The pros and cons of explicit software prefetching

March 31, 2024Low Level Performance, Memory Subsystem Performance, PerformanceLeave a Reply

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

This is the 16th post in the series about memory subsystem optimizations. You can also explore other posts in the same series here.

Introduction to Software Prefetching

Many computer architectures offer prefetching instructions. A prefetching instruction takes a memory address as an argument, and as a consequence it prefetches the cache line associated with that memory addresss to cache. The use case for this is to prefetch the data we plan to use later and improve the performance of our programs.

We call the approach of using prefetch instructions *software prefetching*. When it comes to software prefetching, there are two kinds. The first is *implicit software prefetching*, where the compiler automatically inserts software prefething instructions to speed up data fetching. For example, GCC has a compiler switch `-fprefetch-loop-arrays` that will emit prefetching instructions to speed up fetching of data inside loops. This option is not enabled by default at any optimization level, and needs to be enabled manually.

Another type of software prefetching is *explicit software prefetching*, where the developer uses prefetching primitives to prefetch the data they need in advance. On GCC and CLANG, prefetching is done using `__builtin_prefetch` builtin. A typical loop with explicit software prefetching would look like this:

```
for (size_t i = 0; i < n; i++) {
    __builtin_prefetch(&a[index[i + 8]]);
    a[index[i]]++;
}
```

WIth explicit software prefetching, you will often see similar code. Explicit prefetching prefetches N iterations in advance (in our case it is eight iterations). The number of iterations you prefetch in advance must be determined by measurement for optimal performance.

Explicit software prefetching is quite popular in the performance community, because you can improve the performance of your code by simply adding one line to your source code. But, in reality, the examples you see online are mostly cherry-picked, and in explicit software prefetching many times fails to deliver what is promised.

Table Of Contents

The Caveats of Explicit Software Prefetching

Getting explicit prefetching right is not an easy task. Let's go through some reasons why this is difficult.

Data Already in the Cache

As already said, prefetching will bring data from the memory to the fastest level of CPU cache. But what happens if the data is already in the cache? In that case, prefetching actually slows down your code instead of speeding it up. It wastes hardware resources without achieving anything.

The case where the data is already in the cache happens more often than you would expect. Imagine having a large binary tree, but you are only accessing a small part of it. Then the part you are accessing will get cached automatically, and access to it will be fast. There is no need for prefetching.

Later we will experiment with software prefetching to determine for what dataset sizes does it make sense to use it.

How Much in Advance to Prefetch?

Prefetching must be done in advance, but not too much in advance. If a prefetch request is issued too late, the data doesn’t arrive to the cache on time and we don’t take full advantage of prefetching. If a prefetch request is issued too soon, the data arrives to the cache but it is never consumed by the CPU before it is evicted from the cache.

Determining the exact number is performed experimentally. You typically start at some number, e.g. 16 iterations in advance and try out smaller and larger numbers. Bear in mind that this number is not necessarily a constant and depends on the dataset size. A dataset which is 4 MB in size fits the L3 data cache. A dataset which is 64 MB in size doesn’t fit the L3 data cache and needs to be prefetched from the memory. A prefetch request for a piece of data coming from the memory needs to be issued earlier than a prefetch request coming from the L3 cache. When calculating this number, you should take into account your typical dataset sizes and maybe even use a different constant for how much to prefetch in advance depending on the dataset size.

Instruction Dependencies

Instruction dependencies can also make explicit software prefetching less efficient or even unneeded. Consider the following loop:

```
double sum = 0.0;
for (size_t i = 0; i < n; i++) {
    double v = a[index[i]];
    sum += v;
}
```

In this loop, the addition operations on line 5 form a dependency chain because this statement depends on the same statement from the previous iteration (`sum += v` from iteration `X` depends on `sum += v` from iteration `X - 1`, etc.). But the load operations on line 4 don’t form a dependency chain, and a modern out-of-order CPU can execute them well in advance.

In this case, adding prefetches probably doesn’t make a lot of sense, because the bottleneck is the summation operation. The CPU can execute load operations (line 4) many iterations in advance, so prefetching will likely not be needed or it will have a smaller effect than expected.

Hardware Prefetching

An important hardware mechanism is hardware prefetching. Let’s say your program is accessing an array of floats from 0 to N, with an offset of X (X is typically 1, but can be any constant number). The hardware prefetcher will quickly pick up on the address pattern and start prefetching data before it is even needed.

In this case, software prefetching just slows things down because hardware prefetching works better and with less overhead. Loops that access data sequentially (from 0 to N with offset 1) or with a stride (from 0 to N with offset X) are typically bad candidates to be made faster through software prefetching.

Additionally, some CPUs (notably Apple’s M silicone) have data dependent prefetchers, where they can perform hardware prefetching on the array of pointers

```
for (size_t i = 0; i < N; i++) {
    object[i]->val = 0;
}
```

In this case, the hardware prefetcher figures out that `object` is an array of pointers, and that the pointers are dereferenced, and starts to prefetch the data pointed by the array of pointers. It roughly corresponds to the following explicit software prefetching:

```
for (size_t i = 0; i < N - 4; i++) {
    __builtin_prefetch(object[i + 4].val)
    object[i]->val = 0;
}
```

This type of hardware prefetching is slowly being introduced to CPUs. If a CPU has this type of prefetcher, then explicit software prefetching is probably not needed and makes things slower.

Unnecessary Prefetching

With explicit prefetching, you can prefetch data that evenutally don’t use. Look at the example of a binary search that uses prefetching:

```
while(low <= high) {
    mid = (low + high)/2;
    __builtin_prefetch (&array[(mid + 1 + high)/2], 0, 1);
    __builtin_prefetch (&array[(low + mid - 1)/2], 0, 1);
    if(array[mid] < key)
        low = mid + 1;
    else if(array[mid] == key)
        return mid;
    else if(array[mid] > key)
        high = mid-1;
}
```

In this case we use prefetching to prefetch one lookup in advance. At the moment of data prefetching, we don’t know if the binary search will continue in the left part of the array or the right part. That’s why we prefetch middle element of both the left and the right part. Only later we determine which part of the array is accessed. What this means is that out of two prefetch requests we made only one prefetch request is actually needed. But we needed to issue the prefetch requests early enough, otherwise prefetching wouldn’t work.

Although this kind of prefetching, where we prefetch data we do end up not using can improve performance, it will cause cache thrashing – unnecessary data is being brought into the data cache and this displaces useful data. This also decreases performance for other applications running on the same core or the same CPU by displacing their useful data. We covered this in the post about frugal programming.

Prefetching in More Detail

When to Prefetch?

This question doesn’t have a definitive answer, and experimentation is needed. But, in general, prefetching make sense when the data set is not large enough and the memory access pattern is unpredictable.

Two Types of Prefetching

Essentially, there are two types of prefetching. The simple type, with a constant offset, we already introduced:

```
for (size_t i = 0; i < n; i++) {
    __builtin_prefetch(&a[index[i + 8]]);
    a[index[i]]++;
}
```

In this case, there is a constant offset prefetching of 8. If the loop is processing `a[index[i]]`, it will prefetch `a[index[i+8]]`. In this case, prefetching will go beyond the array boundaries (when `i = n - 1` the loop will prefetch `a[index[n + 7]]`, but this is not typically a problem because prefetching instructions won't cause segmentation faults if the address is not valid.

The second type of prefetching looks like this:

```
for (size_t ii = 0; ii < n; ii +=8) {
    size_t ii_end = std::min(ii + 8, n);
    for (size_t i = ii; i < ii_end; i++) {
        __builtin_prefetch(&a[index[i]]);
    }
    for (size_t i = ii; i < ii_end; i++) {
        a[index[i]]++;
    }
}
```

In this case, there are two loops working on a batch of 8 pieces of data. The first loop does only prefetches, the second loop does the computations.

In our experiments, the second approach was faster. However, we wouldn't generalize it that it is always faster.

Like what you are reading? Follow us on LinkedIn , Twitter or Mastodon and get notified as soon as new content becomes available.
Need help with software performance? Contact us!

Pointer Chasing Codes

Pointer chasing codes (linked lists or binary trees) need to be rewritten in order to benefit from prefetching. Here is the example binary tree lookup:

```
bool lookup(node* n, int val) {
    if (n == nullptr) {
       return false
    } else if (n->val == val) {
        return true;
    } else if (val < n->val) {
        return lookup(n->left, val);
    } else {
        return lookup(n->right, val);
    }
}
```

This code has a instruction dependency on memory loads. In order to prefetch, we need to know a few memory addresses in advance. However, with these type of codes this is not possible: we don’t know the address of the next node until we have loaded the current node and performed the comparison. Issuing a prefetch request immediately before accessing a variable doesn’t work.

Luckily, there is a solution. We already covered the techniques which are aimed at braking dependency chains or interleaving several dependency chains in our post about instruction level parallelism. Techniques presented there combine nicely with explicit software prefetching. Let’s take our binary tree example and perform 16 parallel searches:

```
void lookup(node n[16], int val, int res[16]) {
    size_t total_count = 0;
    for (size_t i = 0; i < 16; i++) {
        if (res[i] == 2) {
            total_count++;
            if (n[i] == nullptr) {
                res[i] = 0;
            } else if (n[i]->val == val) {
                res[i] = 1;
            } else if (val < n[i]->val) {
                n[i] = n->left;
            } else if (val > n[i]->val) {
                n[i] = n->right;
            }
        }
    }
    if (total_count > 0) {
        lookup(n, val, res);
    }
}
```

This code performs 16 parallel searches. We can easily add two prefetches like this:

```
void lookup(node n[16], int val, int res[16]) {
    size_t total_count = 0;
    for (size_t i = 0; i < 16; i++) {
        if (res[i] == 2) {
            total_count++;
            if (n[i] == nullptr) {
                ...
            } else if (val < n[i]->val) {
                __builtin_prefetch(n->left);
                n[i] = n->left;
            } else if (val > n[i]->val) {
                __builtin_prefetch(n->right);
                n[i] = n->right;
            }
        }
    }
    ...
}
```

This will cause the CPU to prefetch the correct child. If we were running one search in parallel, the access to this child would be immediately after the prefetch and prefetching wouldn't make sense. With 16 parallel searches, there is enough time for the hardware to prefetch the child before it is being accessed.

> *Like what you are reading? Follow us on LinkedIn , Twitter or Mastodon and get notified as soon as new content becomes available.*
> *Need help with software performance? Contact us!*

Experiment

In this experiment we want to measure the effect of prefetching to speed up access to a hash set. For this, we implemented a hash map that performs prefetches to speed up access to its data. The source code is available here.

We test on two architectures, Intel(R) Core(TM) i5-10210U and ARM's Cortex-A78AE running on NVIDIA Jetson Orin board. The Intel's platform has 32 kB L1D cache per core, 256 kB L2 cache per core and 6 MB of L3 cache common to all cores. The ARM's has 64 kB L1D cache per core, 256 kB L2 cache per core and 2 MB of L3 cache common to all cores.

Originally we intended to test on Cortex-A72 on Raspberry Pi 4, but prefetching didn't have any effect and only made our test slower. We don't know what is the reason for this, if you do know, feel free to leave a comment.

Our test consists of hash maps of various sizes 16 kB, 64 kB, 256 kB, 1 MB, 4 MB, 16 MB, 64 MB and 256 MB. As far as lookup algorithms, we have the following:

- An `std::unordered_set` implementation and its `find` function we use for reference.
- Our `fast_hash_map` implementation (actually it is a hash set) with `find_multiple_simple` function, which uses simple hash set lookup (calculate hash and perform a lookup immediatellu)
- Our `fast_hash_map` implementation with `find_multiple_nanothreads` function which performs lookups in batches of size X (with X being 1, 8 ,16, 32, …, 256):
  - Calculates X hashes for X input values
  - Optionally: performs X prefetches
  - Accesses X buckets to check if the value is present there

Displaying all the results wouldn't make sense, so let's go through the most important ones.

The Fastest Algorithm

Here are the side-by-side comparison of STL implementation, fast hash map simple implementation, fast map batches implementation without prefetching and fast map batches implementation with prefetching for various hash map sizes for Intel's platform:

Size	STL	Fast Map Find Simple	Fast Map Batches No Prefetching	Fastest Map Batches Prefetching
16 KB	0.781	0.777	0.724 (1 batch size)	0.748 (1 batch size)
64 KB	0.842	0.848	0.842 (1 batch size)	0.859 (1 batch size)
256 KB	0.904	0.925	0.969 (16 batch size)	0.911 (16 batch size)
1 MB	0.866	0.976	1.02 (1 batch size)	0.872 (16 batch size)
4 MB	2.045	2.189	1.864 (64 batch size)	0.928 (128 batch size)
16 MB	2.522	2.587	2.107 (128 batch size)	0.961 (128 batch size)
64 MB	2.864	2.841	2.27 (192 batch size)	1.052 (192 batch size)
256 MB	3.075	3.064	2.447 (128 batch size)	1.183 (256 batch size)

The same measurement, but for our ARM platform:

Size	STL	Fast Map Find Simple	Fast Map Batches No Prefetching	Fastest Map Batches Prefetching
16 KB	0.256	0.255	0.266 (1 batch size)	0.254 (1 batch size)
64 KB	0.32	0.312	0.352 (16 batch size)	0.314 (8 batch size)
256 KB	0.471	0.496	0.5 (1 batch size)	0.465 (16 batch size)

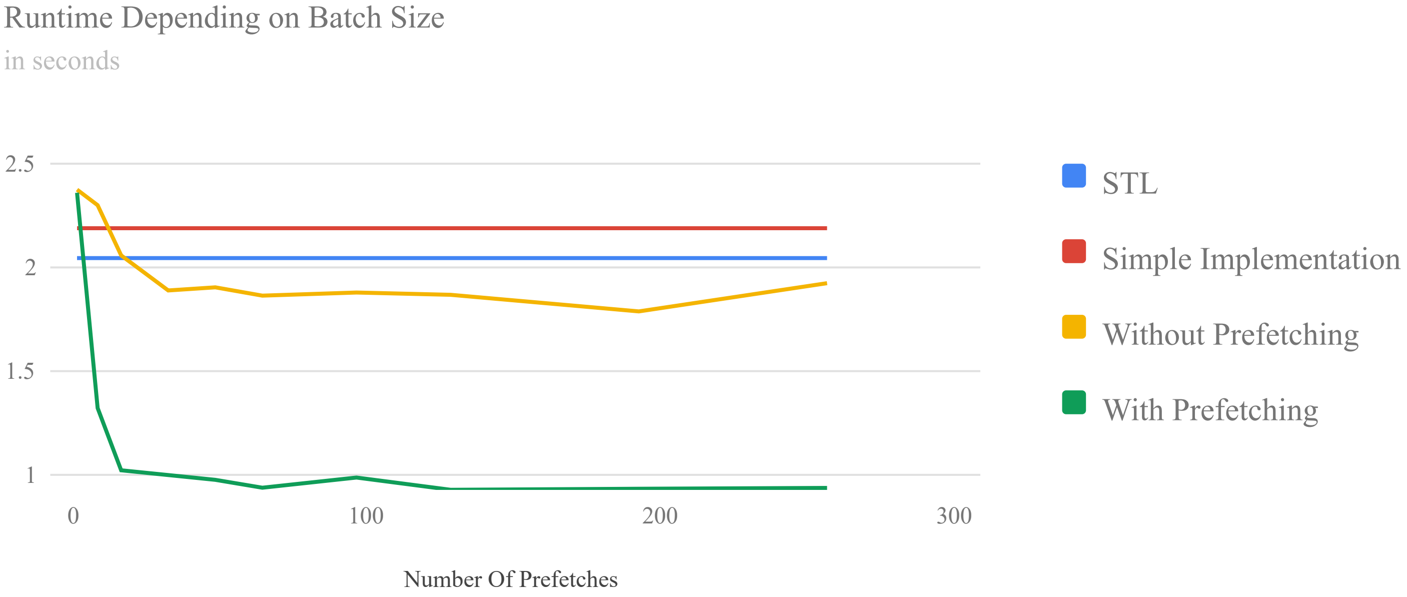
Size	STL	Fast Map Find Simple	Fast Map Batches No Prefetching	Fastest Map Batches Prefetching
1 MB	0.905	1.114	0.91 (256 batch size)	0.745 (48 batch size)
4 MB	1.91	2.417	1.982 (256 batch size)	1.557 (16 batch size)
16 MB	2.38	2.601	2.101 (256 batch size)	1.674 (16 batch size)
64 MB	3.008	2.635	2.131 (256 batch size)	1.691 (16 batch size)
256 MB	5.446	2.633	2.142 (256 batch size)	1.705 (16 batch size)

We see similar results for both architectures. For small hash sets, prefetching doesn't pay off. It only pays off for hash sets of 4 MB and larger. The speedup depends on the hash map size, and generally the bigger the hash set the bigger the speedup. Speedup is largest for hash set of 256 MB in size, 2.6 on Intel's plarform and 1.55 on ARM platform.

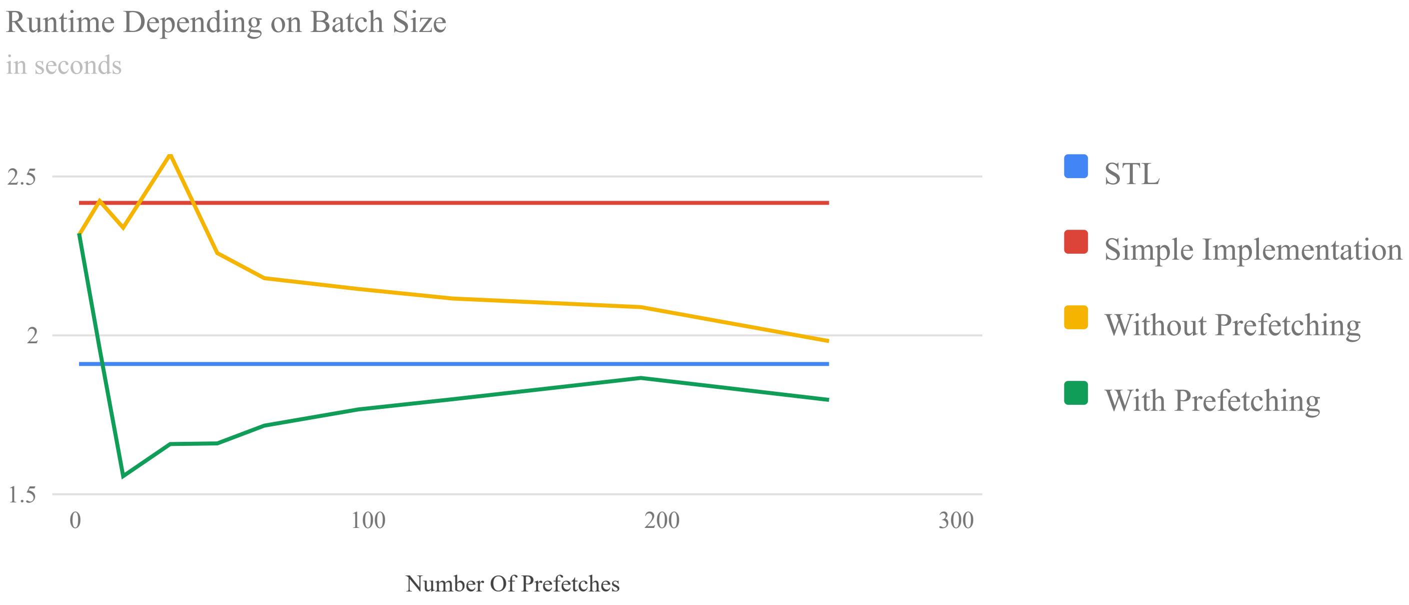
How Much in Advance Should We Prefetch?

In our case, the batch size determines how much in advance we are prefetching. Let's look at the runtime for hash set size 4 MB depending on the number of prefetches for Intel's platform:

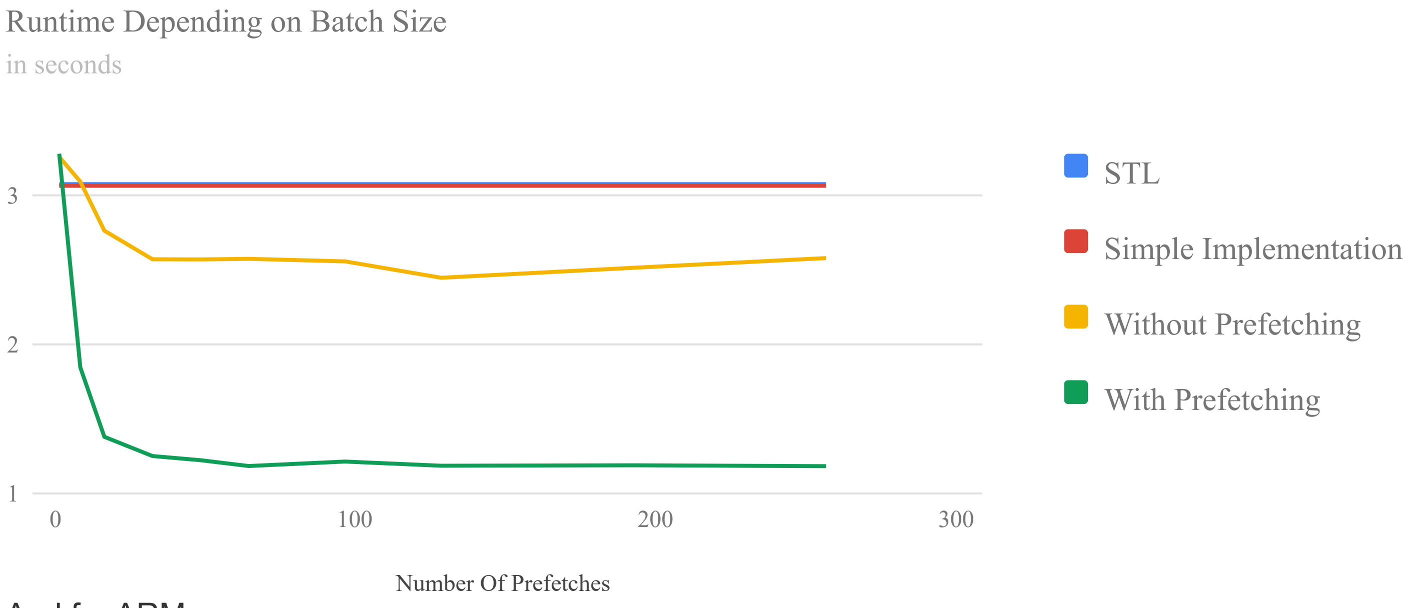
Unknown renderer type ×



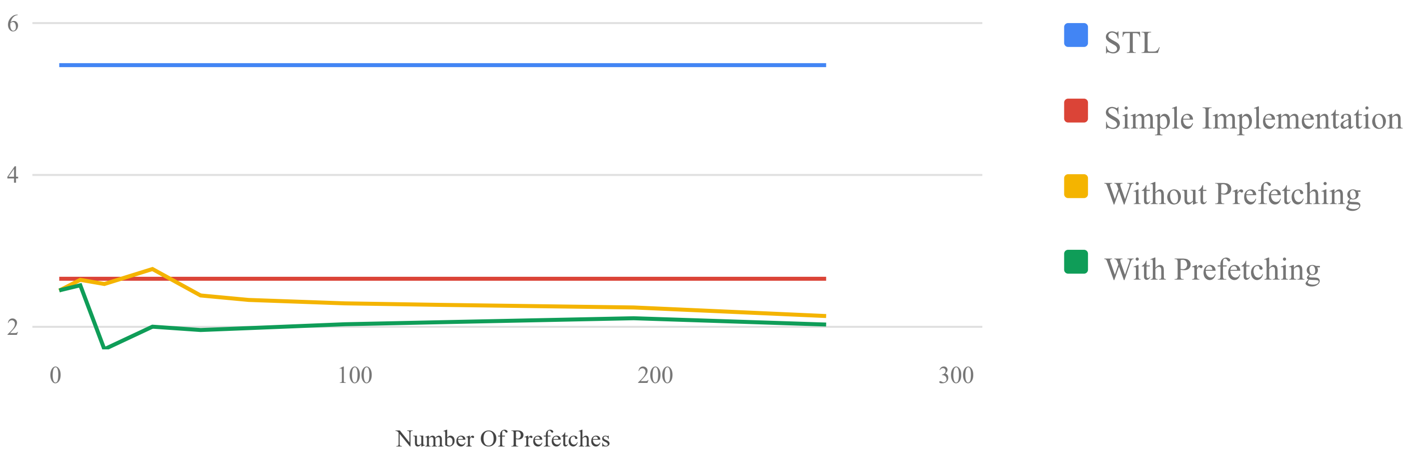
Unknown renderer type ×



Here are the same graphs, but for the largest dataset of 256 MB:



And for ARM:



Runtime Depending on Batch Size

in seconds