Improving performance of BWA alignment of short sequences with coroutines

David Jonsson
Abstract

Improving performance of BWA alignment of short sequences with coroutines

David Jonsson

For the Burrows-Wheeler Aligner (BWA), previous experimentation has shown that alignment of short sequences (reads) can benefit from increased instruction level parallelism (ILP) using hyperthreading. This thesis investigates and demonstrates that one can increase the performance of short read alignment with BWA without hyperthreading enabled, by employing coroutines in order to increase ILP. Analyzing performance when aligning short reads shows that stalls are concentrated to a specific part of the codebase, namely the functions that are used to give an interval of occurrences using the Burrows-Wheeler Transform. By interleaving the BWT lookups from the processing of several separate reads with coroutines, a speedup of 1.28 was achieved compared to an optimized version of BWA, or 1.51 compared to the original version.
## Contents

1 Introduction 2

2 Related work 2

3 Background 3
   3.1 CPU architecture ........................................ 3
   3.2 Fibers / Coroutines .................................... 4
   3.3 Burrows-Wheeler Aligner ............................... 4

4 Interleaving lookups on Burrows Wheeler Transforms with coroutines 6
   4.1 Data selection ........................................... 6
   4.2 Identifying bottlenecks .................................. 7
   4.3 Implementation ......................................... 8
   4.4 Results .................................................. 12

5 Discussion 13

6 Future work 14


1 Introduction

Burrows-Wheeler Aligner[8][9][10] is a software tool, consisting of three algorithms used for mapping low–divergent genome sequences, also called reads, to some known reference genome. The number of base pairs that makes up a sequence determines which algorithm should be used, and in the case of shorter sequences (up to 70 bp), BWA-backtrack is preferred. When mapping a sequence, lookups are done on the Burrows-Wheeler transform of the reference genome. This offers some challenges, particularly when mapping shorter reads. This is because when lookups are done on the BWT, memory is in essence accessed randomly, which results in cache misses and thus stalling of the process. A contributing factor is that the CPU is not able to speculate because each successive lookup will be dependent on the results of previous lookups, thus the possibility for efficient out-of-order execution will be limited.

This thesis will describe how we employ an optimization technique that runs lookups concurrently by interleaving a number of coroutines, in an attempt to avoid stalls caused by caches misses related to lookups on the BWT. In chapter 3, the Burrows-Wheeler methodology, the algorithms to search the Burrows-Wheeler transform, and the differences and similarities between execution in threads, fibers, and coroutines are discussed.

2 Related work

In [18] the authors proposed a solution for hiding main memory latency for processors with in-order microarchitectures, by applying compilation techniques that reorders and clusters memory accesses in access-phases, aiding the CPU in increasing memory level parallelism.

In [16] an approach to hiding stalls in index joins in a database was suggested, using coroutines. In a similar fashion, accesses to the databases were coalesced to a prefetch phase, by means of employing a number of coroutines to fetch data in parallel. In the next stage index joins were performed on the fetched data.

Previous experimentation [13] has been done on BWA with hyperthreading. Hyperthreading is an Intel technology that allows 2 threads to simultaneously execute on the same physical processor, sharing the same resources [12]. A thread is ”the smallest unit of execution schedulable by an operating system’s process scheduler”[11]. This experiment did provide a significant boost in performance when aligning short sequences with BWA. This indicated that running lookups in parallel is beneficial for performance. It is reminiscent of loop unrolling in that the effects exposes more instruction level parallelism, though in contrast to loop unrolling, hyper threading exposes the parallelism on a hardware level. In this case the complexity of the loop makes loop unrolling an unattractive approach. However it would be desirable to look for alternative ways to expose this parallelism for CPUs that do not have hyperthreading, which is what is done in this thesis.
3 Background

3.1 CPU architecture

Despite substantial progress in main memory storage in recent years in regards to storage capacity and bandwidth, it remains one of the major bottlenecks on modern computers because of the latency between main memory and the CPU [1]. This can be alleviated by the cache, a type of smaller but much faster memory. The cache is often divided up in 2 or 3 layers, in which the level 1 (L1) being the smallest and fastest. When some data that the CPU accesses is not in the cache, it is fetched from main memory and into the cache, until it is evicted due to a cache conflict or the need to load another line. If the data is in the cache, CPU can access this data very quickly, and thus the latency bottleneck is avoided.

A CPU instruction is said to have a data dependency if it relies on data created by another instruction. A data dependent instruction cannot be executed before this dependency is resolved. Many modern processors have strategies to execute instructions out of order, so that other instructions that do not have a dependency or are ready to execute gets executed, while the one with a dependency waits. This is all well and good, but the processor might run out of instructions that are ready while it is waiting for some dependency to be resolved. Thus, the processor has to stall. One can see that one would like to access the main memory as little as possible, and try to arrange instructions without tight data dependencies, in order to avoid stalls.[20]

Another feature of modern processors is speculative execution, which makes the CPU speculate on the outcome of some instruction which other instructions depend on [14]. This enables the execution of the dependant instruction before the dependency is actually resolved. This functionality is not necessarily exclusive to hardware, but can also be implemented as part of a compiler. Speculative execution requires some form of mechanism for evaluating if the speculation is still valid after the dependency has been resolved. If the speculation is not valid, then any instruction with the dependency in question needs to be revoked and executed again. Some forms of speculation involves guessing which path a branch instruction might take, where a branch is the result of some conditional statement that affects the execution path of a program, like an if-statement. This is called branch prediction, and involves hardware logic which keeps track of historically if a specific branch has been taken or not. Another way to speculate is prefetching, where the address to some data is not known but the CPU fetches the data from some speculated address long before it is actually used. This form of speculation differentiates itself from branch prediction in that it does not involve any instructions that potentially needs to be revoked if the prediction is wrong. The only side effect of a misprediction in this case is that another access to main memory is necessary, and the possibility of a stall.
3.2 Fibers / Coroutines

Fibers and coroutines are non-preemptive lightweight units of execution, as opposed to preemptively scheduled threads, which are managed by the operating system scheduler. Instead, for coroutines and fibers, cooperative scheduling is employed, which means that a fiber or coroutine may yield. Yielding means that the currently executing unit of execution stops executing. At the same time, a unit of execution may await. Await also stops execution for the unit calling await, furthermore the execution unit passed as argument to await will then start executing. When that unit then yields, execution is then returned to the one that called await in the first place. As seen in the simple example that has just been presented, no scheduler is involved in the scheduling of these units of execution. That is the cooperative nature of this type of scheduling. Another thing to note is that an execution unit that yields, similarly to returning from a regular function, may return information to the caller, i.e. the unit that has called await. Also, these types of execution units may return execution, which means the unit is finished executing. This in contrast to yielding means that resuming execution at a later stage is not possible. [11]

The boost library provides a fiber implementation for C++, which provides functionality for spawning fibers inside of threads, and these fibers are cooperatively scheduled within the thread that spawned them by yielding. Each fiber has its own state including CPU context, stack pointer etc. that can be saved and restored. A context switch in this context is about 1 order of magnitude faster than a regular context switch. [7]

The Coroutines Technical Specification [6] proposed ”the addition of Coroutines and related keywords and std utilities” into the C++ standard, and parts of it was incorporated into the C++20 working draft [2]. A major difference between C++ coroutines and Boost fibers is that the former are stackless and handled at compiletime. Consequently, state shifts can be even more lightweight and the compiler can perform optimization operations across coroutine boundaries, which is not possible for fibers with only library support. As of now, a few compilers do have support for coroutines, among them clang-11 which was the compiler used in this experiment. However, at this point, it is a daunting task to use the C++ coroutine support directly. Since only the most basic primitives are featured, the programmer has to implement library support for practical use of coroutines. Thankfully, there exists libraries like CppCoro [3] that provide a convenient framework for creating a handful of different types of coroutines. One of particular interest is the generator type, which requires the use of the co_yield keyword.

3.3 Burrows-Wheeler Aligner

Burrows-Wheeler Aligner [10] is a tool that uses the Burrows-Wheeler transform to align sequence reads to a reference genome. The Burrows-Wheeler transform was originally developed as a compression method, but does have some properties that also makes it suitable for searching.
To create the transform, the following steps should be performed:

- Create an array of all substrings of the original string
- Sort the array of substrings in lexicographical order
- Concatenate the index of each substring in the unsorted array to create a suffix array.
- Concatenate the last character of each substring in the sorted order to create the Burrows Wheeler Transform.
- Map the last character in the of each string to the index of the substring where it appears as the first character (Last-First mapping[5]).

Figure 1 gives an example of this procedure. The transform is an array of characters, where identical characters tend to stick together, which makes it good for compression. The Last-First mapping is exploited to find occurrences of some sequence of characters in the original string. The search is reminiscent of binary search in that the search is done in steps, where each step is dependant on the previous step, and this makes speculation problematic for the CPU. Furthermore, each step essentially accesses a random memory location within the transform data structure. If the data structure is significantly larger than the cache size, each access will most likely incur a cache miss and possibly a stall. Figure 2 details how searching for substrings in the Burrow Wheelers Transform is performed.

When occurrences are calculated, two searches are done in parallel to determine an upper and lower bound. The algorithm for finding the occurrences of exact matches was described in [5], and is shown in algorithm 1. The procedure described in Figure 2 roughly resembles this algorithm. Some variables (K and
Figure 2: Searching for the substring goog in the string googol. The first step is finding the interval of all the occurrences of the last character in the substring in F. Next step is finding the interval in L where the characters match the previous character in the substring. Then find that corresponding interval in F, which is obtained by last-first mapping. This process is repeated until an interval in F of the first character in the substring is obtained. The corresponding interval in the suffix array will give all the positions of the substring in the original string.

L) were renamed to comply with the algorithms described in [10]. K and L represent the upper and lower bound, respectively, of the interval containing the occurrences of a substring. The array C requires some explanation. C[c] contains the number of occurrences of all characters that are lexicographically smaller than c, while C[c] + 1 is equal to the first occurrence of c in column F, detailed in Figure 1. Furthermore, "the subroutine Occ(c, 1, k) ... reports the number of occurrences of the character c in BWT[1, k]; hence it is employed to implement the LF-mapping"[10].

4 Interleaving lookups on Burrows Wheeler Transforms with coroutines

4.1 Data selection

The National Center for Biotechnology provides a sequence read archive, including for Drosophila melanogaster (common fruit fly) [19]. The Drosophila genome can be found at [4]. The main consideration here is that the BWT is big enough that it does not fit in cache memory. Since the transform is 160.9 MiB this is not an issue. The sequence archive consists of 32.6 k reads, and each sequence has a length of 152 base pairs. Fastq-dump [17] was used to convert the reads to a format that BWA can process.

The strongest case for using coroutines in BWA is when aligning short sequences, because the matching of sequences is dominated by the fetching of data from memory. Therefore the sequence reads were cut to 40 base pairs. The dataset was duplicated to around 1 million sequences to make the initial-
Algorithm 1
Coroutines implementation of Match Exact
Gives an exact interval of occurrences of P in C
W: a query string
w: the length of W

procedure Match Exact(W[1, w])
    c ← W[w]
    i ← w
    k ← C[c] + 1
    l ← C[c + 1]

    while k ≤ l and i ≥ 2 do
        c ← W[i – 1]
        k ← C[c] + Occ(c, 1, k – 1) + 1
        l ← C[c] + Occ(c, 1, l)
        i ← i – 1
    end while

    if l < k then
        return "notfound"
    else
        return "found(l – k + 1)"
    end if
end procedure

Figure 3: Calls to occ

ization of coroutines and BWA itself negligible.

4.2 Identifying bottlenecks

To identify bottlenecks in BWA when aligning sequences like the ones described in the previous chapter, Perf [15] was used. Perf is a profiling tool for linux, that counts hardware events specified by the user when running some program. Stalls and cache misses were measured; these measurements identified the 4 variations of the Occ function to be major bottlenecks, producing ~ 89% of all stalls. Table 1 details the results from 1 profiling session of BWA with Perf. Several profilings were done to get a stable result. Figure 3 illustrates a callgraph indicating the locations of calls to the occ family of functions.
### 4.3 Implementation

Threading was not used in this implementation, mainly because the purpose of this thesis is to achieve similar performance gains to hyperthreading by using coroutines. Therefore, all coroutines are executed in a single thread.

**Algorithm 2**

Coroutine implementation of Reg_Gap
Acts as a generator for other coroutines
S: a set of query strings
z: nr of tolerable mismatches
next: index of the next string to process
X: is the reference string

1: procedure REG_GAP(S, z, next)
2:     while next < |S| + 1 do
3:         P ← S[next]
4:         next ← next + 1
5:         cal_width ← CAL_WIDTH(P)
6:         match_gap ← MATCH_GAP(P, |P|−1, z, 1, |X|−1)
7:     while cal_width = 0 do
8:         co_await cal_width
9:         co_yield 0
10:    end while
11:    while match_gap = 0 do
12:        co_await match_gap
13:        co_yield 0
14:    end while
15: end while
16: co_yield "Done"
17: end procedure

By studying how the calls to occ were made, as illustrated in Figure 3, it was concluded that one approach to incorporate coroutines in BWA was by a hierarchy of coroutines. Coroutines would spawn new coroutines according to Figure 3, where Reg_Gap, algorithm 2, would be the "base" generator. It is comparable to InexactSearch in [10], however in the coroutine version the code

<table>
<thead>
<tr>
<th></th>
<th>Cache misses</th>
<th>Stalls</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWA</td>
<td>276k</td>
<td>271k</td>
<td>68.14</td>
</tr>
<tr>
<td>bwt_occ</td>
<td>33.35 %</td>
<td>32.62 %</td>
<td>-</td>
</tr>
<tr>
<td>bwt_2occ</td>
<td>22.46 %</td>
<td>21.56 %</td>
<td>-</td>
</tr>
<tr>
<td>bwt_occ4</td>
<td>16.18 %</td>
<td>17.88 %</td>
<td>-</td>
</tr>
<tr>
<td>bwt_2occ4</td>
<td>14.19 %</td>
<td>17.44 %</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Perf results on original BWA
is captured within a while loop since it works on a batch of sequences instead of just one. `Reg_Gap` works at one sequence at a time, represented by $V$. A number of `Reg_Gap` coroutines would be processing sequences in parallel, coordinated by cooperative scheduling as described in the previous chapter. Instead of dividing the sequences evenly between each instance of `Reg_Gap`, the coroutines share an index variable that points to the next sequence to be processed. This is to avoid creating an unbalanced workload in case some coroutine get an disproportional amount of quickly processed sequences compared to other coroutines. Because threading is not used, and the coroutines are cooperatively scheduled within one thread, no synchronization is needed between the coroutines. As shown in algorithm 2, instances of `Cal_Width` and `Match_Gap` are spawned inside of the coroutine.

Algorithm 3
Coroutine implementation of `Cal_Width`
Gives a non-optimal bound for $k$ and $l$
$W$: a query string

1: procedure `Cal_Width`(W)
2:  $k \leftarrow 1$
3:  $l \leftarrow |X| - 1$
4:  $z \leftarrow 0$
5:  for $i := 0$ to $|W| - 1$ do
6:    prefetch($k$, $l$)
7:    co.yield $0$
8:    $k \leftarrow C(W[i]) + Occ(W[i], k - 1) + 1$
9:    $l \leftarrow C(W[i]) + Occ(W[i], l)$
10:   if $k > l$ then
11:      $k \leftarrow 1$
12:      $l \leftarrow |X| - 1$
13:      $z \leftarrow z + 1$
14:   end if
15:   D($i$) $\leftarrow z$
16: end for
17: co.yield $z$
18: end procedure

Algorithm 4
Coroutine implementation of `Match_Gap`
$W$: a query string
$i$: end of the query
$z$: nr of tolerable differences
$k$ and $l$: the interval of occurrences of $W[0:i]$

1: procedure `Match_Gap`(W,$i$,z,$k$,$l$)
2:   if $z < D(i)$ then
3:     co.yield $0$
4: end if

9
end if
if \( i < 0 \) then
c0 yield \([k, l]\)
end if
\[ I \leftarrow \emptyset \]
match\_gap \( \leftarrow \text{MATCH\_GAP}(W, i - 1, z - 1, k, l) \)
\[ \text{while} \text{ match\_gap} = 0 \text{ do} \]
c0 await match\_gap
c0 yield 0
\end while
\[ I \leftarrow I \cup \text{match\_gap} \]
for \( b \in A, C, G, T \) do
if no diff allowed then
match\_exact \( \leftarrow \text{MATCH\_EXACT}(W[1, i]) \)
\[ \text{while} \text{ match\_exact} = 0 \text{ do} \]
c0 await match\_exact
c0 yield 0
\end while
end if
prefetch\((k, l)\)
c0 yield 0
\[ k \leftarrow C(b) + \text{Occ}(b, k - 1) + 1 \]
\[ l \leftarrow C(b) + \text{Occ}(b, l) \]
if \( k \leq l \) then
match\_gap \( \leftarrow \text{MATCH\_GAP}(W, i - 1, z - 1, k, l) \)
\[ \text{while} \text{ match\_gap} = 0 \text{ do} \]
c0 await match\_gap
c0 yield 0
\end while
\[ I \leftarrow I \cup \text{match\_gap} \]
if \( b = W[i] \) then
match\_gap \( \leftarrow \text{MATCH\_GAP}(W, i - 1, z - 1, k, l) \)
\[ \text{while} \text{ match\_gap} = 0 \text{ do} \]
c0 await match\_gap
c0 yield 0
\end while
\[ I \leftarrow I \cup \text{match\_gap} \]
else
match\_gap \( \leftarrow \text{MATCH\_GAP}(W, i - 1, z - 1, k, l) \)
\[ \text{while} \text{ match\_gap} = 0 \text{ do} \]
c0 await match\_gap
c0 yield 0
\end while
\[ I \leftarrow I \cup \text{match\_gap} \]
end if
end if
Algorithm 5 Match_Exact
Coroutine implementation of algorithm 1

1: procedure MATCH_EXACT(W[1, w])
2: c ← W[w]
3: i ← w
4: k ← C[c] + 1
5: l ← C[c + 1]
6: while k ≤ l do
7:   prefetch(k, l)
8:   co_yield 0
9: c ← W[w - 1]
10: k ← C[c] + Occ(c, 1, k - 1) + 1
11: l ← C[c] + Occ(c, 1, l)
12: i ← i - 1
13: end while
14: if ep < sp then
15:   return "notfound"
16: else
17:   return "found(ep – sp + 1)"
18: end if
19: end procedure

Cal_Width, algorithm 3, is the coroutine variant of CalculateD in [10], which gives an "gives a better, though not optimal bound" for k and l. Match_Gap, algorithm 4, corresponds to InexRecur[10]. Its purpose is to return an interval, if any, in the suffix array that matches the query string with at most \( z \) differences. As explained in that paper, the pseudocode is defined as an recursive algorithm, but in reality it is implemented with a heap structure, as is the coroutine version. Inside of Cal_Width, instances of Match_Exact are spawned, and this coroutine is the equivalent of Bw Search in [5] and algorithm 1.

Algorithm 6 prefetch

1: procedure PREFETCH(k, l)
2: k ← k - 1 - (k ≥ bwt.primary)
3: l ← l - (l ≥ bwt.primary)
4: builtin_prefetch(bwt_occ_intv(bwt, k))
5: builtin_prefetch(bwt_occ_intv(bwt, l))
6: return
7: end procedure
In any place where a coroutine calls `occ`, we prepend a call to `prefetch`, followed by the coroutine yielding, before the actual call to `occ`. The prefetch step is described in algorithm 6. It basically converts \( k \) and \( l \) to pointers before prefetching the data. In the pseudocode only 1 cacheline for each of the pointers are fetched, but in reality we tested prefetching a few of the subsequent cachelines to see how this affected performance.

To get a better understanding of how the coroutines are interleaved, think of the four coroutines mentioned above as a part of an instruction stream. At some point the stream will yield its execution, but before that, it will prefetch some data. When execution is resumed, the data (which hopefully is in the cache by this point) will then be processed. Then the process repeats. Figure 4 (based on Figure 2 in [16]) illustrates how 3 of these streams are interleaved, however this is an ideal scenario where no cache misses occur. Also, any penalty for yielding or continuing execution is disregarded in this figure.

To really enjoy the benefits of coroutines it is important to inline aggressively so that the compiler can optimize as much as possible. Therefore the roof for inline function size was raised to insure that all coroutines and all calls to the `occ` functions were inlined. Clang-11 where used to compile the library. The `-fno-exceptions` flag was also used, since in this implementation incrementing the iterators should theoretically never throw an exception. In either case, any exceptions thrown would have been ignored since BWA is written in C and C does not have support for exception handling.

The same heavily inlined code that was used for the coroutines, but stripped of the coroutine code, was used to implement a version of BWA comparable to our solution. This version was then used as a baseline reference for comparing our results with. This was done so that the importance of coroutines was not confused with other optimizations that the inlined code made possible for the compiler.

### 4.4 Results

These tests where done on a Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz with L1d = 32 KiB. Each test where done 10 times to make factors like CPU turbo even out, and the best and worst runtimes were selected. The running time results are shown in table 2. Only the results for 3 prefetched cache lines for the unoptimized version of the algorithm is presented here, since the results for 1 or 2 fetched cachelines are identical. However, prefetching the first cacheline
do actually do something for performance for the original algorithm, as with no prefetching, running time is $\sim 64$ seconds.

Table 3 presents the number of stalls and cache misses for each combination of nr coroutines and number of fetched cache lines. Examination of the compiled code revealed that vestiges of exception handling were still present in the generated code from CppCoro.

### 5 Discussion

The results show that for our dataset, we need to fetch 2 cache lines in order to hide stalls effectively. Furthermore, 3 coroutines seems to perform best, but this would probably change if longer sequences were being matched to the bwt. We get $\frac{57.77}{45.09} \sim 1.28$ speed up with these settings, meaning that we can process $\sim 1.28$ times more work compared to the optimized original algorithm. When compared to the unoptimized original algorithm, a speed up of $\frac{68.14}{45.09} \sim 1.51$ is achieved.

Investigating table 3 confirms that using coroutines, the numbers of stalls can be reduced when aligning short sequences with BWA. These numbers give a similar performance gain when compared to the runtime-speed ups. We get $\frac{189}{276} \sim 71\%$ stalls with 3 generators and 2 fetched cache lines compared to the optimized original BWA. Compared with the unoptimized version we get $\frac{189}{276} \sim 71\%$ stalls.

As a final note for this discussion, it should be pointed out that to make

<table>
<thead>
<tr>
<th></th>
<th>No prefetch</th>
<th>1 CL prefetch</th>
<th>2 CL prefetches</th>
<th>3 CL prefetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original BWA</td>
<td>68.14</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Optimized BWA</td>
<td>-</td>
<td>57.77 / 59.20</td>
<td>46.37 / 48.70</td>
<td>47.04 / 50.10</td>
</tr>
<tr>
<td>2 generators</td>
<td>-</td>
<td>61.20 / 61.51</td>
<td>45.09 / 45.33</td>
<td>46.53 / 49.74</td>
</tr>
<tr>
<td>3 generators</td>
<td>-</td>
<td>62.27 / 63.70</td>
<td>48.04 / 49.62</td>
<td>48.05 / 51.15</td>
</tr>
<tr>
<td>4 generators</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Best / worst run times, 1048576 sequences (sec), with 0, 1, 2 or 3 cache lines prefetched for k and l

<table>
<thead>
<tr>
<th></th>
<th>No prefetch</th>
<th>1 CL prefetch</th>
<th>2 CL prefetches</th>
<th>3 CL prefetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original BWA</td>
<td>276K / 271K</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Optimized BWA</td>
<td>-</td>
<td>229K / 224K</td>
<td>196K / 193K</td>
<td>213K / 209K</td>
</tr>
<tr>
<td>2 generators</td>
<td>-</td>
<td>232K / 228K</td>
<td>189K / 186K</td>
<td>199K / 195K</td>
</tr>
<tr>
<td>3 generators</td>
<td>-</td>
<td>240K / 237K</td>
<td>195K / 192K</td>
<td>203K / 199K</td>
</tr>
<tr>
<td>4 generators</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: Cache misses / stalls, 1048576 sequences (sec), with 0, 1, 2 or 3 cache lines prefetched for k and l
better use of the optimizations done in this project, the number of coroutines
could be increased by using multi-threading, where one thread would be running
a subset of 2-3 coroutines. Work could be distributed evenly between threads
or delegated by means of some simple synchronization mechanism.

6 Future work

Previous experimentation shows that hyper threading does improve performance
of BWA alignment of short sequences, with a speedup similar to what was
achieved in this experiment. However, since the previous experiment was done
on a different architecture, comparing them on an identical architecture would
provide further insight in the results provided by this study.

It should be investigated if the overhead introduced by coroutines could
be lowered by implementing a tailored framework, to avoid code related for
exception handling being generated. It has not been researched in this thesis
if there is further optimizations that could be done in regards to coroutines, so
this could be an area to investigate further.

References

[1] K Chang. “Understanding and Improving the Latency of DRAM-Based


05.22.2021).

Drosophila_melanogaster/dmel_r5.37_FB2011_05/fasta/dmel-all-

plications”. In: Proceedings 41st Annual Symposium on Foundations of

download/attachments/93291100/Kolb%5C20report%5C20-%5C20Coroutines%5C20in%5C20C%5C20B%5C20.pdf?version=1&
modificationDate=1532345204090&api=v2. (accessed: 06.23.2021).

//www.boost.org/doc/libs/1_76_0/libs/fiber/doc/html/fiber/

[8] H Li. “Aligning sequence reads, clone sequences and assembly contigs with


