CS5234 Mini-Project Final Report

Tran Ba Phong, Apivich Hemachandra

January 8, 2022

Abstract

In our project, we empirically investigate the performance of various matrix multiplication algorithms under a cache-adaptive model. We try to simulate a two-level memory hierarchy and see how the algorithm performs in the external memory model when the cache changes size. We then attempt to test the algorithms on a real machine, by checking the number of page faults the algorithm requires with varying cache availability. Both the simulation and on the real machine demonstrate that cache-adaptive algorithms are more stable as the cache profile varies.

1 Introduction

Cache-adaptive model is an extension of the external memory model which creates a cache whose behaviour is closer to that of a cache in real life. Rather than assuming that a cache will always have a fixed memory size, we now assume that the size of cache available for a process can now change as the program execution progresses [3]. This mimics a behaviour of a cache in real life, where the process will not always see a fixed amount of cache throughout its execution, but instead will have to share resources with another concurrent process.

While there has been a number of papers who study theoretical aspects of cache-adaptivity [3], there has been a more limited number of works who have tried to experiment these algorithms against real program executions [8], or even on simulators. In our project, we would like to investigate an algorithm, particularly the matrix multiplication algorithm, which is claimed to be cache-adaptive. We will be testing different matrix multiplication algorithms on a system whose cache size changes over time. We will attempt to run experiments both on a simple memory hierarchy simulator (where we can directly influence the cache size) and also on real systems (where we can only *in*directly influence cache size).

The report will be organised as follows. In Section 2, we will first provide background information on the cache-adaptive model and on existing matrix multiplication algorithms. Then, in Section 3, we will describe the experimental setup and results from running the algorithms on a simulator. Finally, in Section 4, we attempt to replicate some of the algorithms and the tests on a real machine. All the code for our experiments can be found at `https://github.com/tpvt99/cs5234_cache`.

2 Theory

2.1 Cache-Adaptivity

Formally, the cache-adaptive model can be seen as an extension to the disk-access model (DAM), where the costs are only incurred when the algorithm has to read or write information to the disk [3]. The cache-adaptive model is the case where we allow the size of the cache to change during the execution of the program. The intention of the cache-adaptive model is to mimic situations in real life where the cache available to a program is not stationary but is being divided based on the CPU and on the needs of the other competing processes.

We can specify the memory profile of a cache as a function $m : \mathbb{N} \times \mathbb{N}$ where $m(t)$ is defined the be the number of cache blocks the cache has at the tth cache miss of the algorithm, and let the total cache size $M(t) = B \cdot m(t)$ where B is the cache block size [3].

Since the memory profile can be chosen by an adversarial, it can be difficult to compare different algorithms under this model, since it is possible to make an algorithm run arbitrarily slower than

another by setting the memory profile badly enough as time goes on. As a result, we have to adjust the notion of optimality of an algorithm in the cache-adaptive model.

Definition 1. *Cache-adaptivity [3]*

- *For any memory profile m, the c-speed augmentation of m is defined as the memory profile $m'(t) = c \cdot m(t)$.*
- *For a problem P, an algorithm A is optimal in the cache-adaptive model if there exists some c and a large enough problem input size that, such that for any memory profile m, the worst case running time of A on a c-speed augmentation of m is better than the worst-case running time of any other algorithm solving P on the memory profile m.*

Intuitively, c-speed augmentation of a memory profile is a modification where we make the memory profile change c times slower. For an optimal algorithm, we should only have to slow the memory profile down up to a certain point before it is able to beat any other algorithm solving the same problem. The optimality condition requires that the cache has got optimal eviction policy (i.e. knows which cache line to evict to give the best possible running time), however it can also be shown that a cache using least-recently-used (LRU) eviction policy will only perform at most a constant factor times worse, given the memory profile is augmented by some constant amount [3].

In [3], the authors have shown that divide-and-conquer algorithms with a particular condition is cache-adaptive. We summarise this result below.

Theorem 1. *If a divide-and-conquer algorithm*

1. *Is cache-oblivious, and*
2. *Has the recursive running time in the form of $T(n) = a \cdot T(n/b) + O(1)$ for some constant a and b (i.e. is a constant-overhead recursive algorithm),*

then the algorithm is cache-adaptive [3].

The full proof to the theorem can be found in [3]. From this result, we can automatically conclude that a large number of problems can in fact be solved using a cache-adaptive algorithm. This includes:

- Matrix transposition. Given a matrix A, we would like to find its transposition A^T. This can be done with a divide-and-conquer approach by dividing a matrix into quadrants, transposing each quadrants, then recombining it into a final answer [3, 10].
- Matrix multiplication. We describe this problem in detail in the next subsection.
- All-pairs shortest path problem. Given a weighted graph G, we would like to find the shortest path between all pairs of vertices in the graph. Traditionally, this problem can be solved using Floyd-Warshall algorithm, which can be implemented naively with 3 nested for loops, similar to a naive matrix multiplication algorithm (see Chapter 25 of [5]). When the graph is stored as an adjacency matrix, we can use a divide-and-conquer approach to solve the problem [3, 9].

2.2 Matrix Multiplication

Matrix multiplication is a fundamental algorithm in linear algebra and is an important function that has uses in machine learning, deep learning, operations research, and more. Below, we describe the storage of matrix in memory, before describing matrix multiplication algorithms which requires $O(n^3)$ multiplications, but whose runtime differs when analysed using the external memory model. We note that matrix multiplication that requires $o(n^3)$ scalar multiplications do exist, such as the Strassen algorithm [12], however we will not be considering their variants here.

2.2.1 Storing Matrix In Memory

In the matrix multiplication problem, we are given input matrices A and B, and our goal is to fine the output matrix $C = AB$. We will assume that all the matrices are stored in row-major order, meaning that in the memory, data is laid out by row, as opposed to column-major order where the data is laid out column by column. See Figure 1 for an example.

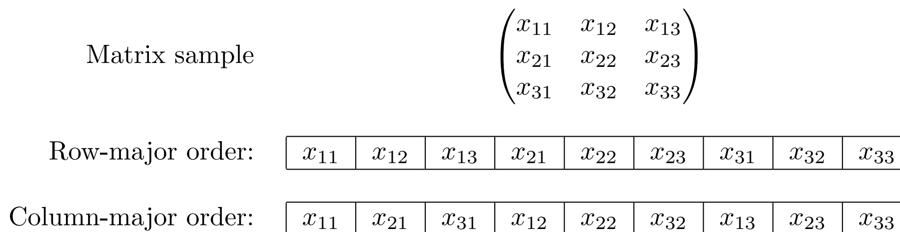

Figure 1: Example of matrix layout in memory.

Since our matrix is stored in row-major order, values which are in the same row of the matrix will have more locality than values in the same matrix. Therefore it is more efficient to scan the matrix in row order than it is in column order. Scanning an $n \times n$ matrix in row order requires $O(n^2/B)$ I/O operations in the external memory model, but scanning the matrix in column order requires $O(n^2)$ I/O operations.

2.2.2 Naive Matrix Multiplication

In the naive matrix multiplication algorithm, we calculate $C[i][j]$ by multiplying row i^{th} of matrix A with column j^{th} of matrix B and sum them up. This equates to $C[i][j] = \sum_k A[i][k] \cdot B[k][j]$. This simply follows the definition of matrix multiplication from linear algebra. Algorithm 1 shows the pseudocode.

Algorithm 1 Naive matrix multiplication algorithm, where $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$.

```
 1: function MatMul-Naive(A, B)
 2:     Initialize all-zeros array C of size m × n
 3:     for i ← 1 : m do
 4:         for j ← 1 : n do
 5:             for k ← 1 : r do
 6:                 C[i][j] ← C[i][j] + A[i][k] · B[k][j]
 7:             end for
 8:         end for
 9:     end for
10:     return C
11: end function
```

For the rest of the report, to easily analyze the computational complexity, we assume that the input matrix A and B are square matrix $\in \mathbb{R}^{n \times n}$. We use **ideal-cache model** that cache blocks can be stored anywhere in the cache. Let M be cache size and B be cache block size, and let $M = \mathcal{O}(B^2)$, which is generally known as *tall cache assumption*

The naive approach takes $\mathcal{O}(n^3)$ works: for each element $C[i][j]$, we need n multiplications and $(n-1)$ additions for each ith row of A and j^{th} column of B. We have total n^2 entries in C, thus we need $\mathcal{O}(n^3)$ works.

To analyze the cache miss, we assume that the memory layout of matrix is row-major order, and $n > B$ so that cache size does not fit all two input matrices. For each calculation of $c[i][j]$, we need to load n/B blocks of ith row of matrix A and n blocks of jth columns of matrix B. Thus for all n^2 entries of C, we incur $\mathcal{O}(n^3/B + n^3) = \mathcal{O}(n^3)$ cache misses.

2.2.3 Transposed Matrix Multiplication

In order to optimise the naive matrix multiplication algorithm in Algorithm 1, we can swap the for loops in Lines 4 and 5. This has the effect of making the algorithm iterate all of the matrices in row-order, hence exploiting locality. A similar effect can be achieved if we arrange matrix B to be in column-major order (through a transposition), then running the unmodified algorithm. Either option is able to reduce the cost down to $\mathcal{O}(n^2/B)$ [1].

2.2.4 Block Matrix Multiplication

The idea behind cache-aware method is to split the each input matrix A and B into smaller matrix that its multiplication fits inside the cache. The pseudocode is as shown in Algorithm 2.

Algorithm 2 Cache-aware matrix multiplication for matrices $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$ with cache size M.

```
1: function MatMul-Cache-Aware(A, B, M)
2:     Allocate all-zeros matrix C of size m × n
3:     sub_size ← √(M/2)
4:     block_i ← m/sub_size
5:     block_j ← n/sub_size
6:     block_k ← r/sub_size
7:     for i ← 1 : block_i do
8:         for j ← 1 : block_j do
9:             for k ← 1 : block_k do
10:                Increment block (i, j) of C with product of block (i, k) of A and block (k, j) of B
11:            end for
12:        end for
13:    end for
14:    return C
15: end function
```

Same as naive method, this method incurs the total works which is $\mathcal{O}(n^3)$. However, the cache miss is different. We analyze by dividing into 2 cases: i) when $n > \mathcal{O}(\sqrt{M/2})$ and ii) $n < \mathcal{O}(\sqrt{M/2})$.

We first begin with case i). Because we divide the matrix whose size is n by n into sub-matrix whose size is $\sqrt{M/2}$ by $\sqrt{M/2}$, there are $2n^2/M$ sub-matrices in both the input and output. To calculate each sub-matrix $C[i][j]$ in output C, we need $n/\sqrt{(M/2)}$ additions of input sub-matrices ($C[i][j] = \sum_k A[i,k] \cdot B[k,j]$, where $k = n/\sqrt{M/2}$). To multiply two input sub-matrices $A[i,k] \cdot B[k,j]$, we cost $2(\sqrt{M/2})^2/M = M/B$ cache misses since the two sub-matrices can fit into cache. Thus the total cache miss is $2n^2/M \cdot n/\sqrt{M/2} \cdot M/B = 2\sqrt{2}n^3/B\sqrt{M}$

For case ii), the input matrix can fit into cache thus it requires n^2/B read operations to load matrix into cache.

Combining two cases, we have the final cache miss is $\mathcal{O}(n^3/B\sqrt{M} + n^2/B)$

2.2.5 Cache-Oblivious Matrix Multiplication

In cache-oblivious model, we do not know the cache size and cache block size in advance. We use divide-and-conquer approach to recursively divide the matrix into smaller sub-matrices that should fit the cache [10]. We try to divide into smallest possible sub-matrices (unlike for the case of cache-aware, we know the cache size so we can divide it to a degree that it fits into cache). See Algorithm 4 for an example of this.

Let $T(n)$ be the cost for running MatMul-Cache-Oblivious on two square matrices of dimension $n \times n$. If $n \leq \mathcal{O}(\sqrt{M})$, then all matrices can fit into cache and the cost required is only $\mathcal{O}(M/B)$. If $n > \mathcal{O}(\sqrt{M})$, the cost comes from 8 recursive calls and allocating array C, which results in $T(n) = 8 \cdot T(n/2) + \mathcal{O}(n^2)$, which solves to $T(n) = \mathcal{O}(n^3/B\sqrt{M} + n^2/B)$ [7]. Even though the bound is same as Block Matrix Multiplication, we obtain this bound without knowing the size of cache.

2.2.6 Cache-Adaptive Matrix Multiplication

MatMul-Cache-Oblivious currently utilises a divide-and-conquer approach, which can be easily made into a cache-adaptive algorithm if we are able to ensure that the additional overhead per recursive call is a constant factor. To do so, we make sure that the algorithm does not allocate extra memory per each recursive call (which otherwise would have been an $\mathcal{O}(n^2)$ factor) and instead write values directly to a pre-allocated array. This modification is reflected in Algorithm 4.

Theorem 2. MatMul-Cache-Adaptive *is a cache-adaptive matrix multiplication algorithm which requires* $\mathcal{O}(n^3/B\sqrt{M})$ *cost [3].*

Algorithm 3 Cache-oblivious matrix multiplication. We assume that A and B are square matrices whose dimensions are a power of 2.

```
1: function MatMul-Cache-Oblivious(A, B)
2:     n ← dimension of matrix
3:     if n < 2 then
4:         return MatMul-Naive(A, B)
5:     end if
6:
7:     A_11 ← A[0 : n/2, 0 : n/2]
8:     A_12 ← A[0 : n/2, n/2 : n]
9:     A_21 ← A[n/2 : n, 0 : n/2]
10:    A_22 ← A[n/2 : n, n/2 : n]
11:
12:    B_11 ← B[0 : n/2, 0 : n/2]
13:    B_12 ← B[0 : n/2, n/2 : n]
14:    B_21 ← B[n/2 : n, 0 : n/2]
15:    B_22 ← B[n/2 : n, n/2 : n]
16:
17:    Allocate matrix C with all zeros and dimension n × n
18:    C_11 ← MatMul-Cache-Oblivious(A_11, B_11) + MatMul-Cache-Oblivious(A_12, B_21)
19:    C_12 ← MatMul-Cache-Oblivious(A_11, B_12) + MatMul-Cache-Oblivious(A_12, B_22)
20:    C_21 ← MatMul-Cache-Oblivious(A_21, B_11) + MatMul-Cache-Oblivious(A_22, B_21)
21:    C_22 ← MatMul-Cache-Oblivious(A_21, B_12) + MatMul-Cache-Oblivious(A_22, B_22)
22:    return C
23: end function
```

To prove the theorem, we can show that MatMul-Cache-Adaptive is recursive with constant overhead per recursive round, and that it is also a cache-oblivious algorithm. Then, its cache-adaptiveness can be shown as a direct consequence of Theorem 1 [3].

3 Experiments On A Simulator

In this section, we describe the experiments done based on a cache simulator. We first describe the structure of the cache simulator, then we describe the experiments performed and the results obtained from them.

3.1 Cache Simulator

We have created a simple cache simulator in Python. Parts of the simulator is based on the `pycachesim` repository [2]. The cache simulator allows loading and storing different values as simple variables and also in an array format. The example of the interface is as shown below.

We have 6 main components in our simulator: CPU, Memory, Cache, Block, EvictionPolicy, and MemoryProfile.

3.1.1 Block

The class represents the block data in both memory and cache. It is a list of integers and each element has 4-byte size. We utilize the object reference trick in which when we write to **Block** in **Cache**, the contents are also updated in **Memory**. In this way, we can remove the write function in **Memory** and speedup the experiments.

3.1.2 Memory

The class plays as a memory to store data. It has only 2 main functions. The first function is named *read_block_from_memory*(), which is used to return a **Block** from a requested address. The second

Algorithm 4 Cache-adaptive matrix multiplication. We assume that A and B are square matrices whose dimensions are a power of 2.

1: **function** MATMUL-HELPER($A, B, C, n, rowA, colA, rowB, colB, rowC, colC$)
2: **if** $n < 2$ **then** multiply naively and write answers to C
3: **end if**
4: $n' \leftarrow n/2$ ▷ to perform on subproblem half the size
5: MATMUL-HELPER($A, B, C, n', rowA, colA, rowB, colB, rowC, colC$)
6: MATMUL-HELPER($A, B, C, n', rowA, colA + n', rowB + n', colB, rowC, colC$)
7: MATMUL-HELPER($A, B, C, n', rowA + n', colA, rowB, colB, rowC + n', colC$)
8: MATMUL-HELPER($A, B, C, n', rowA + n', colA + n', rowB + n', colB, rowC + n', colC$)
9: MATMUL-HELPER($A, B, C, n', rowA, colA, rowB, colB + n', rowC, colC + n'$)
10: MATMUL-HELPER($A, B, C, n', rowA, colA + n', rowB + n', colB + n', rowC, colC + n'$)
11: MATMUL-HELPER($A, B, C, n', rowA + n', colA, rowB, colB + n', rowC + n', colC + n'$)
12: MATMUL-HELPER($A, B, C, n', rowA + n', colA + n', rowB + n', colB + n', rowC + n', colC + n'$)
13: **end function**
14:
15: **function** MATMUL-CACHE-ADAPTIVE(A, B)
16: Allocate matrix C with all zeros and dimension $N \times N$
17: **return** MATMUL-HELPER($A, B, C, 0, 0, 0, 0$)
18: **end function**

function is *allocate()* which is used to initialize the memory according to the matrix data we used for multiplication. We store matrix in row-major order.

3.1.3 Cache

Our **Cache** class is the linchpin of the simulator. It communicates with **Memory** through **CPU** and transferring data using **Block**. The three functions: *_get_set()*, *_get_tag()* and *_get_offset()* are used to decode the memory's address to read/write at the correct blocks in cache. The function *read_from_cache()* returns a **Block** (of memory) at a requested address if that **Block** is in cache, otherwise null is returned. The complement function is *load_from_memory()* loads a **Block** from memory's address to cache. In this scenario, if there is an existing **Block** at this cache's location, the old cache will be evicted based on one of **EvictionPolicy**. The function *overwrite_cache()* overwrites the content in bytes of **Block** at an address. This won't evict any **Block**, instead its contents are updated. Because Python use object by reference, this **Block** in **Memory** is also updated; hence, we do not need any write function in **Memory**.

3.1.4 Eviction Policy

There are 3 main eviction policies we implemented: **LRUPolicy** (Least Recently Used), **LFUPolicy** (Least Frequently Used), and **FIFOPolicy** (First-In First-Out). **LRUPolicy** evicts least recently used **Block**. To do this, the algorithm maintains an *OrderedDict* which has Doubly Linked List as underlying data structure. Everytime a block is accessed, it moves the index of block to the end. When cache is full, it would remove the index of blocks at the beginning. In **LFUPolicy**, the policy evicts the one with least frequently used. We use a hashmap to maintain the frequencies of **Block** index. In removing phase, we need to loop through hashmap to find the index with least frequencies and remove it. For **FIFOPolicy**, cache is evicted in order as they added. We maintain an OrderDicted to serve this purpose.

3.1.5 CPU

CPU contains three important functions. The first one is *read* which is used to read a **Block** from **Cache**. If this **Block** is not inside the **Cache**, it would be fetched from **Memory** and store inside **Cache**. The second function is *write* which writes a byte to a **Block** in cache. If no existing **Block**, then the function requests **Block** from **Memory**, update this **Block**, and write the **Block** to **Cache**.

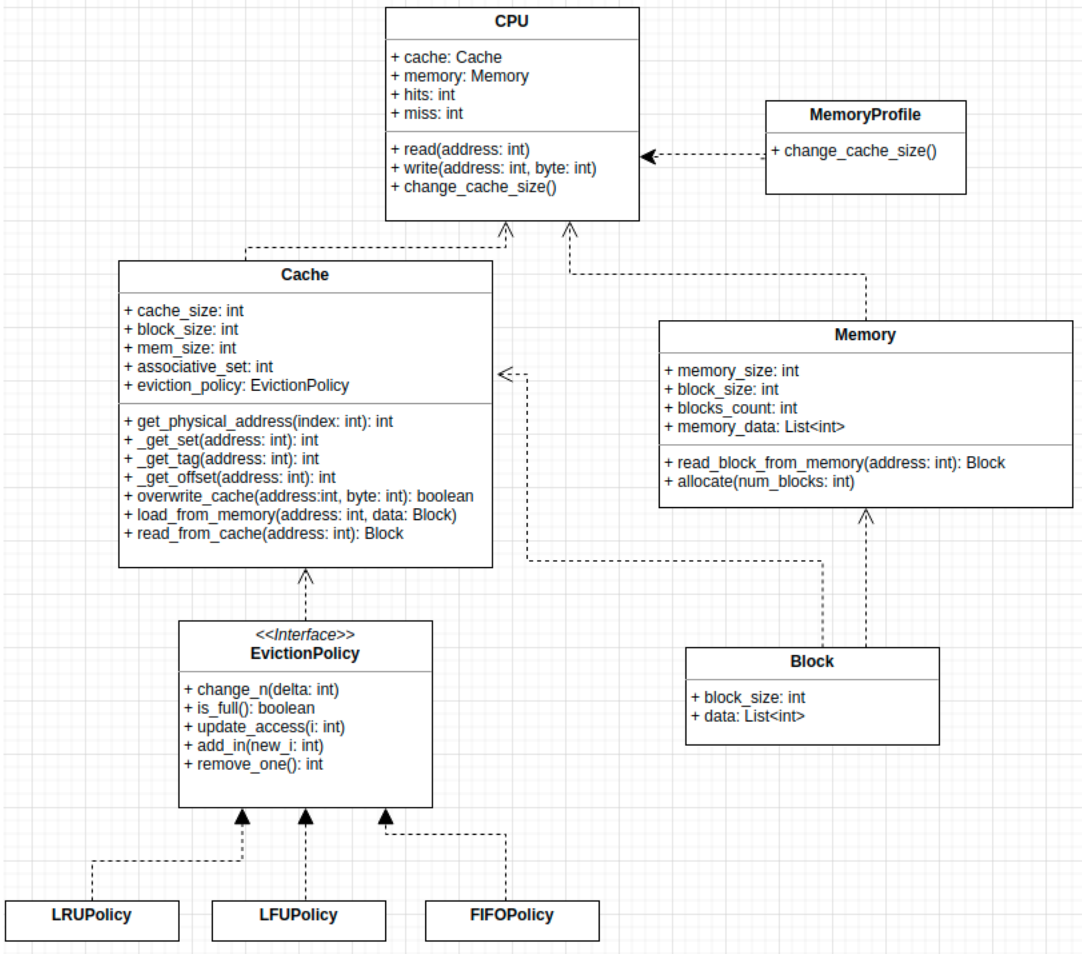

Figure 2: Cache Structure

The final function is for the purpose of cache-adaptive: *change_cache_size*. It updates the size of the cache every time a cache miss occurs according to a specific memory profile.

3.1.6 MemoryProfile

This class is responsible for changing cache's size every time a cache miss occurs as in [3]. The memory profile is as followed:

$$m(t) = \begin{cases} m & \text{if } t \leq m^{5/2} \\ 2m^{5/2} - t & \text{if } m^{5/2} < t < 2m^{5/2} - B \\ B & \text{if } t \geq 2m^{5/2} - B \end{cases}$$

where t is the tth cache misses, $m = c_1 B$, where c_1 is any constant speed augmentation. Here we choose $c_1 = 1$.

3.2 Experiment Results

3.3 Simulator With Fixed-Size Cache

In this section, we use our simulator written in Python to report the cache miss of all four presented algorithms for matrix multiplication. We use **ideal-cache model** and enforce **tall cache assumption**. In each of the figure, we report the cache miss, cache access and the ratio of cache miss to cache access (the ratio is written on the top of each bar)

3.3.1 Naive Matrix Multiplication

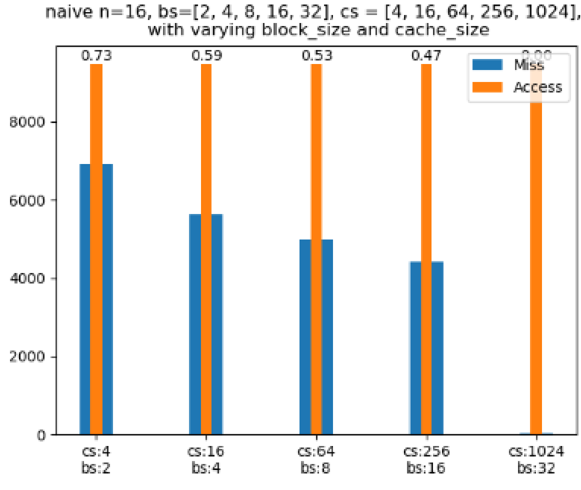

Figure 3: Cache miss and access of Naive Matmul with matrix size 16x16

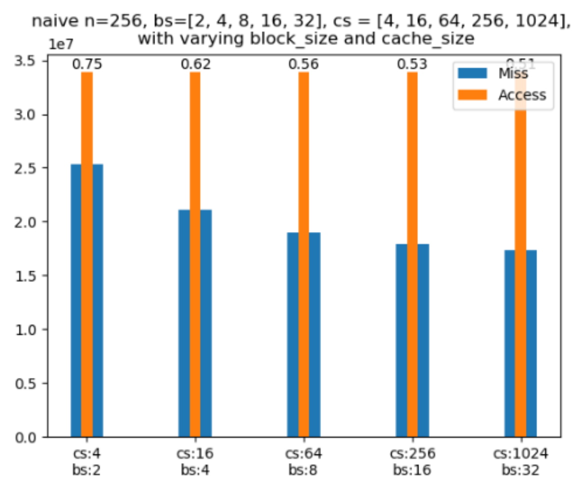

Figure 4: Cache miss and access of Naive Matmul with matrix size 256x256

Figure (3) and (4) show the cache miss of naive matrix multiplication. We already proved that the cache miss for this method is $\mathcal{O}(n^3/B + n^3)$. Let we verify this result with the figures above. In figure (3), when matrix size 16 by 16, block size is 2, the experimental cache miss is 6456, while the theoretical one is $16^3/2 + 16^3 = 6144$ which is nearly equal. The closed equality hold for cache size up to 256. When cache size is 1024, it can fit both three matrices (two inputs and one output), thus there is no cache miss in this case. The theoretical proof holds only when the cache cannot fit all matrices. For bigger matrix in figure (4), experimental cache miss is also same as theoretical cache miss.

3.3.2 Cache-Aware Matrix Multiplication

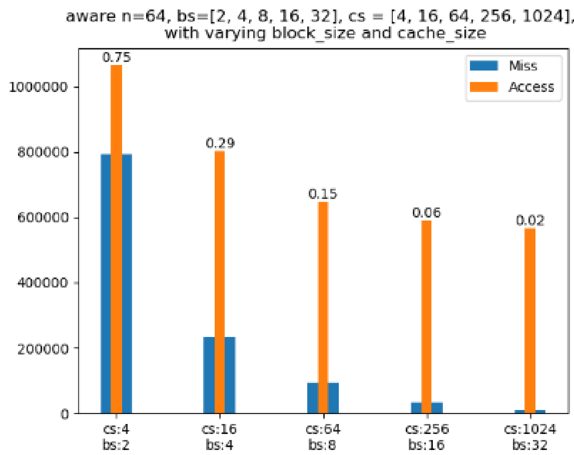

Figure 5: Cache miss and access of Cache-Aware Matmul with matrix size 64x64

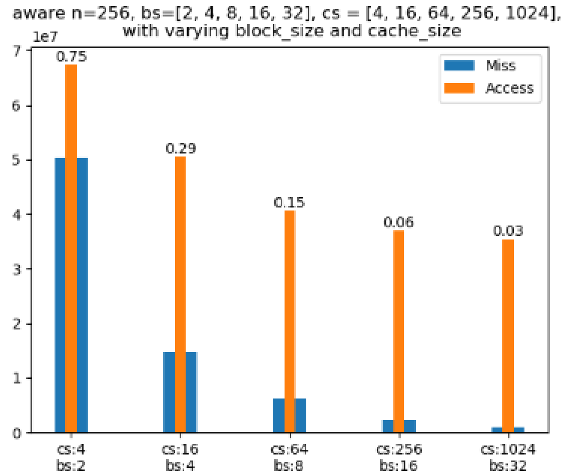

Figure 6: Cache miss and access of Cache-Aware Matmul with matrix size 256x256

The experimental cache miss figures confirm the theoretical cache miss in section 2.2.4. For example, at $n = 64, B = 2, M = 4$, theoretical cache miss is $n^3/B\sqrt{M} + n^2/B = 64^3/2\sqrt{4} + 64^2/2 = 67584$, while the experimental one is around 80000.

3.3.3 Cache-Oblivious Matrix Multiplication

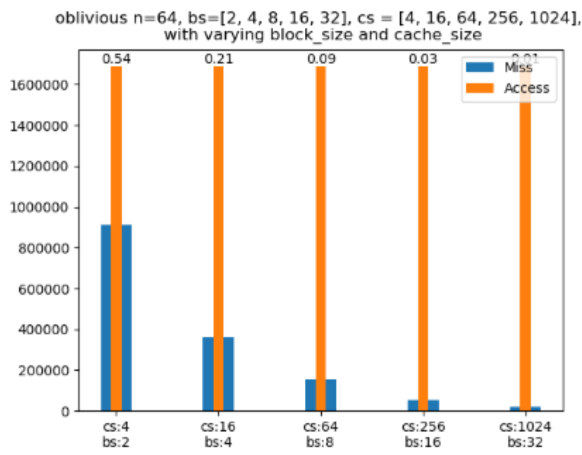

Figure 7: Cache miss and access of
Cache-Oblivious Matmul with matrix size 64x64

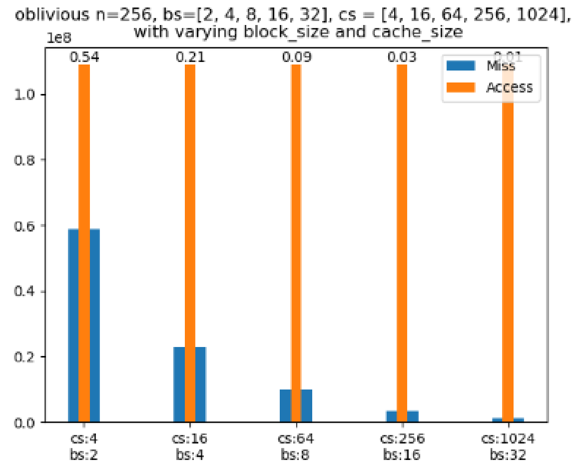

Figure 8: Cache miss and access of
Cache-Oblivious Matmul with matrix size 256x256

3.3.4 Cache-Adaptive Matrix Multiplication

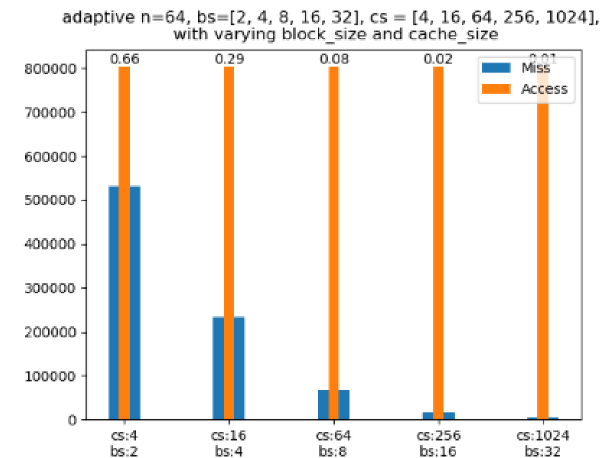

Figure 9: Cache miss and access of
Cache-Adaptive Matmul with matrix size 64x64

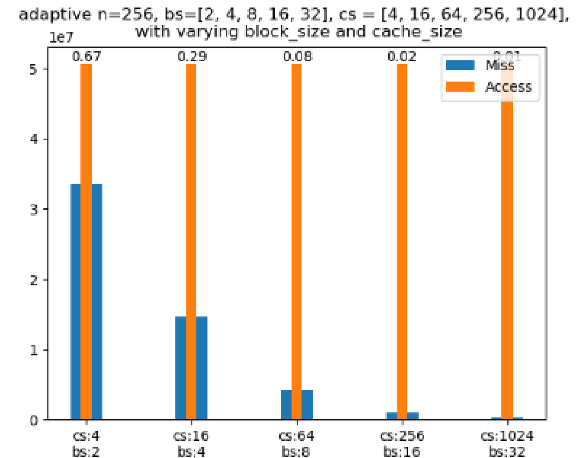

Figure 10: Cache miss and access of
Cache-Adaptive Matmul with matrix size 256x256

In Figures 11 and 12, we show the amount of cache misses each algorithms incur with changing cache block sizes, for different matrices sizes. As we can see, in both examples, the naive algorithm has the highest number of cache misses, followed by the cache-oblivious implementation. This is because of the fact that the cache-oblivious implementation requires allocation of extra memory in each recursive call, which increases the number of data accesses required in allocation and also in copying answers between arrays. We can see that the cache-adaptive implementation requires the less cache misses out of the four algorithms tested.

We also note the effect of block sizes with the cache misses incurred. A higher block size tends to decrease the number of cache misses of an algorithm, assuming the cache size is kept the same.

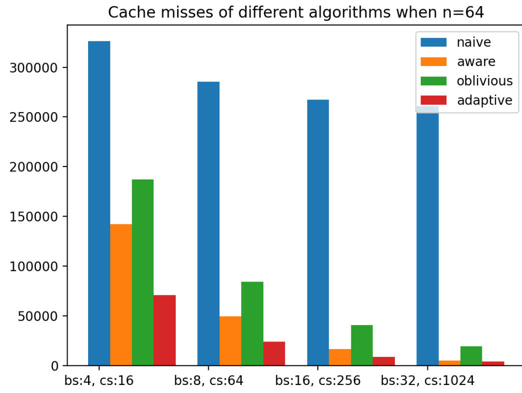

Figure 11: Cache misses of 4 different algorithms with matrix size 64×64

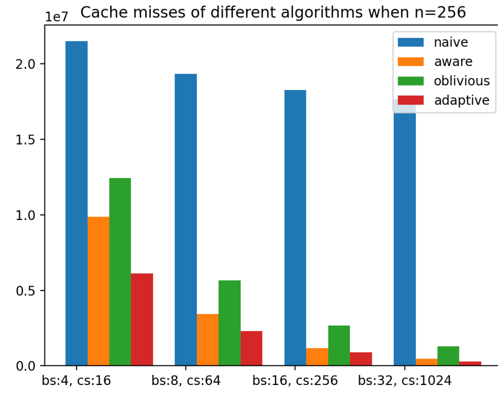

Figure 12: Cache misses of 4 different algorithms with matrix size 256×256

3.4 Simulator with changed-size cache

3.4.1 Naive Matrix Multiplication with changed-size cache

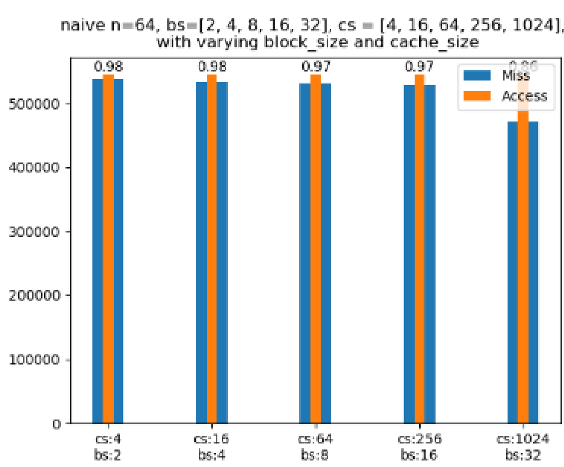

Figure 13: Cache miss and access of Naive matrix multiplication algorithm with matrix size 64x64

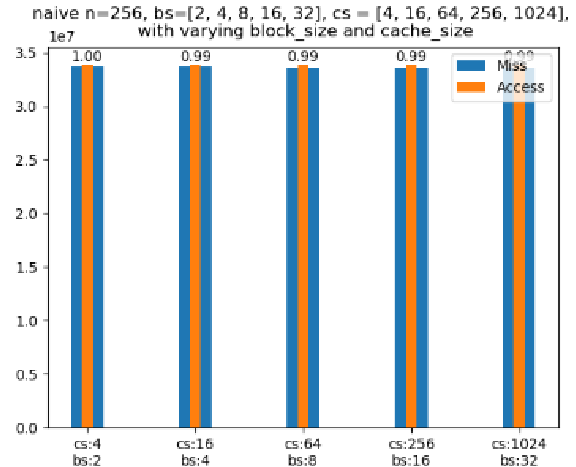

Figure 14: Cache miss and access of Naive matrix multiplication algorithm with matrix size 256x256

As we see from Figure 13 and 14, the miss rate of naive matrix multiplication in the case of changed-size cache is extremely high, suggesting that naive method is not effective. In the case of cache-aware, cache-oblivious and cache-adaptive (Figure 15 to Figure 20), the cache miss is significantly reduced.

3.4.2 Cache-aware Matrix Multiplication with Variable Cache Size

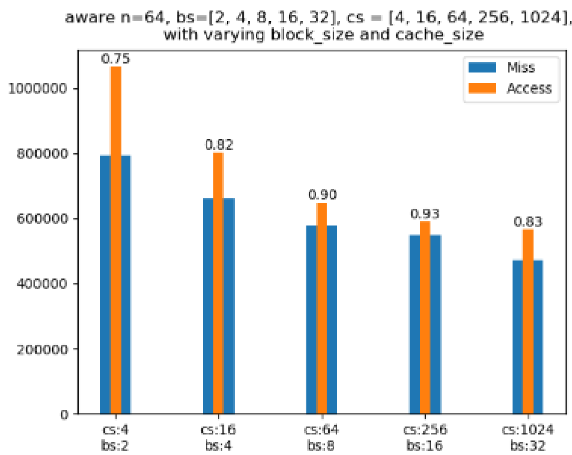

Figure 15: Cache miss and access of Cache-Aware matrix multiplication algorithm with matrix size 64x64

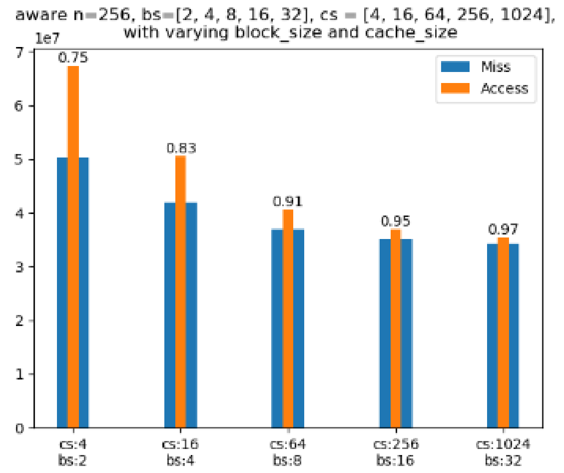

Figure 16: Cache miss and access of Cache-Aware matrix multiplication algorithm with matrix size 256x256

3.4.3 Cache-Oblivious Matrix Multiplication with Variable Cache Size

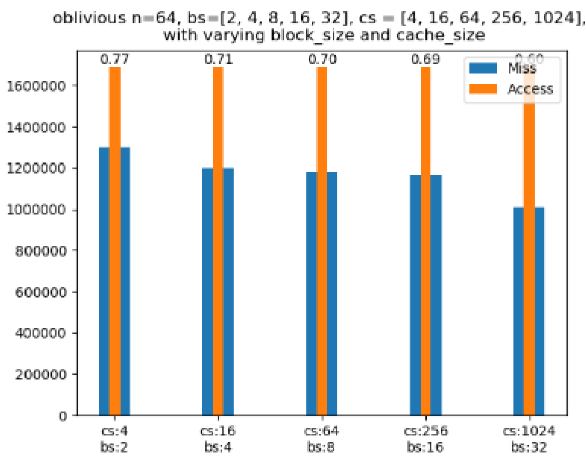

Figure 17: Cache miss and access of Cache-oblivious matrix multiplication algorithm with matrix size 64x64

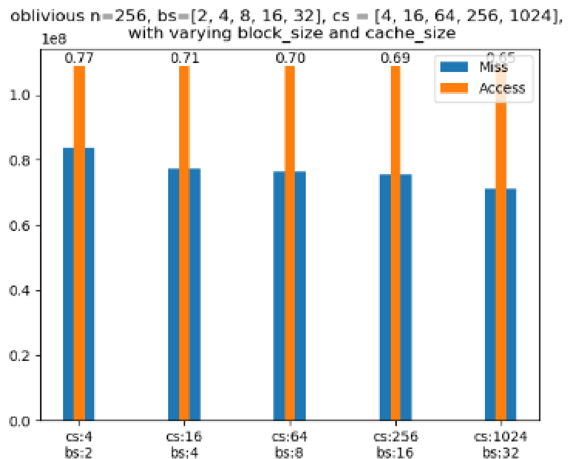

Figure 18: Cache miss and access of Cache-oblivious matrix multiplication algorithm with matrix size 256x256

3.4.4 Cache-adaptive Matrix Multiplication with Variable Cache Size

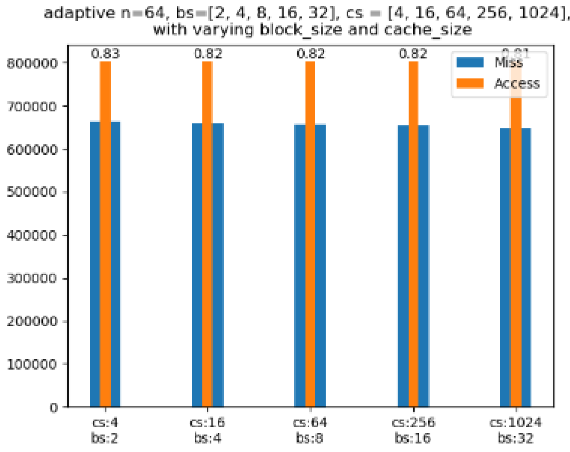

Figure 19: Cache miss and access of Cache-adaptive matrix multiplication algorithm with matrix size 64x64

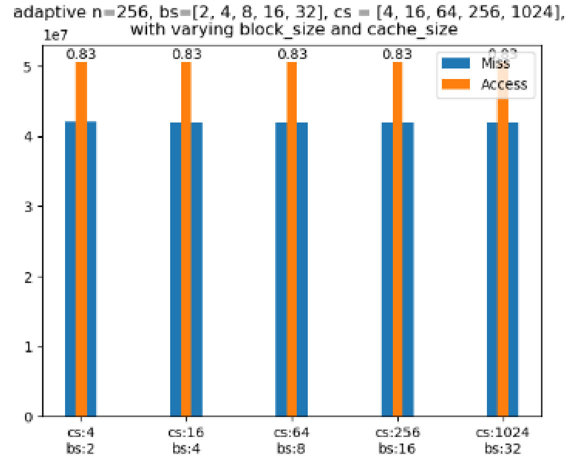

Figure 20: Cache miss and access of Cache-adaptive matrix multiplication algorithm with matrix size 256x256

In Figures 21 and 22, we plot the cache miss rates (i.e. cache misses divided by total cache accesses) between the different algorithms. From the graphs, we see that the cache-oblivious algorithm has the lowest cache miss rate, however with a caveat that it requires a higher number of total cache accesses. Out of the remaining algorithms, the cache-adaptive algorithm is seen to have the lowest cache miss rate, and hence will result in a better performance than the other algorithms, regardless of how the cache size changes.

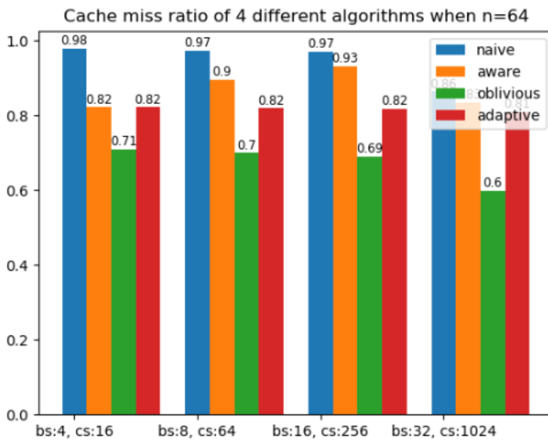

Figure 21: Cache miss ratio of 4 different algorithms with matrix size 64×64

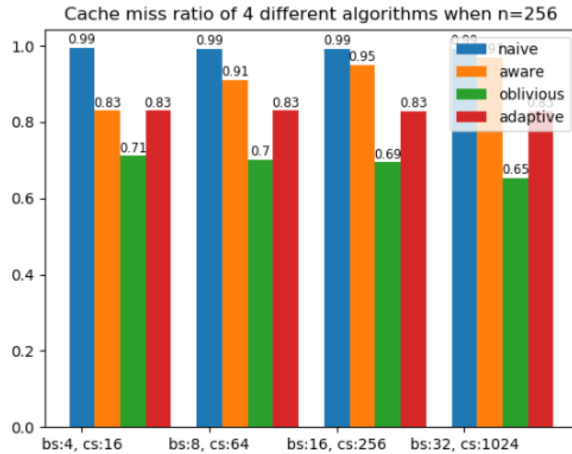

Figure 22: Cache miss ratio of 4 different algorithms with matrix size 256×256

4 Cache-Adaptive Algorithms On A Real Machine

4.1 Difficulties Of A Real Machine

Another test we wanted to perform is to see how the matrix multiplication algorithms were able to perform on real machines, rather than on a simulation with simplified two-level memory hierarchy.

Whilst coming up with an appropriate setup, we found a couple of challenges in running the code and measuring cache misses. We list some considerations and challenges below.

- Running experiments on a real machine may be challenging due to the different processes that goes on in a machine, and therefore is more difficult to control factors such as processes running and memory being allocated to each processes (including for our algorithm). This is another consideration since for cache-adaptive algorithms, we will need the ability to adjust our cache size throughout the execution. It therefore might be better to run on a machine that has a barebones Linux distribution, or running on a virtual machine or on Docker instead, since there are fewer processes that are being run concurrently in the background.

- One method of counting the number of cache misses is to use the tool called `perf`. Calling `perf stat` allows us to count numerous stats about a program's execution, including the number of cache accesses and cache misses [6]. However, the program is only available on Linux machines, and cannot be ran on virtual machines due to a lack of particular program counters[1].

- Another tool that can be used for counting cache misses is `cachegrind`, which is part of `valgrind`, a popular tool for debugging binary files [11]. However, `cachegrind` ultimately is another simulator (albeit a more sophisticated one than we have described in the previous section), and cannot properly simulate cases of competing cache.

In the end, we find that in the external memory model, we are assuming a two-level memory hierarchy where accessing one hierarchy is much more costly than another. Therefore, rather than considering the L1-cache and the L2-cache as the two hierarchies, we can consider the computer memory and the disk as the two hierarchies. In this case, the cost incurred would be the number of page faults that occurs, since this is the case where the system will need to read data from the disk.

This has two benefits for our setup.

- Firstly, on Docker, one can control the size of the memory easily by setting the `--memory` or `-m` flag in the command line. This means we can squeeze the memory size down low enough to force more page faults. Caches, on the other hand, is more difficult to control.

  Being able to run on Docker also means the experiment can be ran on our personal machines or on cloud servers (which will usually also be a virtual machine), rather than having to find a physical machine whose host system is Linux-based.

- Secondly, page faults can actually be measured on virtual machines using the command `time -v`[2], which can measure the number of major and minor page faults of our algorithm [4]. `time` is also available on other OSes such as MacOS (which does not support `perf`), making this setup easier to be replicated or tested on a wider varieties of machines.

4.2 Setup For Real Machine Tests

For our experiment, we have implemented the naive matrix multiplication, the transposed matrix multiplication and the cache-adaptive (recursive) matrix multiplication in C++. We omitted the cache-aware (block) matrix multiplication since it relies on knowing the cache size, which is not possible on real machines. We also do not try the cache-oblivious (basic recursive) algorithm since from tests, it is inefficient due to the extra memory allocations required in each recursive call.

We test our algorithm on matrices of dimensions 1024×1024. The code is compiled and ran on a Debian Docker image, where we have made sure there is minimal background processes. We limit the memory size of the image to 8 megabytes and allow for as much memory swap as the host machine allows.

In addition to the matrix multiplication algorithm, we also create a disruptor program, whose goal is to take up a changing amount of memory. This is to simulate a situation where multiple programs are competing for the same resource, and the amount of cache available to our program is changing. Our disruptor allocates an array of size 2^k for some integer k, and then proceeds to access the array in random order. Then after T seconds, the disruptor deallocates the array and relocates another array

[1] See discussion at `https://www.researchgate.net/post/Why_doesnt_perf_report_cache-refernces_cache-misses`.

[2] Depending on how the libraries are installed, some machines will require the user to call `/usr/bin/time` rather than just `time` due to multiple binaries being called `time`.

Algorithm 5 Disruptor function which is to be ran simultaneously with the algorithms in order to compete for memory resources of a machine. This program is to continue forever and will only terminate when disrupted. Parameter k controls the size of the memory required by the disruptor, while T controls how quickly the memory profile of the disruptor changes (higher T means the memory profile changes slower).

```
 1: function DISRUPTOR(k, T)
 2:     for 5 iterations do
 3:         Allocate array A of size 2^k            ▷ or deallocate if has been allocated previously
 4:         while loop has repeated for less than T seconds do
 5:             i, j, k ← random elements from [1, 2^k]
 6:             A[i] ← A[j] + A[k]                  ▷ a random procedure that accesses the array
 7:         end while
 8:         k ← k + 1
 9:     end for
10:     while true do                               ▷ loop forever
11:         i, j, k ← random elements from [1, 2^k]
12:         A[i] ← A[j] + A[k]                      ▷ a random procedure that accesses the array
13:     end while
14: end function
```

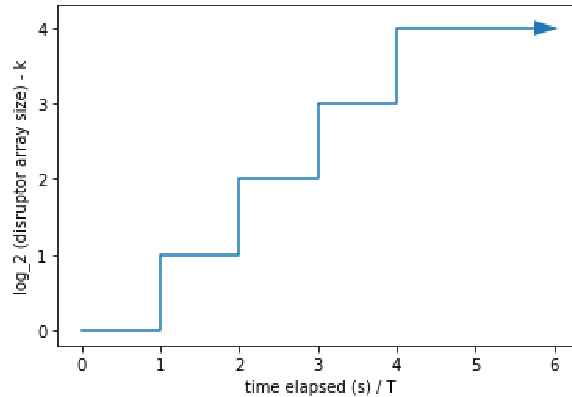

Figure 23: The memory profile required for the disruptor for a given k and T. The memory required for the disruptor starts at 2^k, then doubles every T seconds, until it eventually stops increasing and uses up a constant memory amount indefinitely.

double the size, and again randomly access the array. We repeat this until a certain array size or if the matrix multiplication program terminates. The action of the disruptor is shown in Algorithm 5, and the intended memory usage profile for our disruptor is shown in Figure 23.

4.3 Results Of Matrix Multiplication

In Figure 24, we present the number of page faults and total running time for different matrix multiplication algorithm requires as we change the behaviour of our disruptor by adjusting T, and fixing $k = 20$. As we can see, the cache-adaptive algorithm is more stable and will incur less page faults as T increases. The stability of the cache-adaptive algorithm can also be seen in the total running time, which does not change with the disruptor memory usage.

We do note that despite the results, there are still limitations in our experiment setup. Firstly, the cache profile that we use increases with time, and so the cache profile for the multiplication algorithm gets smaller over time (due to the larger memory required by our disruptor). This means that algorithms that takes a longer time may be penalised more. However, we find that this is necessary since on real machines, the memory profile can only realistically depend on the time, rather than with the number of cache misses so far like in the cache-adaptive model.

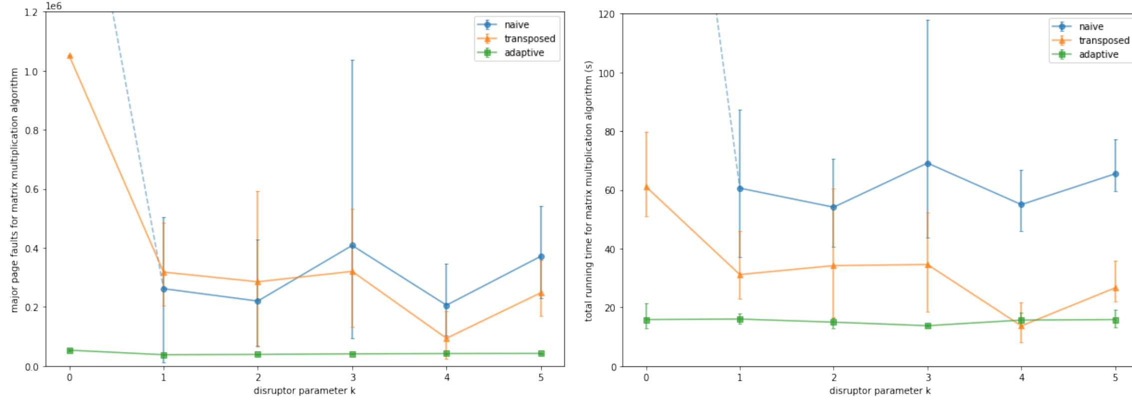

Figure 24: The number of major page faults (left) and total running time (right) required by each matrix multiplication algorithm as the disruptor parameter $k = 20$ and T changes. In both graphs, for each value of k and for each matrix multiplication algorithm, we repeat the experiment 3 times and report the mean (the point on the line), the minimum and maximum found (the error bars). The naive algorithm's attempt when $k = 0$ took too long that the process was terminated.

4.4 Tests of the Floyd-Warshall Algorithm

Based on the tests for matrix multiplication, we also try to extend the same tests on the Floyd-Warshall algorithm. We implement a transposed version of the algorithm, where we use the naive implementation [5] but reshuffle the for loops (similar to the transposed variant of the matrix multiplication algorithm) in order to be more cache efficient. We also implement a cache-adaptive version which is based on the recursive block algorithm from [9]. We ignore results from the naive algorithm since it takes significantly longer time and requires much more page faults than the other two implementations.

The Floyd-Warshall algorithm bears very close resemblance to the matrix multiplication algorithm, and so the structure of code implementation from the matrix multiplication can be reused. One difference is that our Floyd-Warshall algorithm is done in place (i.e. we write the shortest path directly to our adjacency matrix), so we only require to allocate one $N \times N$ matrix, rather than three of them as in matrix multiplication.

We run the Floyd-Warshall algorithm variations on a graph of 1024 vertices (meaning the input is a 1024×1024 adjacency matrix), along with a disruptor with $k = 23$ and varying T (we increase k since the Floyd-Warshall algorithm itself requires less memory, so the memory usage of the disruptor goes up to be more competitive). In Figure 25, we show the number of page faults and the time required by the algorithms.

Similar to the matrix multiplication tests, we see that again the cache-adaptive version is able to withstand the changing cache availability better than the other algorithms. Even though the allocated memory amount for the algorithms are the same for all algorithms, as the disruptor takes up larger memory, the transposed variation was not able to cope as well as the cache-adaptive algorithm, as seen in the more fluctuating amounts of page faults required. We do note that the amount of time between the two algorithms are about the same, but again more stable for the cache-adaptive algorithm.

5 Conclusion

We have empirically demonstrated the ability of different matrix multiplication algorithms in performing under a changing cache profile. We have also ran experiments of both matrix multiplication algorithms and variations of the Floyd-Warshall algorithm on a real machine, and measuring how the number of page faults differ as the cache profile changes. In both cases, we have demonstrated that the cache-adaptive algorithms works well on a variable cache profile.

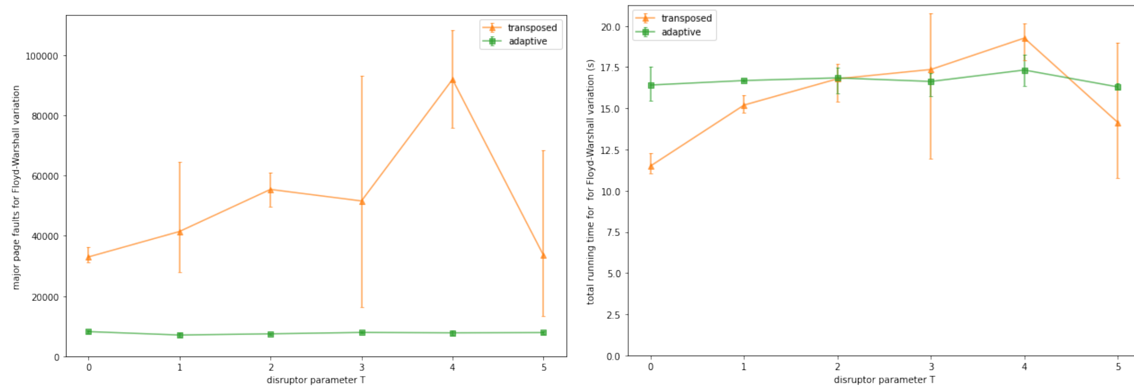

Figure 25: The number of major page faults (left) and total running time (right) required by each variation of the Floyd-Warshall algorithm with the disruptor parameter $k = 23$ and T changes. In both graphs, for each value of T and for each variation of the Floyd-Warshall algorithm, we repeat the experiment 3 times and report the mean (the point on the line), the minimum and maximum found (error bars).

5.1 Future Extensions

The experiments we were able to conduct were limited by the hardware we were able to acquire. The experiment design therefore took into account the fact that some tools for measurement is not available on all machines, and we had to use `time` which can be ran on virtual machines and also on more OSes. To measure further affects such as the number of cache misses on the actual L1 and L2 cache, we would need a setup where Linux is ran as a host OS. We then should be able to repeat the experiments and see the performance of the algorithm in this condition.

References

[1] Cache-Oblivious Algorithms. `https://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/slidesS18/cache-oblivious.pdf`.

[2] RRZE-HPC/pycachesim. `https://github.com/RRZE-HPC/pycachesim`.

[3] Bender, M. A., Ebrahimi, R., Fineman, J. T., Ghasemiesfeh, G., Johnson, R., and McCauley, S. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 2014), SODA '14, Society for Industrial and Applied Mathematics, p. 958–971.

[4] Brouwer, A. time(1) — Linux manual page. `https://man7.org/linux/man-pages/man1/time.1.html`, 2019.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[6] Eranian, S. Tutorial - Perf Wiki. `https://perf.wiki.kernel.org/index.php/Tutorial`, 2015.

[7] Lincoln, A. Analysis of recursive cache-adaptive algorithms.

[8] Lincoln, A., Liu, Q. C., Lynch, J., and Xu, H. Cache-Adaptive Exploration: Experimental Results and Scan-Hiding for Adaptivity. SPAA '18, Association for Computing Machinery.

[9] Park, J.-S., Penner, M., and Prasanna, V. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems 15*, 9 (2004), 769–782.

[10] Prokop, H. Cache-Oblivious Algorithms.

[11] Valgrind™ Developers. Cachegrind: a cache and branch-prediction profiler. https://valgrind.org/docs/manual/cg-manual.html.

[12] Weisstein, E. Strassen Formulas. https://mathworld.wolfram.com/StrassenFormulas.html.