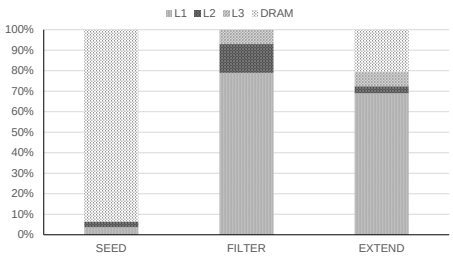

Fig. 21: Per-phase memory stall breakdown for with-indel dataset

answer these questions, we need to perform a stage-by-stage analysis of sequence alignment.

All aligners we have considered in this study work by considering one read at a time. Each aligner processes each read using three distinct steps. In the first stage, seeds are extracted from each read and used to lookup the index to retrieve candidate locations. The second stage is the filtration stage where heuristics and theoretical lower bounds are used to reduce the number of candidate locations that must be examined. The third stage is the extension stage where the entire read is aligned with each candidate location to identify the best match. Once a read has been processed by all the three stages, the aligner writes the alignment information to the output file and moves to the next read.

The analysis results we have presented so far are from an inter-stage analysis that spans across all stages. In this section, we present intra-stage analysis where we explore processor utilization within each stage to identify which stages account for stall cycles so that we can answer the aforementioned questions. In order to perform the intra-stage analysis, we modified *FSVA* so that data is processed in a stage-at-a-time fashion. We chose *FSVA* as it is was the fastest aligner in our study and has a simpler code base due to its focus on single-threaded performance.

In our modified *FSVA*, which we henceforth refer to as *staged-FSVA*, all reads are first broken down into seeds and a hashtable lookup is performed to identify candidate locations. All such locations are saved in intermediate data structures together with metadata to keep track of the mapping between locations and reads. Once all reads are processed by the first stage, the output from the first stage is passed to the second stage. In this stage, the candidate locations are sorted and filtered using the seed-and-vote filtration approach used by FSVA [15]. The output of this stage are two candidate locations that are the two most voted locations for each read. Once these location pairs are identified for all reads, staged-FSVA proceeds to the third stage where Smith-Waterman alignment is used to perform approximate alignment.

By using a stage-at-a-time execution approach, staged-FSVA makes it possible for us to accurately measure and profile each stage independently.

A. IPC and execution cycle breakdown

Stage	Time (secs)
Seeding and lookup	14
Filtration and voting	11
Extension	35
Misc.	22

TABLE II: Execution time in seconds of the three main alignment stages and rest (loading reference, I/O, writing out SAM file, etcetera) of FSVA

Table II shows a breakdown of execution time of each stage of staged-FSVA under the with-indel simulated dataset. As expected, the Smith-Waterman extension stage dominates overall execution time. Figure 17 shows the IPC values for each phase of staged-FSVA. Whlie the filtration and extension stages have IPC values of 2.5 and 2.9, the seeding and

hashtable lookup stage has only an IPC value of 0.5. This highlights that the main bottleneck leading to low IPC count, and hence poor processor utilization, is the first stage.

Figure 18 shows the breakdown of execution cycles per stage for staged-FSVA. These results mirror the IPC values in Figure 17. In the first stage, the processor spends 87% of execution cycles in the stalled state, while retiring instructions only in 13% of cycles. In the second and third stages, this trend is reversed, as the processor spends 63% to 72% of cycles retiring instructions.

B. Stall cycle breakdown

Figure 19 shows the breakdown of stall cycles for each stage. Clearly, there is a dramatic difference between stages. In the seeding and lookup stage, Back-end stalls account for 90% of stall cycles. Figure 20 shows the breakdown of Back-end stalls and as can be seen, memory stalls contribute to over 90% of the Back-end stalls in the seeding stage. Figure 21 shows the memory stall breakdown. Clearly, long-latency DRAM access dominate this category and account for 93% of all memory stalls in the seeding stage.

During this first stage of alignment, seeds of length 31 bp are extracted from each read, hashed, and used to probe the hashtable to determine coordinates in the reference. As each seed potentially hashes to a completely random value in a 4GB range, there is little spatial locality in this workload. Thus, hashtable lookups do not benefit from processor caches, the corresponding memory load instructions result in long-latency DRAM accesses. As instructions that follow these memory loads are dependent on the load, the pipeline is stalled and the processor idles waiting for data to arrive from memory.

Unlike the seeding stage, the filtration stage is not entirely bottlenecked on the backend. During filtration, Front-end, speculation, and Back-end stalls contribute equally to stall cycles as shown in Figure 20. *FSVA* uses a seed-and-vote-based filtration scheme where all coordinates gathered are used to vote for candidate locations where the read must be aligned. This voting is accomplished by sorting the coordinates determined in the seeding stage, eliminating duplicates and low-frequency locations, and identifying the top two locations with most votes. The branching logic in the sorting algorithm results in speculation stalls. Mispredicted branches add delay to the Front-End as it has to fetch operations from corrected path, resulting in Front-end stalls. The Back-end stall breakdown shown in Figure 20 shows that core stalls are the major (71%) contributor to Back-end stalls. Figure 21 shows that the remaining 29% contribution from memory stalls is due to L1 cache accesses and not due to DRAM access. Thus, computation rather than memory access is the bottleneck in the filtration stage.

The extension stages is similar to the filtration stage microarchitecturally. As we already mentioned, the processor is stalled in only 30% of cycles (Figure 18). As shown in Figure 19, Back-end stalls account for 40% of execution cycles while Front-end and speculation account evenly for the other 60%. While Back-end stalls are relatively higher in the extension stage compared to the filtration stage, these stalls are once again due to core stalls, which contribute to nearly 90% of Back-end stalls, rather than memory stalls as shown in Figure 20. During the extension stage, *FSVA* uses the Smith Waterman dynamic programming algoritm for aligning reads to the reference. As the amount of data that the algorithm operates on fits easily in the processor cache, there are very few data misses. However, the complex control flow and branching logic in the dynamic programming algorithm creates speculation and core stalls.

Insights summary. *There is a clear dichotomy with respect to microarchitectural utilization of various stages of sequence alignment. The seeding stage exhibits very low processor utilization with memory stalls in the Back-end contributing to majority of processor stall cycles. The filtration and extension stage, in contrast, have much better utilization, and are bottlenecked on Front-end speculation stalls and Back-end core stalls.*

VI. Implications

In this section, we will discuss the implications of our findings on the design of next-generation of sequence aligners. We will consider three dimensions, namely, performance, scalability, and energy efficiency.

A. Performance implications

Even though modern sequence aligners are able to map reads to reference at a very high throughput, our analysis reveals that they still substantially underutilize the processor. Irrespective of the aligner used, the processor remains stalled over 50% of time suggesting that there is room for further improvement. Our analysis also showed that different stages of sequence alignment behave differently with respect to processor utilization. The seeding stage has the worst utilization of all stages as memory stalls caused by hashtable lookups dominate execution cycles. The filtration and extension stages, in contrast, utilize the processor better and do not experience such memory stalls.

Given this dichotomy between stages, it is clear that the two stages should be optimized differently. Given that the filtration and extension stages are bottlenecked on resource conflicts (core stalls), using "beefier" processors with more execution units, or using software techniques like vectorization to feed more data to existing units, should both assist in improving performance of these two stages. However, given that the seeding stage is bottlenecked on memory stalls, simply using faster processors will only result in an increase in stall cycles instead of improved performance. The solution is to use latency-hiding techniques for masking the overhead of memory accesses. Thus, techniques like software prefetching, simultaneous multithreading, can be explored further to ensure that the processor continues to retire instructions corresponding to one read while waiting for data to arrive from memory for another read.

B. Scalability implications

Modern sequencing technologies produce millions of reads in a single run. Given that each read is independent of another, scaling sequence alignment is an embarrassingly parallel problem as each read can be assigned to a different processing thread. Given that modern servers are equipped with multicore processors, state-of-the-art aligners have started exploiting the thread-level parallelism of these processors to scale alignment. However, given that a single processor is stalled 50% of cycles, and given that various stages of alignment differ in their usage of processor resources, such an approach of using homogeneous multiprocessing where processors is identical to another will only aggravate underutilization.

A promising alternative is to consider the use of heterogeneous parallelism using accelerators like Xeon Phi or GPGPUs. However, although GPGPUs provide massive thread-level parallelism with thousands of CUDA cores, research has shown that current GPGPU-based aligners provide only around 2–3× improvement compared to CPU-based aligners [17], [19]. Unlike CPUs which excel task parallelism, GPGPUs excel at data parallelism. Sequence alignment, however, lends itself naturally to task parallelism rather than data parallelism due to the complex branching logic used in dynamic programming-based extension algorithms. Thus, current GPGPU-based aligners that attempt to execute the extension stage on GPGPUs suffer from scalability limitations due to warp divergence caused by branching logic.

Our analysis suggests a natural division between CPUs and GPGPUs. CPUs are underutilized substantially during the seeding phase due to long-latency data misses. GPGPUs, in contrast, are capable of hiding long-latency accesses using hardware-assisted multithreading. Thus, GPGPUs are a natural fit for the seeding phase of sequence alignment. Similarly, filtration phase of sequence alignment uses sorting, duplicate removal, and candidate selection. As these operations are data parallel, they also likely to benefit by execution on GPGPUs. However, given that the extension stage uses complex branching logic, and given that CPU utilization is already high during this stage, it might be better to schedule it on CPUs instead of GPGPUs. Thus, further research is necessary to understand the pros and cons of such a design as opposed to a CPU-only or GPU-only aligner.

C. Energy efficiency implications

With the advent of cloud computing, it has become increasingly more important for data analytics platforms to be energy proportional [3], meaning that they consume power proportional to the amount of work performed. Any program that results in the processor stalling for a substantial portion of time adversely impacts energy proportionality as the power consumed by the processor is not used for performing useful work. Given that the processor is stalled for over 50% of the execution cycles under sequence aligners, we believe that there is much work to be done in improving the energy efficiency of these tools.

One promising research direction would be to use the dichotomy between stages to implement sequence alignment on heterogeneous big-LITTLE processor architectures like ARM. As processors are equipped with both "beefy" cores and "wimpy" cores, a sequence aligner that uses the latter for the first stage and former for the latter two stages would consume much less power than one built for contemporary server-grade processors.

VII. Conclusion

Sequence alignment is the first stage of genomic data analysis and a very well-studied problem. Decades of research on scalable, high-performance approximate string matching algorithms have led to the development of fast sequence aligners that can map thousands of reads per second. However, there is no work that analyzes the efficiency of sequence aligners with respect to processor utilization.

In this study, we presented the first microarchitectural analysis of four state-of-the-art aligners. Our analysis on simulated datasets as well GIAB data revealed that all aligners result in substantial underutilization as the processor remains stalled for 50%-70% of execution cycles. We identified back-end memory stalls and speculation stalls as leading sources of inefficiency. To understand the source of these stalls, we also presented a stage-by-stage analysis which mapped memory stalls to the seeding stage and speculation stalls to the filtration and extension stages. This microarchitectural study shows an in-depth view of processor usage for one of the many steps in genomic data analysis pipeline and opens up new opportunities for extending this analysis to other phases as well. Given the growing popularity of heterogeneous parallelism, such microarchitectural studies will play an important role in determining the ideal processor type for each phase of the genomic data analysis pipeline.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proc. of the 25th Intl. Conf. on Very Large Data Bases*, 1999.

[2] L. B and S. SL. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4), 2012.

[3] L. A. Barroso, J. Clidaras, and U. Hoelzle. *The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines*. 2013.

[4] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[5] S. Canzar and S. L. Salzberg. Short read mapping: An algorithmic tour. *Proceedings of the IEEE*, 105(3), 2017.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[7] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4), 2005.

[8] N. A. Fonseca, J. Rung, A. Brazma, and J. C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics*, 28(24), 2012.

[9] L. H and H. N. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5), 2010.

[10] A. Hatem, D. Bozda, A. E. Toland, and m. V. atalyrek. Benchmarking short sequence mapping tools. *BMC Bioinformatics*, 14(184), 2013.

[11] 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), Jan 2012.

[12] 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), Mar 2009.

[13] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *ArXiv e-prints*, 2013.

[14] H. Li and R. Durbin. Fast and accurate short read alignment with burrowswheeler transform. *Bioinformatics*, 25(14), 2009.

[15] S. Liu, Y. Wang, and F. Wang. A fast read alignment method based on seed-and-vote for next generation sequencing. *BMC Bioinformatics*, 17(17), 2016.

[16] Y. Liu and B. Schmidt. Cushaw2-gpu: Empowering faster gapped short-read alignment using gpu computing. *IEEE Design Test*, 31(1), 2014.

[17] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, H.-F. Ting, S.-M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T.-W. Lam. Soap3-dp: Fast, accurate and sensitive gpu-based short read aligner. *PLOS ONE*, 8, 05 2013.

[18] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13, 07 2015.

[19] R. Wilton, T. Budavari, L. B., W. S.J., S. S.L., and S. A.S. Arioc: high-throughput read alignment with gpu-accelerated exploration of the seed-and-extend search space. *PeerJ*, 808, March 2015.

[20] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[21] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and more accurate sequence alignment with SNAP. *CoRR*, abs/1111.5572, 2011.

[22] J. M. Zook, D. Catoe, J. McDaniel, L. Vang, N. Spies, A. Sidow, Z. Weng, Y. Liu, C. Mason, N. Alexander, D. Chandramohan, E. Henaff, F. Chen, E. Jaeger, A. Moshrefi, K. Pham, W. Stedman, T. Liang, M. Saghbini, Z. Dzakula, A. Hastie, H. Cao, G. Deikus, E. Schadt, R. Sebra, A. Bashir, R. M. Truty, C. C. Chang, N. Gulbahce, K. Zhao, S. Ghosh, F. Hyland, Y. Fu, M. Chaisson, J. Trow, C. Xiao, S. T. Sherry, A. W. Zaranek, M. Ball, J. Bobe, P. Estep, G. M. Church, P. Marks, S. Kyriazopoulou-Panagiotopoulou, G. Zheng, M. Schnall-Levin, H. S. Ordonez, P. A. Mudivarti, K. Giorda, and M. G. Salit. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific Data*, 3(160025), 2015.